A Conceptual Framework for Constructing Distributed Object Libraries using Gellish

Michael Rudi Henrichs

Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
A Conceptual Framework for Constructing Distributed Object Libraries using Gellish

Master’s Thesis in Computer Science

Michael Rudi Henrichs
mikehenrichs@idoro.eu

Parallel and Distributed Systems group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

June 1, 2009

Student
Michael Rudi Henrichs
Studentnumber: 9327103
Oranjelaan 8
2264 CW Leidschendam
mikehenrichs@idoro.eu

MSc Presentation
June 2, 2009 at 14:00
Lipkenszaal (LB 01.150), Faculty EWI, Mekelweg 4, Delft

Committee
Chair: Prof. Dr. Ir. H.J. Sips h.j.sips@ewi.tudelft.nl
Member: Dr. Ir. D.H.J. Epema d.h.j.epema@ewi.tudelft.nl
Member: Ir. N.W. Roest n.roest@idoro.eu
Supervisor: Dr. K. van der Meer k.vandermeer@ewi.tudelft.nl
Preface

This thesis is the result of my project work for the Masters degree in Computer Science at the Department of Software Technology, Delft University of Technology. During the last two years, I have worked on this research project, while also working as a developer for Idoro and Warehousematch in Waddinxveen.

The completion of this project did not happen overnight. This was partly due to the enjoyment of developing new types of applications for Idoro. As a consequence understanding the core principles of Gellish especially benefitted greatly from these activities at Idoro, and I do believe that the results of this research project, in turn will be beneficial for Idoro.

Acknowledgement

The completion of this research project would not have been possible if it wasn’t for a number of people. Therefore I would like to take a moment and thank all those people, who supported me during the project and writing of this thesis.

First and foremost I would like to thank my thesis advisor Dr. Kees van der Meer, for his feedback, and for keeping me focused during this project. Thank you to Nic Roest, for if it wasn’t for him I wouldn’t have taken up this research project. Our discussions and work together have greatly influenced this thesis. I would like to thank Andries van Renssen for his support and readiness to help me better understand Gellish and see its potential. I would also like to express my appreciation to the other committee members, Prof. Dr. Henk Sips and Dr. Dick Epema.

This research project would not have been possible without support from COINS, CROW, STABU and USPI-NL, thanks to all attendees from these organizations for their comments and feedback during the various presentations in the course of this project.

Finally I would especially like to thank my family and friends who have supported and encouraged me to finish this.

Michael Henrichs

Waddinxveen, April 2009
Abstract

This master's thesis describes the processes concerned with, and motivations behind the construction of a framework for building distributed object libraries using Gellish. This framework, together with the applications implemented with the goal of demonstrating and managing the capabilities of the distributed object library, is designed to provide users the capability of sharing and reusing information in a distributed fashion.

Although a large number of ontology modeling languages exist, the problem of mapping knowledge from one modeling language onto another was placed outside the scope of this project, which resulted in Gellish being selected as the modeling language used to conceptualize the information stored at the distributed data stores. The need to classify ontology modeling languages, based on their subject and structure of conceptualization, resulted in the construction of the conceptualization graph. This graph is used to compare ontology modeling languages used within the building and construction industry, but also to motivate the aforementioned choice for Gellish as the modeling language used throughout the distributed object library.

With the distributed object library framework and associated applications constructed, the final part of this thesis will conclude with the identification of limitations, based on tests performed with these applications, and present recommendations for improving the framework and applications where needed.

Keywords: distributed object libraries, distributed information, conceptualization graph, classifying modeling languages, Gellish
Contents

Preface and Acknowledgement v
Abstract vii
List of Tables xiii
List of Figures xvi

1 Introduction 1
  1.1 Problem Statement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
  1.1.1 The Building and Construction Industry . . . . . . . . . . . . . . . . . . 1
  1.1.2 Technical Improvements . . . . . . . . . . . . . . . . . . . . . . . . . . 3
  1.1.3 Systems, Standards and Projects . . . . . . . . . . . . . . . . . . . . . . 4
  1.2 Research Questions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
  1.3 Project Scope . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
  1.4 Thesis Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

2 Modeling Languages 7
  2.1 Object Libraries . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
  2.2 Conceptualization Graph . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
    2.2.1 Structure of Conceptualization . . . . . . . . . . . . . . . . . . . . . . 8
    2.2.2 Subject of Conceptualization . . . . . . . . . . . . . . . . . . . . . . . 10
    2.2.3 Comparing Languages . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
  2.3 Candidate Languages . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
    2.3.1 OWL . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
    2.3.2 ISO 10303 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
    2.3.3 ISO 12006 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
    2.3.4 IFD-LexiCon . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
    2.3.5 ISO 15926 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
    2.3.6 Gellish . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
  2.4 Selecting Gellish . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
    2.4.1 Definition Gellish . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
    2.4.2 Gellish Queries . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

3 Requirements and Design 21
  3.1 Requirements Elicitation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
    3.1.1 Functional Requirements . . . . . . . . . . . . . . . . . . . . . . . . . . 21
    3.1.2 Technical Requirements . . . . . . . . . . . . . . . . . . . . . . . . . . 24
    3.1.3 Other Requirements . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2</td>
<td>Design Goals</td>
<td>25</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Interoperability</td>
<td>25</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Extensibility</td>
<td>25</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Readability</td>
<td>26</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Usability</td>
<td>26</td>
</tr>
<tr>
<td>3.2.5</td>
<td>Response Time</td>
<td>26</td>
</tr>
<tr>
<td>3.3</td>
<td>Subsystem Overview</td>
<td>26</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Overall Structure</td>
<td>26</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Gellish Data Subsystem</td>
<td>28</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Peer Tracker Subsystem</td>
<td>28</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Engine Subsystem</td>
<td>30</td>
</tr>
<tr>
<td>3.3.5</td>
<td>DOL Manager Subsystem</td>
<td>31</td>
</tr>
<tr>
<td>4</td>
<td>Implementation</td>
<td>33</td>
</tr>
<tr>
<td>4.1</td>
<td>Programming Language</td>
<td>33</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Microsoft .NET Framework</td>
<td>33</td>
</tr>
<tr>
<td>4.2</td>
<td>Developer Tools</td>
<td>36</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Microsoft Visual Studio</td>
<td>36</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Subversion</td>
<td>37</td>
</tr>
<tr>
<td>4.2.3</td>
<td>SQL Server Management Studio</td>
<td>37</td>
</tr>
<tr>
<td>4.3</td>
<td>Persistent Data</td>
<td>37</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Gellish Table Format</td>
<td>38</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Database Systems</td>
<td>39</td>
</tr>
<tr>
<td>4.3.3</td>
<td>XML Schema</td>
<td>40</td>
</tr>
<tr>
<td>4.3.4</td>
<td>XML Document</td>
<td>47</td>
</tr>
<tr>
<td>4.3.5</td>
<td>GMZ Files</td>
<td>47</td>
</tr>
<tr>
<td>4.3.6</td>
<td>SOAP Messages</td>
<td>48</td>
</tr>
<tr>
<td>4.3.7</td>
<td>Application Data</td>
<td>48</td>
</tr>
<tr>
<td>4.4</td>
<td>Constructing the Framework</td>
<td>48</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Naming Conventions</td>
<td>48</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Class Libraries</td>
<td>50</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Service Libraries</td>
<td>53</td>
</tr>
<tr>
<td>4.4.4</td>
<td>Applications</td>
<td>57</td>
</tr>
<tr>
<td>4.4.5</td>
<td>Assembly Versions</td>
<td>59</td>
</tr>
<tr>
<td>5</td>
<td>Evaluation of Framework</td>
<td>61</td>
</tr>
<tr>
<td>5.1</td>
<td>Evaluation Setup</td>
<td>61</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Gellish Datasets</td>
<td>61</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Network Layout</td>
<td>62</td>
</tr>
<tr>
<td>5.2</td>
<td>Testing the Framework</td>
<td>63</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Validating Distribution</td>
<td>63</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Exploding Down</td>
<td>64</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Distributed Queries</td>
<td>65</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Protected versus Public</td>
<td>65</td>
</tr>
<tr>
<td>5.3</td>
<td>Limitations Prototype</td>
<td>65</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Improved Protocol</td>
<td>65</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Gellish Interpretation</td>
<td>66</td>
</tr>
</tbody>
</table>
6 Conclusions and Future Work

6.1 Conclusions

6.2 Recommendations

6.3 Future Work

Appendices

A Gellish Table Column Definitions

B SQL Script

C Gellish XML Schema

D Sample Gellish XML Document

E Installatie en Basis Handleiding

E.1 Installatie DOL Prototype

E.1.1 Microsoft .NET Framework Vereist

E.1.2 DOL Engine Service

E.1.3 DOL Peer Tracker Service (optioneel)

E.1.4 Activeren van een Windows Service

E.1.5 DOL Manager (optioneel)

E.1.6 Demo Applicaties

E.1.7 Port-forwarding Instellen

E.1.8 Deinstallatie Componenten

E.2 DOL Manager Quick Start Guide

E.2.1 Schermopbouw

E.2.2 Peer Tracker Beheren

E.2.3 Engine Beheren

E.2.4 Gellish Feiten Bekijken

E.2.5 Gellish Queries Uitvoeren

E.3 Demo Environment Setup

E.4 Demo Definition Viewer
List of Tables

4.1 Column names used in the SQL implementation of a Gellish table. Each name is accompanied by the used data type and whether the values stored in the respective column may remain null. ................................................. 41

4.2 Capitilization rules used to name the different identifiers. Note: Interface will always be prefixed by "I" and exception classes will always be suffixed by "Exception". ................................................................. 49

4.3 Overview of the operations defined in the IEngine contract, and implemented in the EngineService class. ........................................................................................................... 55

4.4 List of operations found exposed by the PeerTrackerService class, as defined by the IPeerTracker contract. ......................................................................................... 57

4.5 Version numbers, and dates of the most recent releases of libraries and applications. ......................................................................................................................... 60

5.1 Results from separating the GWB Riolering (GWBR) dataset into three non overlapping datasets, based on the relation type found in each Gellish fact. .................. 62

A.1 Gellish column definitions, including the SQL and XML datatypes and additional constraints (taken from Van Renssen’s Definition of Gellish Databases [VR08]). .................................................. 82
List of Figures

2.1 Scheme for classifying ontologies based on the structure and subject of conceptualization, by Van Heijst and colleagues [VHSW97]. .......................... 8

2.2 Classification of ontologies according to McGuinness [McG02]. The mereotopology relation is separated from the inverse and disjointness structure, originally combined into one by McGuinness. .................................................. 8

2.3 The different subjects of conceptualization stacked on top of each other to form a pyramid. .............................................................. 11

2.4 Conceptualization graph with five different ontology languages, A to E, located within it. .............................................................. 12

2.5 Conceptualization graph containing a number of ontology modeling languages used in the B+C industry. ........................................... 17

2.6 A simple request and confirmation of "P-1" being classified as a "pump". .. 18

2.7 Gellish queries can also contain questions about unknowns, in this case the question is asked to return any classifications of "pump" (if they exist). .. 18

2.8 When constructing an answer a DOL Engine should take into account inheritance between concepts, like the fact that when searching for a "pump", a subtype of "pump" will also do, like in this case a "centrifugal pump". .. 19

2.9 Questions can also contain multiple unknowns, for instance when looking for both classifications of "pump" and "compressor" at the same time. .... 19

2.10 Additional constraints can be added to a question about unknowns, in this case classifications of "pump" are requested, which also have a "capacity" of at least "30 dm$^3$/s." ......................................................... 20

2.11 Additional Gellish fields can be used to indicate the source of the question and answer. .............................................................. 20

3.1 Overview of the communication and layout of different engine and trackers (see Scope and Vision document). ........................................... 27

3.2 An overview of the four different subsystems of the distributed object library working together. ......................................................... 28

3.3 Decomposition of the peer tracker subsystem into the higher level parts, including the flow of interactions inside and outside the subsystem. .... 29

3.4 Detailed version of the engine subsystem, which identifies the main parts of the subsystem, as well as the interaction with other subsystems. .... 30

4.1 Visual overview of the Common CLR Language Infrastructure, and how the components relate to each other (from http://en.wikipedia.org). .... 34

4.2 Overview of the different .NET frameworks illustrating the main components of each of these versions of the framework. ......................... 35
4.3 Screenshot of the source code editor included within the Microsoft Visual Studio developer environment.

4.4 Screenshot taken from the SQL Server Management Studio, while viewing a table containing Gellish facts.

4.5 Class diagram of the Gellikx class library. The classes within these namespaces are the most generic classes within this project.

4.6 Class diagram of the Gellikx.Gellish class library. Classes within these namespaces are used to retrieve, represent, and query Gellish tables.

4.7 Class diagram of the Gellikx.Dol.Windows.

4.8 Class diagram of the Gellish class library (Gellikx.Dol.Engine). The classes within these namespaces are the most generic classes within this project.

4.9 Class diagram of the Gellish class library (Gellikx.Dol.PeerTracker). The classes within these namespaces are the most generic classes within this project.

4.10 Screenshot of DOL manager application.

4.11 (left) Main interface of the demo definition viewer, used to search for objects. (right) Definition window presenting a single integrated result from a distributed query based on the selected object.

4.12 Example query for retrieving the discriminating aspect compared to a supertype of a selected object.

4.13 Screenshot of demo DOL setup application, used to configure and run multiple engines on the same machine.

5.1 Setup of multiple engines ($E_0$ through $E_3$) and peer trackers ($T_A$ and $T_B$) used during the evaluation of the framework.

E.1 DOL Engine Service setup venster.

E.2 Stap in DOL Engine setup waarmee de lokatie op uw computer kan worden opgegeven en de gebruiker accounts voor wie de installatie van toepassing is.

E.3 DOL Engine setup stap voor het opgeven van de port waarop de service zal luisteren.

E.4 Windows Services venster, voor het beheren van Windows services.

E.5 Setup gebruiken voor het deïnstalleren van DOL services.

E.6 Windows venster voor het toevoegen of verwijderen van applicaties.

E.7 Schermopbouw DOL Manager applicatie.

E.8 Openen van het Peer Tracker snelmenu.

E.9 Venster voor het wijzigen van de naam voor de peer tracker.

E.10 Nieuwe peer tracker toevoegen aan engine.

E.11 Venster voor het ontdekken van nieuwe engines.

E.12 Venster voor het toevoegen van een Gellish datastore.

E.13 Nieuwe publicatie toevoegen.

E.14 Venster voor het aanvragen van abonnementen op publicaties.

E.15 Sorteer mogelijkheden Gellish fact viewer.

E.16 Selectie menu met daarin de Gellish subset varianten die worden ondersteund in de DOL Manager fact viewer.

E.17 Gellish fact viewer venster.

E.18 Hoofdvenster DOL Environment setup applicatie.

E.19 Venster voor het toevoegen van een nieuwe engine.

E.20 Hoofdvenster Demo Definition viewer.

E.21 Voorbeeld van een soortenboom.

E.22 Definition venster Demo Definition viewer.
Chapter 1

Introduction

This first chapter will kick off with presenting the problem statement, which will be based on the observations made in section 1.1. Using these observations, section 1.2 will formulate two research questions, which will be answered during this research project. Next section 1.3 will set forth a project scope, including the limitations of the project. Finally the last section of this chapter will give an overview of the remainder of this thesis.

1.1 Problem Statement

Observations discussed in this section originate from two different fields. The first observation is concerned with the information reuse of companies within the building and construction industry (B+C industry). Whereas the second has to do with the technical improvements enjoyed by computers, and the (inter)networks connecting them.

The reason for looking at the information reuse within the B+C industry is partly due to the fact that this project is sponsored by actors active within this industry. Also, as was pointed out by Van Rees [VR06], cooperation between actors of the B+C industry, based on the information exchange and reuse, is an area that could benefit greatly from new ICT usage.

1.1.1 The Building and Construction Industry

Operating within the B+C industry is a large number of different actors, ranging from architects and designers to contractor and maintenance personnel. All these actors need to access information stored in one form or another, but more importantly they need to be able to share this information with one another. As discussed by Van Rees the sharing of information between actors within the B+C industry, using new IT solutions, has been faced with various problems and is still lagging compared to other industries.

As illustrated above there is a need within the B+C industry to share and reuse information concerned with building objects, during the various phases of their lifecycle. Nowadays this type of information is mostly stored digitally within documents, electronic drawings, and other types of files or databases. However there is not a single (controlled) database for actors within the industry to use as basis for collaboration [VR06].

Due to the absence of a single repository, a number of actors have constructed their own object library, which only contains the knowledge of the actors’ domain of interest. In this
Chapter 1. Introduction

case the domain of interest is limited to the lifecycle phases of the building objects, for which
the actor is responsible. Section 1.1.3 discusses the drawbacks of these projects.

These libraries are constructed using different ontology modeling languages. The next
chapter will cover these modeling languages and standards, used by actors within the B+C
industry. Also in order to compare these languages, a classification scheme will be con-
structed, which will allow modeling languages to be compared based on their structure and
subject of conceptualization.

The fragmented nature introduced above is also observed by the Dutch Technical Agree-
ment of the NTA 8611 study group [NTA07]. The goal of this group is to give recommen-
dations for improving integration between existing object libraries, while adhering to ISO
guidelines. The following problems are taken from Guidelines for Object Libraries [NTA07],
written by the NTA study group, and highlight the difficulties surrounding the integration
of existing object libraries, owned and maintained by actors within the B+C industry.

Fragmented Developments

Within the B+C industry organizations are working independently on developing object
libraries, because they are unaware of initiatives currently being undertaken by others. Cur-
rently no overview exists listing these initiatives, which could be used to promote the inte-
gration of libraries.

Resistance to Migration

According to the NTA study group organizations exhibit resistance towards migrating to a
large object library [NTA07]. Main reason for this is, differing definitions of concepts and
objects, and as a consequence, libraries which are hard to link.

Quality of Library Content

The quality of content stored within an object library is dependent on the competences of
the users providing and storing this knowledge. When the representation is not correct, or
perceived by other actors not to be correct, the library will not be used, and other solutions
will be sought.

Different Modeling Languages

Currently a number of different ontology modeling languages are used to model the struc-
ture and content of an object library. Examples of ontology modeling languages primarily
found within the B+C industry, are Lexicon, STEP, Gellish, ISO 15926, etc. (see section
2.3). There is not a single best language, and the choice for selecting a particular language
is difficult to analyze. In addition the NTA study group states that an ontology modeling
languages needs a certain maturity and critical mass, before being adopted by more actors.

The fragmentation of object library initiatives, may also result from the organizational
fragmentation within the B+C industry, as discussed by Van Nederveen. There are two
competing, or dual, forces at work here: opposite goals of competition versus cooperation
on common interests [VN00].

By sharing and reusing the information, created during the different lifecycle phases of
building objects, cooperation between actors within the B+C industry could, however the
fragmented structure of the industry and difficulty of integrating object libraries presents a
problem.
1.1.2 Technical Improvements

A second observation is concerned with the increased computer power, and risen popularity of distributed and service oriented applications. This increase in popularity is partly due to the improved connectivity between users in the form of broadband (inter) networks connecting their systems.

Distributed Applications

One characteristic of a distributed application is connecting users to a single coherent system in a transparent, open, and scalable fashion, while providing access to shared resources. Each computer, or node, within a distributed application can be located at a different geographic location, while connected to others via an interconnecting network. A common misconception is that a distributed system is a synonym for a network of computers [NdAaB06]. Although the distributed system can be build on top of a network of computers, the system should hide this fact, and act as a single system to its users.

In the last decade distributed applications, especially peer-to-peer applications, have become very popular for sharing files [SKGG02]. However the files shared using these applications is restricted to software, and multimedia files. This research project will not focus on these types of files, but on the information stored within object libraries.

Transparency

Transparency will be the most important aspect of distributed applications during this project. Transparency is concerned with hiding the fact that a user is using multiple, possibly heterogeneous, computing elements, which can be physically distributed across a network. As described in [TVS01], transparency can be applied to several aspects of a distributed system, various forms of transparency can be found in the ISO 10746-3 standard [ISO96].

Access transparency hides the heterogeneity of the different distributed sources. As a result the different machines within the system will be able to interoperate, despite differences in data representation [NdAaB06]. For the reuse of information within the B+C industry this could mean that actors, sharing their information through the distributed application, can use their own modeling language.

Another type of transparency is location transparency, which hides the actual location of a piece of information. If there is no need to know the physical location of a particular resource, location transparency can be used to hide it. However complete location transparency is not always preferable, for instance when retrieving contradictory information from different users, the receiver will want to know where this information originated from in order to evaluate which part(s) of the information can be perceived to be correct. Thus total location transparency is not always preferable.

Transparency will give users of a distributed object library the idea that they are using a single object library while accessing the information, when in actuality parts of the library are geographically distributed, maintained by different actors, and conceptualized using different ontology modeling languages.

Challenges of Distribution

In contrast to advantages gained by the forms of transparency offered by distributed application, a framework for constructing distributed object libraries will also need to address a number of issues resulting from the distribution of information.
For instance, because information is distributed and maintained by different actors, any information retrieved from the distributed object library could be a combination of data stored at different sources. Therefore users of these applications should be able to check this information for consistency, completeness and correctness, in a timely fashion.

Just as with central database solutions, authentications and authorization is needed to guarantee that information is only retrieved and viewed by users who are allowed to do so. However, since a central authority is absent, the distributed nodes will themselves be responsible for managing the access to the information they provide.

1.1.3 Systems, Standards and Projects

Currently, there are a number of standards and applications used to model, view and retrieve information stored within object libraries. By observing the characteristics of the initiatives undertaken by actors within the B+C industry, and based on discussions with CROW and STABU, it is clear that a number of projects are being undertaken which are aimed at combining or integrating data from different data repositories, and making them available through a single interface.

Examples of these projects are the development of the Basismodule Cheobs, undertaken by CROW [Che08], as well as the international IFD Library for buildingSMART. Both these systems take advantage of web services, however these services are located on a single web server. The drawback of these systems, therefore, is that they are still based on a central repository and controller. A problem of such a central repository is that this central point needs to be capable of serving all incoming requests for information stored at this central repository. Due to this load, the central repository can become a bottleneck, and a system failure will result in clients not being able to access any of the information.

Another problem of a single repository is the fact that all actors, willing to share the information of their object library, will have to store this information on the central system. If any modifications or updates are needed, these have to be transferred to the central repository in a controlled fashion. Also some actors might not appreciate the fact that their object library is stored on a public server, especially if this information contains sensitive, or licensed information, for which access should be restricted to trusted users.

A distributed network of data stores solves the issues discussed above: no central bottleneck; actors can easily modify their own (local) data stores; and each actor can restrict access to the information found in their data stores, based on the credentials provided in a request.

1.2 Research Questions

Section 1.1 illustrated the fragmented nature of object libraries and their developments within the B+C industry in the Netherlands. This fragmentation has resulted in a plethora of different ontology modeling languages, object libraries of varying quality levels, and as a result too few possibilities for integration [NTA07]. However, current improvements in computers and (inter)networks, including the benefits of distributed applications also presented in the previous section, promises a solution to overcome this fragmentation.

To increase cooperation between the different actors in the B+C industry in terms of information exchange and reuse, a distributed object library will be constructed, which offers its users a system capable of sharing information in a coherent and transparent fashion. Each

---

1http://dev.ifd-library.org/index.php/Main_Page
actor will still be able to store their specific lifecycle information, however knowledge can be shared using this Distributed Object Library (DOL) system.

As a result the following two questions will be the focus of this research project.

1. The B+C industry currently uses a number of different ontology modeling languages, each capable of representing information of a particular structure, and capturing a different level of detail in their conceptualization. How can these modeling languages be categorized to reflect these conceptualization characteristics.

2. Taking into account the requirements of the B+C industry to share information, combined with the advantages offered by distributed applications, is it possible to construct a framework on which distributed object libraries can be built. Also what are the limitations of using this framework for sharing information, stored at the distributed data stores.

1.3 Project Scope

As can be seen from the two presented research questions, the B+C industry has been taken as a starting point of this research project. However the solutions given and implemented framework will be applicable for other industries, which require its actors to share information in a distributed fashion.

Goal of this project is to construct a distributed object library, capable of offering its users the capability to share information, and identify possible problems and challenges faced when building. However the translation of information, represented by one ontology modeling language into a representation in another language will be outside the scope of this project. Although the translation will be outside the scope, the classification scheme developed in chapter 2, can be used to analyze the possibility of these translations. The reason for excluding the translation from this project is the fact that the focus will be on developing a distributed object library application framework, and although such translations can be very problematic [NTA07], they are assumed to be possible.

The second research question above refers to the implementation of a framework for constructing object libraries on. This distinction comes from separating the distributed object library system into two tiers. The lower of these two, which will be called the framework, is responsible for storing and retrieving the information, and for performing the operations needed within the distributed object library. Applications, on the other hand, will operate as the front-end for the framework, by accepting input from its users, communicating with the framework, and displaying results received from the framework.

By separating the framework (or back-end) from the applications (the front-end), the distributed object library can be built in a modular fashion, whereby the framework will be required to accept instructions from an applications and return any results back to the application. This separation also allows applications with different interfaces to be built on top of the same framework. Examples of these different types of applications, constructed on top of the framework, will be given in more details in chapters 3 and 4.

1.4 Thesis Overview

Chapter 2 starts off by defining what an object library is and what they are used for (section 2.1). Following this definition, section 2.2 presents the conceptualization graph and illustrates how it was constructed and can be used. Armed with this conceptualization graph, section
2.3 places various ontology modeling languages inside the graph. Section 2.4 motivates, based on the graph given in the previous section, the choice for selecting Gellish as the initial modeling language for the data stores used during this project.

The next chapter begins by summarizing the requirements in section 3.1. These requirements were elicited during the second phase of this project. The second part of chapter 3 takes these system requirements, and gives a high level description of the various subsystems and how they will communicate with one and another (see section 3.3).

Based on the requirements and decomposition of subsystems, presented in the previous chapter, chapter 4 will be concerned with the actual constructing of the prototype framework. This is done by first considering the appropriate programming language, as well as off-the-shelf components and packages (section 4.1). Next the tools of the trade are briefly introduced in section 4.2, which is followed by the persistent data 4.3 and the various class libraries and applications in section 4.4.

Once the various libraries and applications have been constructed, chapter 5 will take them for a test drive. First the test setup is presented in section 5.1, including a description of the data stores used during testing. Next section 5.2 will discuss the actual testing and report the acquired results. Chapter 5 is concluded in section 5.3 with a discussion on how the framework and associated applications performed during testing, and highlight possible limitations.

Finally chapter 6 revisits the initial research questions. Sections 6.2 and 6.3 give recommendations and discuss future developments needed, if the components within the framework and the applications built are to be successfully deployed within the B+C industry after this research project has ended.
Chapter 2

Modeling Languages

Within this chapter a number of ontology modeling languages will be presented, and compared based on the structure and subject of their conceptualization. The result of this comparison will be used to motivate the selection of a single language, which will be used in the distributed object library. However before presenting this comparison the next section will first give a brief definition of an object library, followed by the construction of a conceptualization graph in section 2.2.

2.1 Object Libraries

An object library is a collection of information, which defines the types of objects, or concepts, that exist in a particular domain. This type of information can include information defining the concepts within the library, their properties, relationships with other concepts, as well as constraints posed upon these concepts, properties, and relationships. Not all object libraries will contain these characteristics to the same degree, as will be shown in the next sections.

Another name for an object library stemming from the field of philosophy is an ontology, which is a combination of two Greek words, which when put together mean "the study of being". A translation from the real world to an ontology is called a conceptualization. The result of such a conceptualization are the objects, concepts and entities, their properties, together with the relations between them[Grud93]. The structure and subject of such conceptualizations varies, and depends on the intended use of the information stored within the object library.

For the purpose of this research project, this collection of information is assumed to be stored digitally within files and/or databases (see section 4.3).

2.2 Conceptualization Graph

The goal during the first phase of this project is to be able to classify ontology modeling languages used within the B+C industry, however as discussed by Gómez-Pérez, a large number of classification schemes exist for classifying ontology modeling languages [GPFLC04].

For the purpose of this project a classification scheme was constructed, which can be used to classify ontology modeling languages according to the relative structure and subject resulting from their conceptualization. By applying this conceptualization graph to ontology modeling languages used by the actors within the B+C industry, it is possible to inspect the
Chapter 2. Modeling Languages

Figure 2.1: Scheme for classifying ontologies based on the structure and subject of conceptualization, by Van Heijst and colleagues [VHSW97].

Figure 2.2: Classification of ontologies according to McGuinness [McG02]. The mereotopology relation is separated from the inverse and disjointness structure, originally combined into one by McGuinness.

The possibility of mapping a conceptualization, given by one ontology modeling language, onto another modeling language. An example of this graph, including the candidate modeling languages used in the B+C industry, can be seen in figure 2.5.

The conceptualization graph was constructed by modifying one of the dimensions of the categorization scheme, presented by van Heijst and colleagues [VHSW97]. Figure 2.1 shows this classification scheme. The modification was concerned with the structure of the conceptualization, which is replaced with the classification scheme given by McGuinness [McG02] (see figure 2.2). Both the structure and subject of conceptualization are discussed in the next sections.

2.2.1 Structure of Conceptualization

The first dimension is based on the spectrum given by McGuinness in [McG02], which can be used to classify different ontology modeling languages according to the structure of conceptualization, which can be represented by the modeling language. Figure 2.2 illustrates this linear spectrum of the possible structures. Each step on this line extends, or has a broader structure, than the previous step, and allows for a more expressive ontology. Although McGuinness combines the last two nodes in her classification, the conceptualization graph used in this research project will separate the mereotopology (part-of) relationship from the inverse and disjointness relations. Each of the structures shown in figure 2.2 are discussed below.

Controlled Vocabulary  This first structure of conceptualization results in the least expressive type of object library (or ontology). A controlled vocabulary (also known as a nomenclature) is concerned with the notation, or syntax, of the terms found in a particular domain of interest. Each term, or descriptor, found in a controlled vocabulary is used to tag a single unit of information, and consists of a single word of a complete phrase.
2.2. Conceptualization Graph

**Dictionary**  Just as a controlled vocabulary, a dictionary (or glossary) consists of a list of terms found in a domain of interest. In contrast with a controlled vocabulary, a dictionary also gives a meaning, or definition, to each term found in the object library.

**Thesaurus**  Similar to a dictionary the thesaurus is a list of terms with their definition. Within a thesaurus this definition is often called a *scope note*, which describes how the term should be interpreted. Also each entry found in the list is accompanied by a list of *related terms*, which can be synonyms, near-synonyms and antonyms. An entry can also contain a list of *broader* and *narrower terms*, which describe more general or specialized terms respectively. Finally some thesauri also contain *use* and *used for* indicators when a term can be used for another term.

**Informal is-a Hierarchy**  This structure of conceptualization offers a hierarchy to the concepts, or classes, found within a domain of interest. The resulting hierarchy is based on an informal *subsumption* - or *specialization* relationship, which is often called the *is-a* relationship. McGuinness classifies this relationship as informal because concepts related through this relationship are not always strict hypernyms and hyponyms. An example given by McGuinness, is the classification scheme used by the Yahoo! search engine. The general category *clothing*, found within the Yahoo classification, includes a subcategory *women*. This subcategory further contains the subcategories *dresses* and *accessories*. Although each type of *dresses* is also a type of *clothing*, a dress is not a type of *women*. As a consequence this structure of conceptualization cannot be used for deductive purposes.

**Formal is-a Hierarchy**  Also known as a *taxonomy*. This type of library is similar to a previous classification, however in a taxonomy the relationship used to group concepts together, is based on the strict subsumption relationship. An example of such a relationship is a *flower* which is a subclass of *plant*, and *plant* is the superclass of *flower*. Compared with the informal is-a hierarchy, this hierarchy can be used for deductive purposes, and allows for exploitation of inheritance between classes within the object library.

**Formal Instance**  A formal is-a hierarchy can be further extended to include formal instances. Compared with hierarchies that don’t include this extension - and only allows for the use of classes - these extended hierarchies also support the creation of instances.

**Frames**  This structure extends the taxonomy in such way that the information of a particular class, can be extended with properties. For example, each instance of the class *pipe* has the property *diameter*, in the case of instance *P-5* this property can have the value 5 cm. Because of the hierarchical structure resulting from the specialization relation, this property, associated with the class *pipe*, is also inherited by all specializations of *pipe*.

**Value Restrictions**  In the previous structure values can be attached to the properties classes, however no restrictions were imposed on the valid values for these properties. The *diameter* property of the class *pipe*, given in the example above, could also have the value *red* assigned to it. A further extension thus, is realized by placing restrictions on both the domains and ranges of values allowed for each property, i.e. the *diameter* of a *pipe* can only have positive centimeter values.

---

1 When class A is a superclass of B, and B is a subclass of A, then A is a hypernym of B, and B is the hyponym of A.

2 [http://www.yahoo.com](http://www.yahoo.com)
Chapter 2. Modeling Languages

**General Logic Constraints**  Similar to the previous restrictions placed on the values of a property, logic constraints allow for more complex logical statements to be used. These statements may contain restrictions which depend also on other properties found in the same, or different classes.

**Mereotopology**  Within this structure classes can be constructed by composing other smaller classes together. Compositions can be created by using the mereotopology relationship. Although McGuinness does not represent the mereotopology as a separate node within her linear spectrum, this structure is separated from the disjointness and inverse structure. The reason for explicitly separating this structure is because the B+C industry uses this type of relation when constructing product models.

**Disjointness, Inverse**  The last node shown in figure 2.2, is labeled with the relationships for disjointness, inverse, and more. This final node contains the remaining set of relationships, which cannot be represented within the previous structures. The reason for grouping all remaining relationship types within this single node, is because the ontologies needed to be modeled within the B+C industry will probably not need such level of detail.

The structures found in this first dimension of the conceptualization graph, are similar to the *functional units* given by the NTA 8611 study group. In their analysis, the group distinguishes between the following functional units: vocabulary, dictionary, taxonomy, knowledge models excluding the mereotopology relationship, and knowledge models including mereotopology relationships [NTA07].

### 2.2.2 Subject of Conceptualization

The second dimension of the classification scheme will be based on the *subject of conceptualization*. Ontologies, or rather the knowledge conceptualized by them, can be organized according to the level of specialized information the concepts and relationships can represent. This characteristic of a modeling language will form the second dimension of the conceptualization graph, and will allow modeling languages to be classified into one of the following four subjects of conceptualization: knowledge representation, upper, domain, and application ontologies.

**Knowledge Representation Ontology**  The most generic subject of conceptualization is realized by the knowledge representation ontologies (KR ontology). The goal of these ontology languages is to give a representation framework, without any claims or extra information about the real world [VHSW97]. Languages within this category can be seen as a meta-meta-language, or structuring language, which offer the modeling primitives used to formalize knowledge. Examples of the primitives found in these types of ontologies are "concepts", "relationships", "properties", etc.

**Upper Ontology**  The concepts and relationships available in the upper ontology languages are very generic, and are common across all possible domains. Examples of concepts found in these languages are "time", "space", etc. Relationships commonly found within these types of ontologies are the specialization, and composition relationships. Other names for these types of ontology languages are generic or world ontologies.
2.2. Conceptualization Graph

Domain Ontology  Domain ontologies contain specialized concepts and relationships from within a specific domain of interest. Examples of domains for which ontology languages have been developed are: medical, engineering, B+C, etc. For this research project, the building and construction domain will be the primary domain of interest.

Application Ontology  The last, most specialized, subject of conceptualization contains concepts and relationships needed to model the knowledge required for a particular application. Examples of different application areas are geometry, cost, and production information of objects.

Another way to look at this second dimension is to place the four types of the subject of conceptualization on top of each other, the result of this can be seen in figure 2.3. By doing so, a pyramid is constructed with the most generic subject of conceptualization on the top, and with increasing specialization of the ontology languages toward the base of the pyramid. The pyramid shape results from the fact that the set of possible concepts, found at each subject, increases in size.

Using the structure of conceptualization, discussed in the previous section, as the x-axis of a graph, and the subject of conceptualization as the y-axis, the conceptualization graph is constructed. An ontology modeling language can now be placed within the graph based on the structure and subject of conceptualization it offers. Note that an ontology language can be placed within more than one subject of conceptualization, since a language can contain concepts, relationships, and properties from more than one level, which results in languages being represented as vertical lines (see figure 2.5). In contrast each language will be placed at the most expressive structure of conceptualization possible.

2.2.3 Comparing Languages

Using the graph constructed in the previous section, ontology modeling languages can be positioned within the figure as vertical lines, depending on the structure and subject of their conceptualization. By positioning an ontology language, relative to other languages, statements can be made concerning the relative loss of expressiveness, and loss of the ability to describe more generic objects. Note that using the conceptualization graph in this manner, gives a rough indication of relative structure and subject of conceptualization, a real
Chapter 2. Modeling Languages

Figure 2.4: Conceptualization graph with five different ontology languages, A to E, located within it.

Take for instance language A and B in figure 2.4, two languages placed at different structures of conceptualization. When B is placed at a more expressive structure, or to the right of language A, the knowledge stored within B cannot be represented in A without loss of expressiveness. However knowledge stored within A can be expressed in B.

A similar comparison can be made by looking at languages with different subjects of conceptualization. Take an upper ontology C and application ontology D, since language D contains more specialized concepts and relationships, more generic concepts and relationships found in C cannot be constructed in language D. However using the more generic concepts represented within language C, the more specialized concepts and relationships of D can be constructed. Although the content of language D can be constructed by C, a language E, which already contains the same concepts, does not need to construct these concepts from scratch.

2.3 Candidate Languages

A number of ontology modeling languages have been placed within the conceptualization graph constructed in the previous section (figure 2.5). This section will give a brief introduction to each of these languages, and discuss why they are placed at the specific subject and structure of the conceptualization they allow.

2.3.1 OWL

Taking DAML+OIL as starting point, the W3C Ontology Working Group defined OWL, the standard accepted, machine-accessible, ontology language of the Semantic Web. OWL uses
2.3. Candidate Languages

the syntax defined in RDF, instances in OWL are presented by RDF descriptions, and most RDFS modeling primitives are used. OWL allows reasoning by (automated) reasoners, e.g. FaCT and RACER.

Different versions exist, each with increasing expressive power (or structure of conceptualization): OWL-Lite, OWL-DL and OWL-Full. However the decidability of reasoning in ontologies developed also decreases because of the size of the model. Both OWL-DL and OWL-Full contain the complete set of constructs possible in OWL, however for OWL-DL the decidability is guaranteed. OWL-Lite lacks a number of constructs, i.e. unionOf, oneOf, and disjointWith, for this reason it will be placed to the left of OWL-Full and OWL-DL in the graph, at the mereotopology structure of conceptualization.

2.3.2 ISO 10303

The ISO 10303 standard, also known as STandard for Exchange of Product model data (STEP), is a very large standard developed by the International Organization of Standardization. STEP is one of the largest ISO standards, with more than hundred parts, and has enjoyed the largest amount of effort ever undertaken by ISO [Pra01]. Although it has often been used successfully [PDE06], its slow development and deployment is still a problem.

The standard is developed for the electronic sharing and exchange of computer-based product model data, it describes the entire life cycle of the product from design to manufacturing. Currently STEP is used to exchange data between CAx applications\(^3\), within different industries, e.g. electrical, architectural, construction, aerospace, automotive, etc.

Each of the following paragraphs gives a overview of the most important parts of the ISO 10303 standard, and their interrelationship for implementing a ontology.

**Part 11: EXPRESS**

The content and structure of a STEP library can be modeled by using the data modeling language EXPRESS (ISO 10303-11). Using this computer interpretable schema, systems are able to verify the syntactical validity of the data models. Apart from the subsumption relation, and entity properties, EXPRESS also uses local and global constraints. These constraints can be added using the WHERE statement.

The structure of conceptualization thus contains general logical constraints, as for the subject of conceptualization, EXPRESS is placed on most generic KR ontology subject level.

**Parts 4x: Generic Resources**

The resources in this part are used to enable interoperability between different AP (Application Protocols), which will be discussed shortly. Since object from different AP’s are mapped onto common generic objects, communication between them is possible, based on these common objects.

For similar reasons as for part 11, parts 4x will be placed at the same structure of conceptualization, however the subject of conceptualization is more specific.

**Parts 2xx: Application Protocols**

The main parts of the STEP standard are the Application Protocols. These AP’s are models of one or more life-cycle stages of a particular domain of interest. Each AP describes the

\(^3\)Applications in which computers are used as aid, e.g. Computer Aided Design, Engineering, and Manufacturing
data structures and constraints of a complete product model [Lof07]. Each AP consists of an Application Activity Model (AAM), which describes the activities in the life-cycle of a product [Lof07]. Also every AP has a top data model, which is called the Application Reference Model (ARM). Objects within the ARM are mapped onto generic objects found in the Generic Resources, which results in an Application Interpreted Model (AIM).

The numbering of these parts is 2xx, e.g. AP225 is an AP for building elements using explicit shape representation, or AP221, which contains functional data and their schematic representation for process plant.

Again the structure of conceptualization is the same as EXPRESS, however the subject of conceptualization is more specific. AP’s should thus be placed at the domain ontology subject level within the graph.

2.3.3 ISO 12006

The name of this ISO standard is Building Construction - Organization of Information about Construction Works. The standard has two main parts, numbered 2 and 3. Each of these parts will be briefly discussed next.

Part 2: Framework for Classification of Information

ISO 12006-2 defines a number of general classes used within construction and facilities management [Ekh05]. These classes cover the complete life-cycle of construction work, without considering the need for interoperability of information- and communication technology, ICT, applications. However when a mapping is needed from classes to generic concepts, ISO 12006-3 can be used.

Part 3: Framework for Object Oriented Information

This part of ISO 12006 specifies a language-independent information model, or meta-model. This model can be used as an upper ontology for the development of other libraries used to store information about construction works. But since it is used for constructing libraries for the B+C industry, it will be placed within the domain subject within the graph. It enables classification systems, information models, object models and process models to be referenced from within a this common framework. For this reason this part of ISO 12006 can be seen as an upper ontology to the building and construction domains, and thus is equal to an domain ontology within the conceptualization graph.

2.3.4 IFD-LexiCon

The LexiCon has been developed to provide a common language for storing and exchanging data between applications, used by participants in construction processes and between owners and users of products resulting from construction activities [Woe00]. The data contained within the LexiCon library describes a particular "Build Object" throughout its life-cycle. The LexiCon can be used as an upper-ontology within the building and construction domains allowing applications, using the library, to refer or derive objects from the object in it. Since LexiCon is used as an upper ontology within the building and construction domains I will place it just inside the upper ontology but for the most part inside the domain ontology. Because the LexiCon consists of a set of concepts, grouped into categories according to ISO 12006-3 [Woe02], each object in the library in mainly concerned with libraries for the B+C industry [VR06].
2.3. Candidate Languages

The LexiCon is the result of the developments on the ISO 12006-3 standard, which was an initiative of STABU\(^4\). During one of the meetings, STABU proposed a "List of Parts", this list was further developed in the Netherlands by the organization BAS\(^5\), and led to the LexiCon library.

Apart from giving descriptions of objects, the LexiCon offers a number of specification properties, e.g. function, quantity, reference. Because of this the structure of LexiCon can contain frames and is found at the frames structure of conceptualization within the graph.

The IFD Library for BuildingSMART project is intended to combine the STABU LexiCon and the BARBi\(^6\) project.

2.3.5 ISO 15926

Also known as Life-cycle Integration of Process Plant Data Including Oil and Gas Production Facilities. The standard currently consists of seven parts, of which parts 2 and 4 will be discussed below.

Part 2: Data Model

ISO 15926 Part 2 can be seen as an upper ontology, it contains 200 generic 4D model concepts [Bat05], that can support all disciplines, industries and life cycle stages. Since it can be used for a number of industries and life cycle stages the subject of conceptualization possible is an upper ontology.

ISO 15926 was developed between 1993 and 2003 by the EPISTLE consortium\(^7\), and is the standardization of the EPISTLE Core Model (ECM). The data model has been written using the EXPRESS language, which was already discussed.

Using this upper ontology data libraries can be built containing information about physical objects, mereotopology, activities and functional requirements. Because the model is four dimensional, it represents concepts both in space, as well as in time. This way it can record the evolution of object, or the life cycle of the concept within the library. Although ISO 15926-2 defines the mereotopology relationship, it does lack the inverse and disjointness abilities of a full ontology, therefore it will be placed in between the two highest possible structures, as was the case for the ISO 10303 parts.

Since the standard contains generic concepts it will be positioned at the upper ontology subject level.

Part 4: Reference Data

This Reference Data is a managed collection of roughly 20000 process plant life-cycle data classes, which are common to many process plants or of interest to many users [Bat05] [EPI07]. These classes, all based on the EPISTLE Reference Data Library (ERDL), are defined by using the Data Model of part 2. For this reason ISO 15926-4 can be found at the same structure of conceptualization as the Data Model, however this Reference Data will be placed at the more specialized domain ontology subject level.

Having developed a generic Data Model, and a Reference Data Library for process plants, it turned out that this subject is already so wide, that actually any state information, from

----

\(^4\)http://www.stabu.org  
\(^5\)Bouw Afsprakenstelsel, a cooperation of producers and users of construction information.  
\(^6\)Bygg og Anlegg Referanse Bibliotek, which is Norwegian for Building and Construction Reference Data Library (http://www.barbi.no/index.jsp)  
\(^7\)EPISTLE is the European Process Industry STEP Technical Liaison Executive, http://www.epistle.ws
any domain, may be modeled with it. ISO 15926 is a standard for data integration, sharing, and exchange between information systems.

2.3.6 Gellish

Gellish, also known as *Generic Engineering Language*, was developed to solve two problems. The first one of these problems was that information is stored in different data models, and as a consequence represented by different system interfaces and/or data structures. An example of this is the EXPRESS data modeling language (ISO 10303-11), discussed in section 2.3.2, which is used to represent STEP data. Another example is that of the UML data model, which can be used independently of the knowledge it represents. As a consequence the exchange of data between systems, each with another data model, becomes non-trivial. However the Gellish language definitions, as well as the knowledge stored within it, can be represented using a single table. This table format is also known as the Gellish Table format [VR08]. Also, as will be discussed in section 2.4.2, a Gellish table can even be used to represent a query used to retrieve specific information from another Gellish table.

Gellish, in this case *Gellish English*, is currently a public domain knowledge base or ontology, called STEPlib and contains concepts from a number of generic data models, some of which were already discussed above: EPISTLE Core Model, ISO 15926-2 and ISO 10303-221. Apart from these general upper ontology concepts, the model is integrated with standard reference data from different sources, like the ISO, IEC and VDI standards, and knowledge from proprietary sources.

The language is, just as a natural language, extensible, and allows for synonyms and multiple names in various languages [VR05]. Because it is possible to extend Gellish with new relationships, without the need to extend the Gellish schema, the structure of conceptualization allowed for a full ontology to be represented. As a consequence Gellish will be placed all the way to the right in the conceptualization graph. Since it contains both application specific concepts, as well as general concepts and relationships, the subject of Gellish covers all four subjects of conceptualization.

2.4 Selecting Gellish

Since this project will focus on the feasibility of building a distributed object library, allowing the distribution of information, the mapping of objects between different ontology modeling languages will be outside the scope of this project. And thus a single ontology modeling is used for conceptualizing the information stored within the distributed object library.

The generic nature of Gellish, resulting from generic concepts and relations already available in Gellish⁹, as well as the high expressiveness due to structure of conceptualization, makes Gellish a prime solution for this project.

A final motivation for initially using Gellish as the ontology modeling language at the distributed nodes, is the ability to formulate Gellish queries using Gellish [VR05] (also see section 2.4.2 below). Because of this all communication between the distributed nodes can be done using a single language: *Gellish*.

---

⁸VDI (Verein Deutscher Ingenieure) is a German engineering network, which defines standards for various technical fields.

⁹These generic concepts and relation types can be found in the TOPini table found in the download section of the Gellish project on Sourceforge (http://sourceforge.net/projects/gellish)
2.4 Selecting Gellish

2.4.1 Definition Gellish

Every row in a Gellish table encodes a single Gellish fact. Such a fact is a representation of an atomic fact or of a proposition about an atomic fact in. The atomic fact is a minimal thing that is the case, and expresses a relation between two things. Each fact can be split into a main atomic fact and zero or more auxiliary atomic facts. For this reason every row in a Gellish table has a number of columns, each recording a particular aspect of a Gellish fact [VR05].

The table arrangement of data is based on a collection of records, or rows, each record representing related data values [EN00]. As presented above, these related values together form a single Gellish fact. Based on the column name, which identifies the location of a data value in the record, the interpretation of that value can be deduced. For instance the integer value found in the column with the name "LeftObjectUid" identifies the UID of the left hand object found in a Gellish fact.

A complete overview of all columns available in a Gellish table, can be found in table A.1 located within the appendices. Section 4.3 will be concerned with mapping a Gellish table onto a digital format, like XML files or SQL databases.

2.4.2 Gellish Queries

The idea of using Gellish to query Gellish has been published by Van Renssen in [VR05]. Gellish thus is not only capable of representing information, but is also capable of representing the questions asked about this information.

Questions can come in two flavors: questions about something which is unknown; and requests which ask for a response, like for instance a confirmation, or a denial. Whereas the distinction between a question and statements can be indicated by a question mark at
Chapter 2. Modeling Languages

the end of a sentence in written English, or by a different tone of a such sentence in spoken English, Gellish uses a designated field or column intention. Examples of valid values for this intention in Gellish English are: a question, a request, an answer, a confirmation, a denial, a promise.

<table>
<thead>
<tr>
<th>Name of left hand object</th>
<th>Intention</th>
<th>Name of relation type</th>
<th>Name of right hand object</th>
</tr>
</thead>
<tbody>
<tr>
<td>P-1</td>
<td>question</td>
<td>is classified as a pump</td>
<td></td>
</tr>
<tr>
<td>P-1</td>
<td>confirmation</td>
<td>is classified as a pump</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.6: A simple request and confirmation of "P-1" being classified as a "pump".

Each of the figures in this section is a graphical representation of a simple Gellish table. The column names are given in the first row of each of the tables shown, all other rows are representations of Gellish facts.

Figure 2.6 shows an example conversation in which two expressions are given. Not all fields or columns known in Gellish are used in these examples, only the ones which are necessary to illustrate the use. When examples such as the one in figure 2.6 are given, the sets of Gellish facts will be prefixed by either a → or ←, which indicates whether the set of facts are part of the question or answer respectively.

In the example of figure 2.6 a simple question and answer dialog is presented, in which the question is asked if "P-1" is classified as a "pump". The answer contains a confirmation of the fact that "P-1" is classified as a "pump".

As discussed earlier questions can also be about unknowns. A question containing unknowns is shown in figure 2.7. Here a question asks for classifications of "pump" using a reserved term "what". Other terms which can be used in Gellish English are: who, which object, which aspect, which person, where, when. The result of the question asked in figure 2.7 is an answer in which "P-1" is classified as a "pump".

<table>
<thead>
<tr>
<th>Name of left hand object</th>
<th>Name of relation type</th>
<th>Name of right hand object</th>
</tr>
</thead>
<tbody>
<tr>
<td>what</td>
<td>is classified as a pump</td>
<td></td>
</tr>
<tr>
<td>P-1</td>
<td>is classified as a pump</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.7: Gellish queries can also contain questions about unknowns, in this case the question is asked to return any classifications of "pump" (if they exist).

As shown in figure 2.9 a question can also contain more than one unknown. Each unknown is identified by an UID taken from the range 1-100, which is a reserved range for unknowns. The example is similar to the previous one, however an additional fact is added asking to also return classifications of "compressor". The answer to this question contains three facts, two classifications of "pump", "P-1" and "P-2", and one classification of "compressor", "C-1".

When a Gellish answer is constructed, the information should not only be checked for matching Gellish facts, but also for facts which are the case based on inheritance. The process of tracing this inheritance in information systems is known as explode down. Figure 2.8 illustrates a data store in which a specialization fact exists between the concept "pump" and "centrifugal pump". Next a question is asked to return any classifications of "pump", as with the previous example "P-1" is returned, but since a "centrifugal pump" is also a kind of "pump", because it is a subtype of "pump", the compressor pump "CP-1" is also returned as a classification of a "centrifugal pump".
2.4. Selecting Gellish

The query mechanism, which will be implemented in the framework, will thus have to extend each of the query facts included in a question. This extension of the original fact, will concern the specializations of the relation type, and both the left and right hand objects.

The following table shows how the query facts can be extended when looking for class names:

<table>
<thead>
<tr>
<th>UID of left hand object</th>
<th>Name of left hand object</th>
<th>Name of relation type</th>
<th>Name of right hand object</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>what</td>
<td>is classified as a pump</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>what</td>
<td>is classified as a compressor</td>
<td></td>
</tr>
<tr>
<td>880101</td>
<td>P-1</td>
<td>is classified as a pump</td>
<td></td>
</tr>
<tr>
<td>870245</td>
<td>P-2</td>
<td>is classified as a pump</td>
<td></td>
</tr>
<tr>
<td>871204</td>
<td>C-1</td>
<td>is classified as a compressor</td>
<td></td>
</tr>
</tbody>
</table>

This extension of the original fact, will concern the specializations of the relation type, and both the left and right hand objects.

Figure 2.9: Questions can also contain multiple unknowns, for instance when looking for both classifications of "pump" and "compressor" at the same time.

More complex question can also contain more additional constraints, which should be checked. The question, shown in figure 2.10, again looks for classifications of "pump", but adds the additional constraint that this classification should have a "capacity" greater than "30 dm$^3$/s"\(^{10}\). Only if a matching fact is found for all facts in the Gellish question, will the answer be returned. It is considered the responsible of the questioner to reformulate its question, when no answer can be returned.

When no UID is known for a particular object, but (part of) the name of the object is known, this name can be used in the question facts in combination with a string commonality parameter. For this purpose Gellish has introduced two string commonality columns: left hand and right hand string commonality. The allowed values for these columns are: case insensitive identical, case sensitive identical, case insensitive partially identical, case sensitive partially identical, case insensitive front end identical, case sensitive front end identical, case insensitive different, case sensitive different, equal, unequal, less than or equal, greater than or equal.

Using the string commonality column in a Gellish question makes it possible to search for names of objects that have a commonality with a specified search string in either the left or right hand object name.

The last section on Querying Gellish covers the use of the author, source and the moment of creation as additional facts to a Gellish English expression. Use of these fields can

\(^{10}\) Complex constraints dependent on the numerical value of an aspect, in combination with relation types concerned with scale, will not be implemented during this research project.
be especially useful within a distributed situation, in which the engine asking a question will probably not be the engine from which the question originated. Similarly the engine supplying the answer to the initial engine, will not be the engine containing the knowledge presented in the answer.

Figure 2.11 illustrates the use of these additional fields. The example again is a request to confirm that ”P-1” is classified as a ”pump”, however the facts also contain the dates, author and source fields. As can be seen the question was created on ”February 25 2008” by ”John Doe”, and similarly the answer found in the ”My Database” source was created on ”February 25 2008” by ”Andrew”.

Figure 2.11: Additional Gellish fields can be used to indicate the source of the question and answer.

All of the aforementioned query scenarios will need to be implemented in the distributed object library framework.
Chapter 3

Requirements and Design

Before discussing the constructing of the prototype framework, this section will enumerate the requirements for the distributed object library framework and applications. These requirements were gathered based on discussions with users of object libraries. Using these requirements a overview will be created of the various parts of the framework and applications in section 3.3.

3.1 Requirements Elicitation

One difficulty in gathering these requirements comes from the fact that the framework will not directly be used by end users. Because these users will use what is called a Gellish liaison application, which is discussed later in this section, the actual requirements for the various components of the framework were constructed based on what would be needed to serve these liaison applications.

All requirements will be organized based on their type. The types of requirements discussed next are functional requirements, technical requirements, and any other requirements which cannot be categorized as functional or technical.

3.1.1 Functional Requirements

During the gathering of the following requirements three groups, or subsystems of the distributed object library framework, were identified. These subsystems will be the Distributed Object Library engine (DOL engine) and DOL peer tracker subsystems, and the Gellish liaison application subsystem.

DOL Engine Requirements

The DOL engine will be responsible for accepting, executing and answering Gellish queries. Each accepted query will be performed on locally attached data stores published through publications, and possibly forwarded to other DOL engines known at the receiving DOL engine. The following enumeration lists the requirements for this particular subsystem.

- A DOL engine can accept queries from, and sent replies to a Gellish liaison application.
- The DOL engine can accept a generic Gellish query, which is domain of interest independent.
Chapter 3. Requirements and Design

- A DOL engine returns a generic Gellish result, which is domain of interest independent.
- A user should be able to add, or remove DOL engines to and from the list of engines.
- Users can inspect the details about a particular DOL engine.
- Users will be able to search for unknown DOL engines using a known DOL peer tracker.
- DOL engines can provide details about themselves, as well as the publications they offer.
- A User will be able to add or remove local data stores to their engine. Removing a data store causes a publication to this store to be automatically canceled (also see Data Store Requirements).
- When a publication has a restricted type, users accessing the knowledge should have acquired a subscription to that publication.
- Users are able to setup, or cancel publications to local data stores.
- Publications can be either public or restricted.
- Subscriptions to a particular publication can be requested (or created) and removed.
- During maintenance activities the DOL engine continues to handle any incoming queries.

The requirements that queries, and the response of these queries, should be domain of interest independent, will be further discussed in the section on the Gellish liaison requirements.

DOL Peer Tracker Requirements

This second subsystem will be responsible for keeping track of the DOL engines available within the network. Thus the peer tracker is essential to constructing the distributed object library. Without a tracker, engines will not be able to discover other engines, and each engine will be merely a single centralized object library.

- Engines are able to connect to a peer tracker, which causes the address of the engine service to be registered.
- Engines can retrieve all known addresses of previously registered engines.
- Engines can unregister from a peer tracker, which removes the engine’s address from the list.

Gellish Liaison Requirements

The goal of a Gellish liaison application is to offer the user of the distributed object library framework a user interface, which communicates with a local DOL engine. One of the main tasks of this user interface, or application, is to translate between the domain of interest of the user and a Gellish question.

- Each Gellish liaison application will offer a domain of interest specific interface for requesting information from DOL engines.
3.1. Requirements Elicitation

- A Gellish liaison application translates domain specific questions into Gellish queries that can be sent to the DOL engine installed locally.

- A Gellish liaison application receives the Gellish query result from the DOL engine and translates it back into a domain specific answer.

In contrast with a DOL engine the Gellish liaison application will allow for domain of interest specific questions to be created by users. For instance a domain of interest specific question could be to retrieve all data on available pumps. The problem of this question is that the qualification "all data" is dependent on the particular use. The task of the liaison is to make this term explicit, but also to provide the UID of the types of pumps referred to within the domain of interest. The end product of this task is a complete Gellish question, similar to the ones presented in the previous chapter.

Error Handling and Extreme Conditions

Each of the following requirements is concerned with the occurrence of (unexpected) errors, and how the framework and applications will deal with them.

- Exceptions that occur during normal execution of the applications need to be caught and handled; either by informing the user of the exception and informing them how to correct them; or by handling them silently.

- Catastrophic termination of the applications should never lead to a loss of data within the data stores; however the user may be required to restart the service and DOL engine.

Security Requirements

Although the primary focus of the framework will not be on security, the framework should be easily extended (also see design goals in the next section) to include these features afterwards.

- Applications will not require the need to check user’s credentials; this responsibility will be passed to the OS.

- Some subscriptions, to restricted publications, could eventually require the user to identify themselves using their credentials.

- Sensitive information, like user credentials, will be sent across using encryption methods.

Quality Requirements

The next set of requirements relate to the retrieval of information using the distributed object library framework.

- All information retrieved from different sources should maintain their integrity.

- Duplicate and identical information may be filtered out by the DOL engine.

- Conflicting results will be highlighted in some way, so that the user may appropriate actions.
Chapter 3. Requirements and Design

System Modification Requirements
There is a single requirement concerned with possible new version of the framework.

- No automatic update scheme will be implemented.

Since one of the goals of this research project is to investigate, and show the feasibility of sharing information in a distributed fashion, using a distributed object library framework, the possibility exists that this version may only be used as a prototype. Therefore updates for this version of the framework are not a requirement.

3.1.2 Technical Requirements
Technical requirements are concerned with the requirements of the Gellish data stores accessed by the DOL engines. Also any requirements of the hardware and software platforms are also presented.

Data Store Requirements
- Queries and their results will be formatted as sets of Gellish facts, in the form of a Gellish table.
- The format of a Gellish table, like the columns used and possible constraints on these columns are based on Gellish Database Definitions.
- Data stores can be represented by both XML files and stored within Microsoft SQL Server tables.

Appendix A contains a table of all Gellish columns. A complete discussing on the construction of the XML and SQL data stores is found in section 4.3.

Hardware/Software Requirements
- If a DOL engine needs to share its publication(s) any firewall between the engine’s computer and the Internet must be configured to allow this communication.
- No configuration changes of a firewall are needed if an engine is only used to access knowledge.
- Both the DOL engine and peer tracker services can be installed to run as a Windows service.
- The application and services will run on the Windows OS, in particular Windows XP.

3.1.3 Other Requirements
This final section lists requirements not yet covered in the previous sections, and contains requirements not directly related to the system. However they are required for a successful completion of this research project.
3.2 Design Goals

User Interface and Human Factor Requirements

• A DOL manager application will be implemented, as a Windows Forms application, allowing users to configure the DOL engine and/or peer trackers installed on their local machine.

• A basic quick-start manual will be included, no further training will be included.

Documentation Requirements

• Inline documentation included in the source code.

• Installation manual.

• Basic quick-start manual.

Resource Requirements

• Users will be responsible for installing the applications on their own systems.

• Administrators will be responsible for installation of the software on dedicated servers.

• Network administrators will be responsible for enabling the network to allow communication between external and internal engines.

3.2 Design Goals

Design goals presented in this section, identify the qualities that the design and implementation of the distributed object library framework and applications should focus on. By determining these goals, before design activities are being undertaken, decisions made during these activities can be made consistently with these goals in mind. The design goals, which will be followed during this project, are based on the groups of criteria presented by Bruegge and Dutoit [BA00].

3.2.1 Interoperability

Interoperability is concerned with the degree to which information systems are capable of working together. Since the core principle behind a distributed object library is the communication, and as a result cooperation between engines and peer trackers, interoperability will be the most important design goal of this project. Also, as discussed in section 1.3, the separation between the framework and actual applications, allows for different types of application to cooperate through the same framework.

3.2.2 Extensibility

This goal is concerned with how easy it is to add new functionality to a developed application. Although it is not possible to envision all future extensions, the impact these changes have on a developed system can be reduced by including structures, which enable these modifications to be made, without the need to completely rewrite the code. An example of such a structure database server agnostic data retrieval, which means that generic code is located in a general class, while specific details for connecting to for instance a SQL database is located in a specialized class, which uses the general class.
3.2.3 Readability

This second design goal is closely related with the extensibility design goal discussed above, and defines how easy it is for other developers to understand the framework by reading the source code. By adhering to known naming conventions, and including inline documentation, for example, future developers will require less time to understand the source code of the distributed object library or its applications.

3.2.4 Usability

In order to focus on the actual framework, the installation and use of the applications should be as easy as possible. For this purpose a basic installation and user manual will be written, also user interfaces will be designed similar to well known programs. The Dutch version of the installation and user manual can be found in appendix E.

3.2.5 Response Time

Response time is both related to the start of a Gellish query and receiving an answer, but also the time spend on accessing facts stored in data stores, or performing queries on these facts. Although no specific requirements are given, algorithms and data structures will be designed to do the work in an efficient manner. Also methods will be executed in an asynchronous fashion, allow multiple actions to be run concurrently, and thus requiring less overall time.

3.3 Subsystem Overview

This chapter gives a decomposition of the distributed object library into the different subsystems. Also dependencies, as well as the flow of events between the components, are shown in the various diagrams throughout this chapter.

3.3.1 Overall Structure

The basis for the communication between "nodes", connected to the distributed object library, was already laid down in the Vision and Scope document. Figure 3.1 illustrates a particular communication scenario. During this scenario a user starts a query from a locally running DOL Manager. This manager application, in combination with an instance of a DOL Engine, is installed on the user’s computer (represented by Node A).

First a Gellish query, constructed by the manager application, is forwarded to the DOL Engine on Node A. This engine will execute the query on any locally attached Gellish data stores, in this case a database containing Gellish facts. Also since "Engine A" is connected1 to peer tracker "Tracker X", it requests a list of known engines also connected to that peer tracker. Using the addresses contained in the response of "Tracker X", "Engine A" forwards the query to "Engine B" and "Engine C". Both these engines execute the query on any locally attached data stores, as well as forward the query again to any other engines which have not yet been addressed (the Gellish message also contains a list with addressees for this reason). Because "Engine C" is also connected to "Tracker Y" it will forward the query also to "Engine D", which was not connected to the first peer tracker.

Once an engine has received a response from all the other engines, it will combine these responses with results obtained by executing the query locally. In the case of our scenario,

1 Connected in this context means that the "Engine A" has made itself known to the peer tracker "Tracker X"; and thus this tracker has added the engine’s address to its list of known engines.
3.3. Subsystem Overview

"Engine D" will only return results from its own datastore to "Engine C". Upon receiving this result, "Engine C" combines it with results retrieved from the locally attached XML file, and replies back to "Engine A". In the meanwhile "Engine B" has queried its locally attached XML file and SQL database, and also replies back to "Engine A". Finally "Engine A" combines the results from its own database, with the responses from "Engine B" and "Engine C", and responds back to the DOL Manager, which display the result on screen for the user.

Note that this scenario presumes that all data stores attached to each engine are created as public publications, and thus will be used in any query received by that engine. However when an engine also has restricted publications only the public publications will be used in queries from other engines. Queries which have been created locally, which is the case for "Engine A" in the scenario above, will not exclude restricted publications. Thus if the database located at "Node A" was published as a restricted publication, it will still have been included by the query executed by "Engine A".

The above scenario has also been worked out in the various use-cases and models presented in the Requirements Analysis Document. However for clarity it has been presented here again.

Based on the components, identified in the aforementioned scenario, figure 3.2 shows a more detailed version of the various subsystems found within the distributed system. The image shows the four main subsystems, each of which will be discussed in the next sections. Also figure 3.2 shows a number of Microsoft .NET libraries, like ADO.NET and WCF, which are used in the construction of the subsystems. These frameworks will be discussed in section 4.1.
3.3.2 Gellish Data Subsystem

In the lower part of Figure 3.2 you see the Gellish Data subsystem. This subsystem will be constructed in isolation of the other subsystems with the primary goal of retrieving and querying Gellish facts. These Gellish facts can be stored in either of the two types of data stores, i.e., XML text files and databases. Although the image only shows a Microsoft SQL server database, it is possible to use any type of database which offers an OLE DB driver.

See Chapter 4 for a complete discussion on the definitions and schemes used to store Gellish facts, as well as the implementation of the different types of data stores. Also the class library containing the components of this subsystem will be discussed in section 4.4.2.

Outside the scope of this project this subsystem will also be made available to other .NET developers in what will be called the Gellish .NET Framework. This development and other future activities will be described in section 6.3.

3.3.3 Peer Tracker Subsystem

As illustrated by the communication scenario at the start of this chapter, the task of the peer tracker will be to offer a kind of white pages service to the DOL engines. Engines will be able to register (or connect) with these trackers and give the address on which they can receive
3.3. Subsystem Overview

request from other DOL Engines. As discussed in the Requirements Analysis Document the trackers (as well as the engines) will run as a Windows service, in the background of the operating system. This Windows service is configured to automatically start when the OS is booted, and shutdown when Windows is shutdown.

Figure 3.3 gives an idea of how this subsystem will interact with the other subsystems. The most important of these interactions is the communication with the DOL Engine. A DOL Engine will initiate this communication through the use of a SOAP message, and therefore is dependent on the .NET WCF Framework (see section 4.1.1). While a peer tracker Windows service is running a Service Host (also a WCF component) is listening on a specific port, which can be seen as an end-point of network communication. Each port is identified by a number and as a consequence each protocol uses a specific number, e.g. port 80 is used for communication between a web browser and a web server. The valid range of number is 0 through 65535, and by default 6120 will be used for the peer tracker service for any communication between an engine and the tracker.

Figure 3.3: Decomposition of the peer tracker subsystem into the higher level parts, including the flow of interactions inside and outside the subsystem.

Once a message is received by the Service Host, for instance the request to return a list of known DOL engines, the Service Host contacts the Controller. Other requests are requests to connect or disconnect to and from a peer tracker. This Controller is the central part of the peer tracker subsystem, which is responsible to retrieve and store application data from and to the file system. Data which is stored on the file systems ranges from a list of connected engines to the settings of the peer tracker Windows service and WCF host. Once the controller has constructed an answer for the engine, it replies to Controller, which in turn sends back a message to the calling engine, via a SOAP message.

The figure also illustrates the interaction of a DOL manager application. The upper arrow in figure 3.3 show the interaction in which the DOL manager retrieves a list of connected engines. This list is used by the DOL manager to display the connected engines within the application.

The second interaction between a DOL manager and the peer tracker subsystems occurs
when the manager application needs to determine the status of a locally installed tracker
or when a user - via the manager application - changes the status of the peer tracker. In
both these situations the tracker subsystem uses a Service Controller, which is based on
components found in the .NET framework\(^2\).

### 3.3.4 Engine Subsystem

Figure 3.4 shows a more detailed version of the engine subsystem. The engine subsystem
has similar structure as the peer tracker subsystem. The engine, once installed on a user’s
machine, will run in the background of the OS as a Windows service.

![Figure 3.4: Detailed version of the engine subsystem, which identifies the main parts of the
subsystem, as well as the interaction with other subsystems.](image)

Again the central component of the engine subsystem is the Controller. Just as the
Controller found in the tracker subsystem, this Controller retrieves and stores application
from and to the file system. However the decomposition of the subsystem also illustrates the
interaction of the Controller with the Gellish framework. As discussed in section 3.3.2, the
Gellish framework subsystem can retrieve Gellish facts stored in XML files or databases.

Figure 3.4 shows the other side of the communication between an engine and a peer
tracker. Instead of using a Service Host to receive a request, an engine uses a Service Client
to send a request to the peer tracker (the Service Client is also used to contact other engine,
as will be discussed below). These requests to a tracker, initiated by the Service Client, can
be either to connect to or disconnect from a tracker, or to retrieve a list of known engines.

The process of sending and receiving of requests, from and to an engine, is shown in the
decomposition figure. Note that the sending of a request is initiated by a Service Client
started from the Controller. In contrast the receipt of a request is handled by the Service
Host, which is part of the engine Windows service. Examples of such request are: execution
of a query on a remote engine; retrieval of a list of publications; requesting a subscription to
a publication; etc.

\(^2\)Objects used for these actions are found in the System.ServiceProcess namespace of the .NET Framework.
3.3. Subsystem Overview

Just as with the communication between a peer tracker and engine, the communication between engines will occur using a specific port number. By default the port number of this communication end-point will be 7464.

Execution of a Gellish query, as discussed in the scenario at the beginning of this chapter, can be started from a DOL manager application. The manager forwards a query, constructed via the user interface of the manager application, to the engine Controller. Next the Controller will retrieve a list from attached Gellish data stores from the file system, using these data stores and the Gellish framework subsystem, the Controller will execute a local query. Also using the Service Client, the Controller will first retrieve a list of connected engines from a tracker, and subsequently will send a execute query request to each of these engines. The result of both of these actions is combined into a single answer, and returned to the manager application.

When an engine receives a request from another engine, via the Service Host, it will call the appropriate method on the Controller. In the case of a Gellish query, the controller will perform in a similar fashion as when the manager application started a query: query locally and forward query to other engines. The result of these tasks is also combined, but this time the result is replied back to the requested engine.

Note that all these actions have a blocking character on the sender, which means that the thread in which a Service Client is started cannot do any other work until a response is received back. This scenario ensures that a requesting engine will not have to open up its firewall if not desired, since the response will be received as a result of sending the request.

Finally, in a similar fashion as with the peer tracker a DOL manager application is able to retrieve and change the status of the Engine Windows service, installed on the local machine.

3.3.5 DOL Manager Subsystem

Using the DOL manager users will be able to monitor and administrate their locally installed engine and/or peer tracker services. Using the manager application data stores, publications, subscriptions can be created or removed; also the existence of other engines can be verified (through the use of one or more peer tracker service).

The manager application is not the focus of this research project, and thus no special decomposition will be given here. All logic associated with querying, publishing, and subscribing is build into the peer tracker and engine subsystems. The manager application is just a shell which offers a user interface to the users of these systems.
Chapter 4

Implementation

With the requirements gathered, as well as having an overview of the various subsystems, presented in the form of a subsystem decomposition, the next step is to develop the actual distributed object library framework and applications.

4.1 Programming Language

This section is concerned with the programming language used to build the actual prototype application. For this research project the Microsoft .NET Framework\(^1\) was selected. The motivation for choosing this framework was based on having previous experience developing applications using this framework as well as a number of available off-the-shelf components, needed to create distributed, service oriented applications.

4.1.1 Microsoft .NET Framework

Microsoft’s .NET Framework\(^2\) is a managed code model which allows programmers to build applications upon the Windows operating systems. However it is also possible to build systems for non-Microsoft systems, like Unix or Linux distributions, or Mac OS X [Tro07].

The .NET Framework is highly reliant on the XML standard\(^3\) and thus allows for easily data sharing and software integration. Also Microsoft .NET enhances integration by embracing other programming languages, which is done by having a dedicated intermediate language, called the Common Intermediate Language (CIL), see figure 4.1.

Several software vendors have already released .NET specific compilers for their programming language. Five languages are shipped by default with the Visual Studio development environment, which will be discussed in section 4.2, namely C#, J#, Visual Basic .NET, Managed Extensions for C++, and JScript .NET. Other languages for which dedicated .NET compilers can be found are for instance Pascal, COBOL, Eiffel, Fortran and Smalltalk.

C# Programming Language

All classes will be built using the C# variant of the .NET Framework. C# is a procedural object oriented programming language which is similar to C++ mixed with influences from

---

\(^1\)http://www.microsoft.com/net/

\(^2\)http://msdn.microsoft.com/en-us/netframework/

\(^3\)http://www.w3c.org/xml
Chapter 4. Implementation

other languages, like Java. Also the C# language has been approved by Ecma\(^4\) and ISO, described in ECMA-334 and the ISO/IEC 23270 standard respectively.

.NET Framework 2.0

A number of different versions or framework exist (see figure 4.2), however only version 1.1, which is not shown in the figure, was actually replaced by the newer version 2.0 in November 2005. The versions 3.0 and 3.5 are built on top of components from the 2.0 framework.

The .NET framework 2.0 has a base class library containing classes, which provide common functions and data structures. Examples of functions offered by these base classes are: collection operations, associated with lists and arrays; basic IO handling, i.e. reading from and writing to files; as well as various mathematical functions.

Figure 4.2 shows the various releases of the .NET framework, starting at version 2.0, followed by versions 3.0 and 3.5. Apart from the base class library, version 2.0 of the .NET framework, also consists of classes for building Windows applications (Winforms), building web applications (ASP.NET), and for interacting with database systems (ADO.NET).

.NET Framework 3.0

In November 2006 Microsoft released version 3.0 of the .NET Framework. Instead of changing the API structure of the previous version, version 3.0 adds a set of APIs which were released with Microsoft Windows Vista. The four components added to the .NET Framework are (see figure 4.2):

\(^4\)International standards organization for information and communication systems.
4.1. Programming Language

Figure 4.2: Overview of the different .NET frameworks illustrating the main components of each of these versions of the framework.

- **Windows Communication Foundation** Especially designed for the process of building applications using service oriented messaging.

- **Windows Presentation Foundation** Combines a number of different APIs concerned with building user interfaces, 2D and 3D graphics, streaming video, and more.

- **Windows Workflow Foundation** This component allows developers to define and execute workflows in their applications.

- **Windows CardSpace** This fourth component allows applications to store and retrieve a person’s digital identities.

Since components, found within the Windows Communication Foundation, will be used in the services developed during this research project, WCF will be discussed in a bit more detail below.

.NET Framework 3.5

Figure 4.2 also illustrates the additional features of version 3.5 of the .NET Framework. The most important of these two features for this project is the LINQ (Language Integrated Query) addition. The query capabilities offered by LINQ allow for the access and integration of information, which is not natively defined in object oriented technologies, like relation databases or XML.

Using LINQ in this project will eliminate the need for different query structures based on the form in which the Gellish information is stored. Thus no SQL queries will be written for Gellish information stored in SQL databases, and no need for using XPath in case of Gellish stored in XML documents.

Windows Communication Foundation (WCF)

Microsoft’s WCF offers an API for creating service-oriented applications, and is entirely built on the .NET Framework [Smi07]. It fully embraces the concepts of service orientation, and
Chapter 4. Implementation

Figure 4.3: Screenshot of the source code editor included within the Microsoft Visual Studio developer environment.

it can create messages that comply with many of the WS-* specifications, but it can also be used in distributed architectures that use Plain Old XML messages. Both the DOL engine and DOL peer tracker services will be built using various components found in WCF. Also the actual communication between engines and peer trackers will be performed by WCF objects.

4.2 Developer Tools

The most important programs, and tools, used during the construction of the framework, are briefly discussed in the following sections.

4.2.1 Microsoft Visual Studio

This integrated development environment is aimed at helping developers build class libraries, console applications, graphical user interface applications, web sites and more. Microsoft Visual Studio 2008 consists of a code editor, integrated .NET compiler, forms designer and debugger.

The code editor supports auto completing, known as IntelliSense, which helps speed up the writing of source code. This is done by displaying possible statements and other identifiers based on only one or two typed characters. Also methods can be accompanied by a description of the method combined with its parameters.

Graphical user interfaces can be created using the graphical forms designer built into the environment. This designer makes it possible to drag and drop basic form components, like textboxes and buttons, into the window or dialog being constructed. Components added using this designer can also easily be resized or modified by either dragging the outsides of the components or using the included properties window. All source code needed to construct these components during runtime is automatically created.

Another important tool is the debugger, allowing a per statement inspection of source code being executed. During debugging the environment can display values of variables within the scope of execution.

4.3. Persistent Data

Figure 4.4: Screenshot taken from the SQL Server Management Studio, while viewing a table containing Gellish facts.

4.2.2 Subversion

In order to keep track of various versions of both the source code and documents produced, Subversion\(^6\) is used as a version control tool. Since the company I worked at during the project had already a Subversion server running, I was able to create a number of repositories using the Tortoise SVN 1.5.5 user interface\(^7\).

4.2.3 SQL Server Management Studio

SQL Server Management Studio is used to manage the SQL servers and databases containing Gellish tables. Databases used will be created on an instance of SQL Server version 9.0 (also known as SQL Server 2005). As will be discussed in section 4.3, each Gellish data store will consist of a single table, containing facts. To retrieve the facts from the tables, stored procedures will be used, which are also created using the user interface.

Apart from viewing, creating and modifying the databases, tables, and stored procedures on the SQL server, the SQL Server Management Studio also provides various tools for importing and exporting from and to other formats. Because all Gellish tables, prior to this project, are stored in Excel workbooks, the import tools will be used to extract these facts from the workbooks and insert them into database tables.

4.3 Persistent Data

This section will describe the considerations and design choices made concerning persistent data used by the framework. The focus will be on the persistent storage of Gellish facts and tables, either in XML files or SQL tables. The last part of this section will briefly discuss the storage and retrieval of application data, like settings of services and paths to data stores.

---

\(^6\)http://subversion.tigris.org/

\(^7\)http://tortoisesvn.net/
4.3.1 Gellish Table Format

The main type of data used within the distributed object library are facts stored in Gellish tables. Therefore this first part will take off the introduction on Gellish given in chapter 2. The definitions found in this section are taken from Definition of Gellish Databases [VR08].

Predefined Subsets

At the moment of writing the Gellish definition identifies about forty columns. Van Renssen presents a number of predefined table subsets, each intended for a specific type of application or use. The default subset is the Business Model subset, designed to describe propositions about imaginary objects (found in designs), as well as real world objects, during all phases of their lifecycle.

These sections will discuss all Gellish columns found in the Extended Table subset.

Value Spaces

Before giving an overview of the Gellish columns found in this Extended Table subset, the main value spaces used by the various columns will be described, as well as the primary constraints on these value spaces. Note that a more specific discussion concerning the database- and XML implementations is given in their respective sections.

**Integer Values** will be used for representing unique (numeric) identifiers. The primary aspect of this value space is the number of bits (or bytes) used to encode the numerical value represented. The size of the representation will be 64 bits (or 8 bytes), and will be used for the UIDs.

Although the usage of negative integers is not explicitly discussed in [VR08], all numerical value are allowed to be negative, and thus values are "signed". In contrast, "unsigned" types only allow values of zero and greater. Note however that a UID having a negative value will not be valid.

As a result the valid range of values for a 64 bits integer is -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807. For each numerical column, discussed in section 4.3.1, the appropriate values space will be given.

**Textual Values** representing strings of characters will be constraint by two properties: number of allowed characters within the string; and the number of bits used to encode each character. The latter is closely related to the number of different characters, which can be encoded. For this reason Gellish columns will be attributed to either support Unicode or not (non-Unicode).

Unicode is being developed by the Unicode Consortium\(^9\) to support text systems for almost all languages. As a consequence Unicode requires a lot more bits to encode characters, however some columns might require such a large address space. The distinction for supporting Unicode will be given per (text) column in section 4.3.1.

Although the length of a valid string of characters can be unbounded and finite, a number of database systems have a limit on the number of bytes they can effectively store in a single row. For instance Microsoft SQL Server has a limit of 8060 bytes per row\(^10\). Note however that it is possible to store large chunks of text using the text or ntext datatypes in SQL

---

\(^8\)Except columns marked as superfluous by Van Renssen in [VR08]
\(^9\)http://www.unicode.org
\(^10\)Starting with version 2005 SQL server allows rows to exceed the 8060 bytes limit.
server, but this incurs a performance penalty, since not the complete text, but only a pointer to the text is stored in the table. For this reason the Gellish column definitions, discussed next, will give an indication of the valid length of a text.

**Date and Time Values** to keep track of the creation- and modification date of a Gellish fact, a date time values is required to store these moments. Data stores should be able to keep track of the date (year, month and day) as well as the time including the fraction (hours, minutes, seconds, and even fractions of seconds). The requirements for date and time values is defined in the ISO 8601 standard\(^\text{11}\).

**Column Definitions**

This section will list all columns found in the Gellish table definitions, excluding columns which are marked superfluous by Van Renssen. Table A.1 is taken from [VR08] and lists each column, a description, the appropriate type of value space, and whether the column is obligatory or optional. In case of integer values the number of bits will be given between parentheses, within the parenthesis of the textual type, the maximum number of allowed characters is given as well as an indication if the Unicode encoding is allowed.

Each column also has an indication of the appropriate datatype for use within an SQL Server or XML file, which will be discussed in sections 4.3.2 and 4.3.3.

Note that Van Renssen uses more abbreviated terms in [VR08], for instance left hand columns are prefixed by "LH" (and "RH" for the right hand side), also Relation is abbreviated by "Rel", Validity by "Val". However I will not abbreviate the parts of the column names, and for clarity write them in full.

### 4.3.2 Database Systems

The first type of data store which will be used to save Gellish facts is a (SQL) database. For this prototype version I will be choosing for the Microsoft SQL Server relational database management system (RDBMS). The reason for considering only this DBMS is because the .NET Framework already contains classes for retrieving data stored in such systems, using the ADO.NET features in .NET version 2.0. However libraries have been build, which enable data providers to connect to MySQL, ProgresSQL, Sybase or Oracle databases. Apart from the Microsoft SQL Server system, XML files will also be used for storing facts (see next section).

**Microsoft SQL Server**

The Microsoft SQL Server version used during development and initial testing is the 2005 edition, also known as version 9.0. Since only the basic database features are used, the free Express edition of the database system suffices. A data store, or collection of (related) Gellish facts will be stored in a single table. The facts will be read from the database using a stored procedure, which accesses the table containing the facts.

**SQL Datatypes**

In order to store values from a particular value space, each column within a database table needs to have the appropriate datatype. This section will describe the mapping from the value spaces defined in section 4.3.1 onto the SQL server datatypes.

\(^{11}\text{http://www.iso.org/iso/date_and_time_format.}\)
Chapter 4. Implementation

SQL server has a number of types for storing integer values, in SQL server the datatype `bigint` is the 64 bits variant of an integer. As a consequence the datatype `bigint` will be used for the unique identifier columns found within a Gellish fact.

Within SQL Server a number of different datatypes are used to represent character strings. Each of these types differs in the allowed range of characters (Unicode versus non-Unicode). When a column supports Unicode the datatype is prefixed with an `n`, e.g. the datatype of a non-Unicode string of variable length is `varchar`, while the same datatype is `nvarchar` for a Unicode string of variable length.

Another important difference is whether the actual string is stored within the table, or if a pointer exists, which point to a blob of data, or chunk of text outside the table. This pointer is used when the size of one or more text columns needs to exceed the 8060 bytes limit for an SQL table. These datatypes are `text` and `ntext` and allow for a maximum of respectively 2,147,483,647 and 1,073,741,823 characters. Apart from the performance loss, when using these types, these types also have additional drawbacks, the most important being that they cannot be compared, sorted, or used in a `GROUP BY` clause (which are basic SQL functions). Therefore the only columns represented by these types are the Gellish columns which require large chunks of text, i.e. "Full definition" and "Remarks".

The selection for the date and time datatype in SQL is easier, since SQL only has one such a datatype, namely the `datetime` datatype.

Table 4.1 summarizes the column names, allowed data types, and wheter or not values may remain `NULL`. These columns will be implemented in the SQL tables containing Gellish facts in an SQL database. Appendix B gives the script used to create a Gellish table within SQL.

4.3.3 XML Schema

Every engine should be able to function, without the use of a dedicated database system, therefore it will also be possible to store Gellish facts within plain text file, or more precisely XML files. The Extensible Markup Language (XML) is an open standard developed by World Wide Web Consortium (W3C) and is used to format data in a structured fashion using plain text files.

In order to exchange valid the structure and content of a Gellish XML file, an XML Schema will be created. Note that valid content here only refers to the fact that the values found for the different Gellish columns is valid within the value space of that column. The XML Schema will not be able to give any indication on the semantical correctness of the content.

Before presenting the XML Schema used for describing the legal structure of an XML file containing Gellish facts, this section will first discuss a number of XML Schema datatypes which will be used in the Gellish XML Schema. Datatypes were already discussed in previous sections, however this section will define the valid value of each of the Gellish columns represented in a Gellish XML document.

XML Schema Datatypes

The first set of datatypes are basic XML Schema datatypes, developed by the W3C. A complete description of the XML datatypes see the W3C website\textsuperscript{12}.

\footnote{\url{http://www.w3.org/TR/xmlschema-2/}}
### 4.3. Persistent Data

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Allow Nulls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>int</td>
<td>yes</td>
</tr>
<tr>
<td>Uid</td>
<td>bigint</td>
<td>no</td>
</tr>
<tr>
<td>LineUid</td>
<td>bigint</td>
<td>yes</td>
</tr>
<tr>
<td>SuccessorUid</td>
<td>bigint</td>
<td>yes</td>
</tr>
<tr>
<td>ValidityContextUid</td>
<td>bigint</td>
<td>yes</td>
</tr>
<tr>
<td>ValidityContextName</td>
<td>nvarchar(255)</td>
<td>yes</td>
</tr>
<tr>
<td>LanguageUid</td>
<td>bigint</td>
<td>yes</td>
</tr>
<tr>
<td>LanguageName</td>
<td>nvarchar(255)</td>
<td>yes</td>
</tr>
<tr>
<td>LeftNameContextUid</td>
<td>bigint</td>
<td>yes</td>
</tr>
<tr>
<td>LeftNameContextName</td>
<td>nvarchar(255)</td>
<td>yes</td>
</tr>
<tr>
<td>LeftObjectUid</td>
<td>bigint</td>
<td>yes</td>
</tr>
<tr>
<td>LeftObjectName</td>
<td>nvarchar(255)</td>
<td>no</td>
</tr>
<tr>
<td>LeftObjectTypeName</td>
<td>nvarchar(255)</td>
<td>yes</td>
</tr>
<tr>
<td>LeftObjectRoleUid</td>
<td>bigint</td>
<td>yes</td>
</tr>
<tr>
<td>LeftObjectRoleName</td>
<td>nvarchar(255)</td>
<td>yes</td>
</tr>
<tr>
<td>LeftCardinalities</td>
<td>varchar(32)</td>
<td>yes</td>
</tr>
<tr>
<td>Reality</td>
<td>nvarchar(255)</td>
<td>yes</td>
</tr>
<tr>
<td>Intention</td>
<td>varchar(255)</td>
<td>no</td>
</tr>
<tr>
<td>RelationTypeUid</td>
<td>bigint</td>
<td>no</td>
</tr>
<tr>
<td>RelationTypeName</td>
<td>nvarchar(255)</td>
<td>no</td>
</tr>
<tr>
<td>RightObjectUid</td>
<td>bigint</td>
<td>yes</td>
</tr>
<tr>
<td>RightObjectName</td>
<td>nvarchar(255)</td>
<td>no</td>
</tr>
<tr>
<td>RightObjectRoleUid</td>
<td>bigint</td>
<td>yes</td>
</tr>
<tr>
<td>RightObjectRoleName</td>
<td>nvarchar(255)</td>
<td>yes</td>
</tr>
<tr>
<td>RightCardinalities</td>
<td>nvarchar(32)</td>
<td>yes</td>
</tr>
<tr>
<td>Description</td>
<td>ntext</td>
<td>yes</td>
</tr>
<tr>
<td>PartialDefinition</td>
<td>ntext</td>
<td>yes</td>
</tr>
<tr>
<td>FullDefinition</td>
<td>ntext</td>
<td>yes</td>
</tr>
<tr>
<td>Remarks</td>
<td>ntext</td>
<td>yes</td>
</tr>
<tr>
<td>UnitOfMeasureUid</td>
<td>bigint</td>
<td>yes</td>
</tr>
<tr>
<td>UnitOfMeasureName</td>
<td>nvarchar(32)</td>
<td>yes</td>
</tr>
<tr>
<td>MappingAccuracyUid</td>
<td>bigint</td>
<td>yes</td>
</tr>
<tr>
<td>MappingAccuracyName</td>
<td>nvarchar(255)</td>
<td>yes</td>
</tr>
<tr>
<td>CollectionUid</td>
<td>bigint</td>
<td>yes</td>
</tr>
<tr>
<td>CollectionName</td>
<td>nvarchar(255)</td>
<td>yes</td>
</tr>
<tr>
<td>PicklistUid</td>
<td>bigint</td>
<td>yes</td>
</tr>
<tr>
<td>PicklistName</td>
<td>nvarchar(255)</td>
<td>yes</td>
</tr>
<tr>
<td>ApprovalStatus</td>
<td>nvarchar(50)</td>
<td>no</td>
</tr>
<tr>
<td>EffectiveFrom</td>
<td>datetime</td>
<td>no</td>
</tr>
<tr>
<td>LatestUpdate</td>
<td>datetime</td>
<td>no</td>
</tr>
<tr>
<td>Author</td>
<td>nvarchar(255)</td>
<td>yes</td>
</tr>
<tr>
<td>Reference</td>
<td>nvarchar(255)</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 4.1: Column names used in the SQL implementation of a Gellish table. Each name is accompanied by the used data type and whether the values stored in the respective column may remain null.
Chapter 4. Implementation

string  The string datatype represents character strings in XML having a set of finite length sequences of characters. A character is an atomic unit of communication and can be either a tab, carriage return or line feed or any character from Unicode.

token  The datatype token represents tokenized strings. The value space of token is the set of strings that do not contain the carriage return, line feed nor tab characters, that have no leading or trailing spaces and that have no internal sequences of two or more spaces [W3C04]. The base type of token is normalizedString (which itself is a derived type from string).

NMTOKEN  The value space of NMTOKEN is the set of tokens that match the Nmtoken production rule found in XML 1.0 (Second Edition) [XML]. The base type of NMTOKEN is token. Valid NMTOKEN values contain characters, digits, period, colons, hyphens, and the characters defined by Unicode.

long  The long datatype is derived from integer by setting the value of maxInclusive to be 9,223,372,036,854,775,807 and minInclusive to be -9,223,372,036,854,775,808. This value space is equal to the value space of the 64 bit representation of an integer discussed at the beginning of this chapter.

dateTime  The value space of dateTime corresponds to the dates and times described in the ISO 8601 standard, and thus allow moments to be represented by the year, month, day, hour, minute, second, and fractions of a second.

Gellish Schema Datatypes

Now that the basic types within the XML Schema definitions are given, this section will define a number of Gellish specific derived types. These type are derived from the basic types and extended with additional constraints.

uid  The uid is derived from long by setting the value of minInclusive to be 0 (maxInclusive still remains at 9,223,372,036,854,775,807), thus guaranteeing that all uids used are non negative. The base type is long. The actual definition in the Gellish uid datatype looks as follows:

```xml
<xs:simpleType name="uid">
  <xs:restriction base="xs:long">
    <xs:minInclusive value="0" />
  </xs:restriction>
</xs:simpleType>
```

cardinalities  The cardinalities datatype is derived from NMTOKEN and supports the simultaneous left- and right hand cardinalities. However cardinalities only accepts values with the pattern ‘((0|\[1-9]\[0-9\]*)\,\,(0|\[1-9]n))’. This pattern will only allow: numeric value, followed by a comma, followed by a numeric value or “n”.

```xml
<xs:simpleType name="cardinalities">
  <xs:restriction base="xs:token">
    <xs:pattern value="\([0\[1-9]\[0-9\]*\,\,(0\[1-9]|0\[0-9\]*)n]\)" />
  </xs:restriction>
</xs:simpleType>
```

13http://www.w3.org/TR/2000/WD-xml-2e-20000814
4.3. Persistent Data

**status** The status is derived from token and supports the approval status of a Gellish fact, the allowed values are: proposed, issue, deleted, proposed to be deleted, ignore, agreed, accepted, accepted association, replaced.

```xml
<xs:simpleType name="status">
  <xs:restriction base="xs:token">
    <xs:enumeration value="proposed"/>
    <xs:enumeration value="issue"/>
    <xs:enumeration value="deleted"/>
    <xs:enumeration value="proposed to be deleted"/>
    <xs:enumeration value="ignore"/>
    <xs:enumeration value="agreed"/>
    <xs:enumeration value="accepted"/>
    <xs:enumeration value="accepted association"/>
    <xs:enumeration value="replaced"/>
  </xs:restriction>
</xs:simpleType>
```

**reality** The reality has the value space required to support the reality classification of a left object: imaginary, materialized, real. The base type of reality is NMTOKEN.

```xml
<xs:simpleType name="reality">
  <xs:restriction base="xs:NMTOKEN">
    <xs:enumeration value="imaginary"/>
    <xs:enumeration value="materialized"/>
    <xs:enumeration value="real"/>
  </xs:restriction>
</xs:simpleType>
```

**intention** Intention can be represented in a XML document using the intention datatype. This type is an enumeration over the various values given in the Gellish Definitions. The base type of reality is NMTOKEN.

```xml
<xs:simpleType name="intention">
  <xs:restriction base="xs:NMTOKEN">
    <xs:enumeration value="true"/>
    <xs:enumeration value="false"/>
    <xs:enumeration value="request"/>
    <xs:enumeration value="question"/>
    <xs:enumeration value="confirmation"/>
    <xs:enumeration value="promise"/>
    <xs:enumeration value="declination"/>
    <xs:enumeration value="statement"/>
    <xs:enumeration value="probability"/>
    <xs:enumeration value="acceptance"/>
  </xs:restriction>
</xs:simpleType>
```

**stringCommonality** The left hand and right hand string commonalities will be represented using the stringCommonality type. The various options given in the Gellish Definitions are enumerated in this type.

```xml
<xs:simpleType name="stringCommonality">
  <xs:restriction base="xs:token">
    <xs:enumeration value="case sensitive identical"/>
    <xs:enumeration value="case insensitive identical"/>
    <xs:enumeration value="case sensitive partially identical"/>
    <xs:enumeration value="case insensitive partially identical"/>
    <xs:enumeration value="case sensitive front end identical"/>
    <xs:enumeration value="case insensitive front end identical"/>
  </xs:restriction>
</xs:simpleType>
```
Chapter 4. Implementation

Complex Complex Types

Each of the complex types described below is constructed using both standard XML Schema datatypes and Gellish simple datatypes from the section above. The types are created in such a way that they may represent identifying components found in each Gellish fact.

**objectName**  The objectName is used for representing the left- and right hand object names within the leftObject- and rightObject types. Also an additional StringCommonality attribute is added to support the left- and right hand string commonalities (columns 80 and 81).

```xml
<xs:complexType name="objectName">
  <xs:simpleContent>
    <xs:extension base="xs:token">
      <xs:attribute name="StringCommonality" type="gellish:stringCommonality" use="optional" />
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

**leftObject**  The leftObject type is used to represent both the left object. Each leftObject must contain the Uid attribute, which is equal to the unique left hand object identifier (column 2). One of the children of the leftObject type is the Name attribute (see objectName type), which contains the left hand object name (column 101). Other possible children are the NameContext, used for the left hand name context (columns 71 and 16); a Cardinalities element (column 44); a Role element (columns 72 and 73); and a TypeName, which is equal to the left hand object type name (column 38).

```xml
<xs:complexType name="leftObject">
  <xs:all>
    <xs:element name="Name" type="gellish:objectName" minOccurs="0" maxOccurs="1" />
    <xs:element name="NameContext" type="gellish:context" minOccurs="0" maxOccurs="1" />
    <xs:element name="Role" type="gellish:role" minOccurs="0" maxOccurs="1" />
    <xs:element name="Cardinalities" type="gellish:cardinalities" minOccurs="0" maxOccurs="1" />
  </xs:all>
  <xs:attribute name="Uid" type="gellish:uid" use="required" />
  <xs:attribute name="Reality" type="gellish:reality" use="optional" />
</xs:complexType>
```

**rightObject**  The rightObject type is similar to the left object but lacks a number of elements and attributes. Each rightObject must contain the Uid attribute, which is equal to the unique right hand object identifier (column 15). One of the children of the rightObject
type is the Name attribute (see objectName type), which contains the right hand object name (column 201). Other possible children are the Role element (columns 74 and 75); and a Cardinalities element (column 45).

```xml
<xs:complexType name="rightObject">
  <xs:all>
    <xs:element name="Name" type="gellish:objectName" minOccurs="0" maxOccurs="1" />
    <xs:element name="Role" type="gellish:role" minOccurs="0" maxOccurs="1" />
    <xs:element name="Cardinalities" type="gellish:cardinalities" minOccurs="0" maxOccurs="1" />
  </xs:all>
  <xs:attribute name="Uid" type="gellish:uid" use="required" />
</xs:complexType>
```

**relationType**  The complex type relationType is used to represent the relation type unique identifier and -name columns (60 and 3). Note that the Uid attribute for the relationType type is required.

```xml
<xs:complexType name="relationType">
  <xs:simpleContent>
    <xs:extension base="xs:token">
      <xs:attribute name="Uid" type="gellish:uid" use="required" />
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

**language**  The language complex type is used to represent the unique language identifier and -name columns (69 and 54).

```xml
<xs:complexType name="language">
  <xs:simpleContent>
    <xs:extension base="xs:token">
      <xs:attribute name="Uid" type="gellish:uid" use="optional" />
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

**context**  A context type is used to represent the either the validity context for the main fact (19 and 18) or the context for the left hand object name (71 and 16).

```xml
<xs:complexType name="context">
  <xs:simpleContent>
    <xs:extension base="xs:token">
      <xs:attribute name="Uid" type="gellish:uid" use="optional" />
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

**role**  The role is used to represent the role of the left hand object (columns 72 and 73) or the right hand role (74 and 75).

```xml
<xs:complexType name="role">
  <xs:simpleContent>
    <xs:extension base="xs:token">
      <xs:attribute name="Uid" type="gellish:uid" use="optional" />
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```
Chapter 4. Implementation

**uom**  Complex type `uom` is used to represent the Unit of Measure identifier and -name columns (66 and 7).

```xml
<xsd:complexType name="uom">
  <xsd:extension base="xsd:token">
    <xsd:attribute name="Uid" type="gellish:uid" use="optional" />
  </xsd:extension>
</xsd:complexType>
```

collection  The collection is used to represent the collection unique identifier and -name columns (50 and 68).

```xml
<xsd:complexType name="collection">
  <xsd:extension base="xsd:token">
    <xsd:attribute name="Uid" type="gellish:uid" use="optional" />
  </xsd:extension>
</xsd:complexType>
```

picklist  The picklist type is used to represent the picklist unique identifier and -name columns (70 and 20).

```xml
<xsd:complexType name="picklist">
  <xsd:extension base="xsd:token">
    <xsd:attribute name="Uid" type="gellish:uid" use="optional" />
  </xsd:extension>
</xsd:complexType>
```

**mappingAccuracy**  The mappingAccuracy is used to represent the accuracy of mapping unique identifier and -name columns (76 and 77).

```xml
<xsd:complexType name="mappingAccuracy">
  <xsd:extension base="xsd:token">
    <xsd:attribute name="Uid" type="gellish:uid" use="optional" />
  </xsd:extension>
</xsd:complexType>
```

**Gellish Table Element**

This last part of the Gellish XML Schema presents the structure of the Gellish Table element. As can be seen below each Table element can contain zero to an unbounded number of Fact elements. Each Fact element contains the all Gellish columns given in the Table 1.

```xml
<xsd:element name="Table">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="Fact" minOccurs="0" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:all>
            <xsd:element name="LeftObject" type="gellish:leftObject" minOccurs="1" maxOccurs="1" />
          </xsd:all>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```
4.3. Persistent Data

4.3.4 XML Document

A XML document contains the Gellish facts found in a Gellish table. These XML documents should adhere to the XML schema given above. Appendix D contains a sample XML document containing four different facts, ranging from a minimal fact to a completely filled fact.

The extension for these files normally is .xml, but when used for specific purposes the first letter of this extension is often replaced with a more appropriate letter, e.g. .html for HyperText Markup Language. For this purpose XML files, containing Gellish facts will also be identified by the .gml (Gellish Markup Language) extension. Because this type of file should not be confused with other files having this extension, like the Geography Markup Language [Cox04] or the Graph Modeling Language [Him95], the XML documents containing Gellish tables should contain a reference to the Gellish namespace.

4.3.5 GMZ Files

During the development of the aforementioned Gellish XML files (having the .gml extension), the zipped version was also created. The goal of this file format was to reduce the size of
files containing Gellish tables, using basic compression tools according to the ZIP format. This type of file was inspired by Google’s .kmz files, which are zipped files containing XML files with the .kml extension (Keyhole Markup Language), together with other types of files, like images.

4.3.6 SOAP Messages

Since XML serialization is used to serialize a Gellish table before it is sent within a SOAP message, the format of a Gellish table in these messages is similar to the XML structure presented in the previous sections. The Gellish table objects all override the basic methods exposed through the IXmlSerializable interface in .NET. More details about XML en SOAP serialization can be found on the MSDN website 15.

4.3.7 Application Data

This last section will briefly discuss the storage of application data. Example of this data is the list of connected engine, maintained by a peer tracker application: or the various data stores attached to a DOL engine. Using the binary serialization of .NET objects, these objects are saved to .dat files on the hard disk of the machine on which these applications are installed.

The standard serializers shipped with .NET version 2.0 will be used for this process, these serializers are capable of creating exact clones of the aforementioned objects used in memory. For a more complete discussion on binary .NET serializers, please visit the MSDN website 16.

4.4 Constructing the Framework

The distributed object library framework and associated applications consist of a number of separate libraries and projects, each of which will be discussed in greater detail in the following sections. However before delving into the various components making up the framework, a number of conventions used during the development are discussed, as well as some design trade-offs.

4.4.1 Naming Conventions

Since a number of libraries will possibly be used by other developers, it is worth pointing out what naming conventions or rules are being used. The guidelines followed during this project were taken from the Microsoft Developer Network website [Mic08], which were also used in the development of the .NET Framework itself. The primary goal of following these guidelines, is to maximize the familiarity developers might have with the .NET namespaces and classes, and thus increases the predictability and discoverability of class libraries.

Capitalization Rules

All identifiers found in the libraries, will follow one of the next three capitalization rules. Table 4.2 lists all types of identifiers found in the libraries together with the capitalization rule used.

4.4. Constructing the Framework

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Rule</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Namespace</td>
<td>pascal case</td>
<td>Gellikx.Gellish</td>
</tr>
<tr>
<td>Interface</td>
<td>pascal case</td>
<td>IGellishEntity</td>
</tr>
<tr>
<td>Class</td>
<td>pascal case</td>
<td>GellishFact</td>
</tr>
<tr>
<td>Method</td>
<td>pascal case</td>
<td>ToString</td>
</tr>
<tr>
<td>Parameter</td>
<td>camel case</td>
<td>tableName</td>
</tr>
<tr>
<td>Property</td>
<td>pascal case</td>
<td>LeftObjectUid</td>
</tr>
<tr>
<td>Local variables</td>
<td>camel case</td>
<td>connectionString</td>
</tr>
<tr>
<td>Enum type</td>
<td>pascal case</td>
<td>GellishField</td>
</tr>
<tr>
<td>Enum values</td>
<td>pascal case</td>
<td>Proposed</td>
</tr>
<tr>
<td>Read-only static field</td>
<td>pascal case</td>
<td>BusinessModel</td>
</tr>
<tr>
<td>Event</td>
<td>pascal case</td>
<td>ValueChange</td>
</tr>
<tr>
<td>Exception class</td>
<td>pascal case</td>
<td>InvalidFactException</td>
</tr>
</tbody>
</table>

Table 4.2: Capitalization rules used to name the different identifiers. Note: Interface will always be prefixed by "I" and exception classes will always be suffixed by "Exception".

**Pascal Case**  Identifiers named according to this rule start with an upper case letter. The first letter of each subsequent concatenated part of the identifier is also upper case.

**Camel Case**  Identifiers named according to the camel case rule start with a lower case letter. However, the first letter of each subsequent concatenated part of the identifier is upper case.

**Upper Case**  Every letter of the identifier is upper case. However this rule is only applied to identifier consisting of only one or two letters. This rule is included for completeness, since no identifiers with less than three characters are used in this project.

As can be seen in the table the exception classes will contain the suffix "Exception". This inclusion of the type of object in the identifier name will be followed where possible, for instance GellishTable, GellishFactCollection, and EngineController.

**Namespaces**

Every C# class libraries, services and applications developed during this project, will be part of the Gellikx namespace, which can be seen as a container for organizing the classes used within these projects. The name Gellikx is derived from the name of the company at which this project was performed. Although the name of the company changed during the project from Gellikx to Idoro, the name of this namespace was not changed.

Namespaces found in the Microsoft .NET Framework all start within the System namespace. In order to be inline with the namespaces used by .NET, namespace names used in the libraries of this project will follow a similar naming convention. Thus classes resembling classes in the .NET Framework will be organized, if possible, in similar namespaces, i.e. the SqlGellishAdapter class, designed to fill a GellishTable object with facts from an SQL database can be found in the Gellikx.Gellish.Data.SqlClient namespaces, similarly to the SqlDataAdapter found in the .NET Framework within the namespace System.Data.SqlClient.
Chapter 4. Implementation

Figure 4.5: Class diagram of the Gellikx class library. The classes within these namespaces are the most generic classes within this project.

Type Names

Apart from the organization of namespaces and their respective names, names of the various classes, interfaces, enums, etc., will also be the similar to the counterparts of these building block, found in the Microsoft .NET Framework, i.e. GellishSet and GellishTable, which are similar to the DataSet and DataTable objects found in .NET. As a result all classes will start with an upper case letter, each next concatenated word is also capitalized.

4.4.2 Class Libraries

Each of the following libraries, contains base classes which will be used by the service libraries and applications. Not every class developed within these libraries will be discussed in these sections, only the most important will be highlighted here.

Gellikx

This is the most generic class library. Classes within the namespaces of this library (see figure 4.5) can be used by any of the other class library, service libraries and/or applications. For this project however only four classes are important, these classes are AsyncResult, ExecutionTimer, GellikxExtensions and XsdValidator.

The class AsyncResult is used as a handle for asynchronously started methods. Example of such methods are the execution of a query on a remote DOL engine or the retrieval of all available publications. A inherited class from AsyncResult contains the input and output parameters of the method, which should be called asynchronously, and offers a state and exception property to determine if a execution has been completed successfully.

By executing these methods asynchronously, the calling application is not blocked, and can start execution of similar methods. This base class implements System.IAsyncResult and will be extended by classes in the Gellikx.Dol.Engine namespace, depending on the type of result required.

ExecutionTimer is used during development to keep track of the execution time of various methods. Once development is done these classes will be removed.

GellikxExtensions contains extension methods. These types of methods were introduced in the .NET framework 3.0 and are a language feature of C#. Extension methods allow for static methods to be created for classes, even if the programmer does not have the sources of the original classes. Examples of methods created in this class are a method to remove the first or last letters of a string. More specific extension methods will be introduced in the Gellikx.Gellish.Extensions namespace.

Finally the class XsdValidator, found in the namespace Gellikx.Xml, allows XML files to be validated against a XSD schema file. During this validation any warnings and error
4.4. Constructing the Framework

Figure 4.6: Class diagram of the Gellikx.Gellish class library. Classes within these namespaces are used to retrieve, represent, and query Gellish tables.

are returned. This class will be primarily used to validate Gellish XML files (having the .gml extension) against the Gellish schema presented in section 4.3.3.

Gellikx.Gellish

The Gellikx.Gellish namespace contains all classes needed to represent, retrieve, and query Gellish tables, stored in either XML files or SQL databases. During development the ExecutionTimer was used to time various methods within this library, but because these were removed after the initial tests, the library is no longer dependent on the Gellikx library. The bulk of classes found in this library are shown in figure 4.6.

The most important classes within this class library are the GellishSet, GellishTable, and GellishFact classes. These classes are based on the classes found in the System.Data. Each GellishSet contains a collection of GellishTable objects, in turn each GellishTable contains a collection of GellishFact objects. Since these three classes implement .NET’s IXmlSerializable they expose a ReadXml and WriteXml method, which allow for the reading of and writing to a Gellish XML file, while adhering to the Gellish XML Schema.

The GellishFact class exposes the columns found in a Gellish fact in the form of properties. Some of these columns are combined into a special entity object, these entities will consist of a unique identifier and name. For instance the GellishRelationType entity is
made up out of the unique identifier of the relation type (RelationTypeUid), and the name of the relation type (RelationTypeName). The GellishFact class exposes both the entity object and separate id and name properties of this entity. Apart from the relation type, other entities are: context (both for fact validity and the left hand name); object (both left hand and right hand); role (left and right); language; collection; mapping accuracy; unit of measure; and picklist.

In order to read from SQL databases the class SqlGellishAdapter is developed. Using this class, which in turn inherits from the abstract class DbGellishAdapter, a GellishSet or GellishTable object can be filled with facts retrieved from the database and stored procedure supplied to the constructor of the SqlGellishAdapter. This database and stored procedure are identified by a connection string, pointing to the correct server and database, and a command(text), which indicates the stored procedure to be called.

To determine the mapping between the database columns and the desired properties of a GellishFact object (either a basic data type or a GellishEntity), two attribute classes are used: GellishEntityMappingAttribute; and GellishColumnMappingAttribute. These classes can be applied to properties as metadata, and can be read during execution using reflection. Both these attribute classes contain a property indicating the correct database column to be used.

Queries on Gellish tables will be made possible through the GellishQueryProvider class. This class has four properties, three of which can be set using the constructor of this class. These three properties are concerned with the actual knowledge, which should be queried, a GellishQuery object representing the query, and possibly an upper ontology\(^\text{17}\). The fourth property is the result, which is available after executing the provided query on the knowledge, based on a possible upper ontology. This execution is triggered by the method Execute, also found on the GellishQueryProvider class.

The actual procedure during the querying of a Gellish tables was already discussed in section 2.4.2. However before the actual execution of a query this query is first grouped into buckets of facts. Such a bucket represents a isolated set of facts, which together needs to be answered for the complete bucket to be successfully answered. After these buckets are created the query is first enriched, during this process the relation types and objects, found in facts contained within each of the buckets, are extended by other relation types and objects. These extensions are either relation types and objects, matching the search term provided, in combination with a string commonality, within the query. Or these relation types and objects are specializations of the relation types and objects provided in the query. Once these processes have finished, each bucket of (query) facts is executed against the indicated Gellish table(s).

Gellikx.Dol.Windows

Within the Gellikx.Dol.Windows library a number of basic components are found, which can be used to visualize and manage DOL engines, and all attached data, like remote engines, data stores and publications. Because the components of this library need to access classes within the service library of the DOL engine, the Gellikx.Dol.Windows library is dependent on Gellikx.Dol.Engine library. However no Gellish data is read or displayed in dialogs and forms exposed in this library, therefore this class library will not be dependent on the Gellikx.Gellish library. However this library is dependent on the Gellikx, Gellikx.Dol.Engine and Gellikx.Dol.PeerTracker libraries.

\(^{17}\)Within this project this upper ontology property is not used, section 5.3 discusses the absense of an upper ontology.
4.4. Constructing the Framework

Figure 4.7: Class diagram of the Gellikx.Dol.Windows.

Figure 4.7 shows the classes exposed in this library. Apart from two specialized MessageBox classes, the components within this library are dialogs and forms which will be used in the DOL manager application, as well as the DemoDolSetup application.

4.4.3 Service Libraries

Compared to the previously discussed class libraries, the two service libraries presented in this section can be compiled into a program. This program can be executed, which results in the specific service being started as a Windows service. Once these Windows services are started, they create a new ServiceHost which will listen for incoming requests. Both service libraries will be discussed on more detail in the next sections.

Gellikx.Dol.Engine

All classes needed to create a DOL engine service, manage it, and represent its data, are collected within this service (class) library. This library also contains the program class, which can be used to start the engine service as a service under Windows. In order to install this service, the Gellikx.Dol.Engine library is accompanied by a Windows setup.

Since the service library is based on classes found in the class libraries Gellikx and Gellikx.Gellish, this service library will be dependent on these libraries.

In the upper left corner of figure 4.8 a number of specialized asynchronous result classes are displayed. All of these classes inherit from the Gellikx.AsyncResult class, discussed in earlier in this section. The classes AsyncEngineDetailsResult and AsyncTrackerNameResult are used to asynchronously retrieve information about a peer, which can be either a DOL engine or DOL peer tracker. This information involves the name of a peer tracker, or the details of an engine, like its guid, name, a description etc.

AsyncEnginesResult is used when retrieving a list of known engines from a known peer tracker. Similarly the classes AsyncPublicationsResults and AsyncRequestSubscriptionResult are used when calling the methods for retrieving publications, or request a subscription from another engine.

Both the classes AsyncLocalQueryResult and AsyncRemoteQueryResult are used while performing Gellish queries asynchronously, however the former is used for queries which should be executed on local publications, while the latter is used in the case of remote engines.

Within the namespace Gellikx.Dol.Engine.Contracts the interfaces are defined for both the DOL engine and DOL peer tracker services, summarized in table 4.3. Reason for also including the service contract in the service library of the engine, is the dependency of
Figure 4.8: Class diagram of the Gellish class library (Gellikx.Dol.Engine). The classes within these namespaces are the most generic classes within this project.
4.4. Constructing the Framework

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetEngineDetails</td>
<td>Returns an instance of the EngineDetails class containing various info about the engine, like name, email, and a further description of the engine.</td>
</tr>
<tr>
<td>GetPublications</td>
<td>When called the engine returns an array containing instances of Publication, for all publications on the engine.</td>
</tr>
<tr>
<td>RequestSubscription</td>
<td>Based on the provided publication Guid, a subscription is requested for the engine, which is identified by a provided engine Guid.</td>
</tr>
<tr>
<td>CheckSubscription</td>
<td>Returns a SubscriptionStatus which represents the status of the subscription for a publication and the calling engine, both identified by a Guid.</td>
</tr>
<tr>
<td>CancelSubscription</td>
<td>Cancels a subscription of the calling engine, for a particular publication, based on the given Guids.</td>
</tr>
<tr>
<td>ExecuteQuery</td>
<td>Using the DolRequest, which contains a Gellish query, a Gellish response is created and returned to the calling engine.</td>
</tr>
</tbody>
</table>

Table 4.3: Overview of the operations defined in the IEngine contract, and implemented in the EngineService class.

The operations defined in the IEngine service contract, will be implemented by the EnginesService class, found in the Gellixk.Dol.Engine.ServiceModel namespaces. Using an instance of this service class, a WCF ServiceHost class will be constructed during execution, which will expose these operations through the WCF framework. Table 4.3 summarizes the operations of the engine service.

Apart from these service contracts, the library also contains a number of data contracts. Enums and classes found in the Gellixk.Dol.Engine.DataContracts namespace will be used in the communication between a peer trackers and/or engines. An example of such a class is the EngineDetails class, which has a number of properties describing a engine, like its owner, a description, and more. This class is used in the communication between two engines, as a result of calling the GetEngineDetails operation.

Another essential class is DolRequest, which is used as parameter in the ExecuteQuery operation offered by the DOL engine service. Using this parameter a receiving engine is able to determine the Gellish query, a request Guid (which is uniquely created on the engine from which the query originates), a Guid of this original engine, and a list of previously visited engines. Using these Guides, engines can determine if they have already received this request and can ignore it, but also if they can forward the request to a particular engine, based on the Guid list of previously visited engines.

Section 3.3.4 gave an overview of the engine subsystem. Figure 3.4, presented during this overview, depicted the central controller of the DOL engine. The class EngineController is the implementation of this central controller. Using the EngineController applications, as well as instances of the EngineService class, are able to add, retrieve, modify and remove engine settings, like known trackers, data stores, publications, etc. Also the EngineController offers methods for (asynchronously) executing Gellish queries, either local or distributed. Using the known list of peer trackers or engines, the EngineController is also capable of creating a ServiceClient, which allows the controller to execute the operations offered by peer trackers or engines.
Chapter 4. Implementation

![Class diagram of the Gellish class library (Gellikx.Dol.PeerTracker)](image)

Figure 4.9: Class diagram of the Gellish class library (Gellikx.Dol.PeerTracker). The classes within these namespaces are the most generic classes within this project.

The EngineController exposes a number of methods, which start with the characters "Begin". Using these methods a asynchronous method can started, the result of which is one of the AsyncResult classes discussed above. By calling the method starting with the characters "End" and AsyncResult as parameter, the actual result of the asyncronous method can be retrieved.

All application data, like the service settings, engine details, peer trackers, engines, data stores, publications, and subscriptions, can be managed through the EngineApplicationData class. This class is responsible for serializing and deserializing application data to and from a file. Also since this class is implemented with the internal access modifier, it will not be able to call this class directly from outside the Gellikx.Dol.Engine library. Therefore all retrievals and modifications should be directed towards the EngineController.

Using the EngineConfigurationManager communication settings like the commination port and endpoint address can be modified, these values are stored in a config file.

The last classes discussed in this section on the Gellikx.Dol.Engine service library, are used to run the engine as a Windows service. For this reason the EngineWindowsService, which inherits from the .NET ServiceBase class, is created. The EngineWindowsService is responsible for creating and starting the WCF ServiceHost, with an instance of the EngineService. Using the EngineServiceController the Windows service can be controlled, like starting and stopping it. Also this controller can be used to retrieve the status, as well as a number of properties of the specific Windows service, like processId or name.

Gellikx.Dol.PeerTracker

This library is similar to the Gellikx.Dol.Engine service library, but instead of offering classes specific to the DOL engine, this library exposes peer tracker service classes. Just as with the engine service library, a separate setup is creating, allowing users to install the service.

As indicated in the previous section, the contracts for the engine and peer trackers services, in the form of interfaces, are found in the Gellikx.Dol.Engine.Contracts namespace. Therefore the peer tracker service library is dependent on the Gellikx.Dol.Engine library.

All classes found within the various namespaces of this library are equivalents of the classes found in the engine service library. The only difference is the functionalities offered by the peer tracker service, which are only the four operations presented in table 4.4.
4.4. Constructing the Framework

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetTrackerGuid</td>
<td>Returns the Guid of the peer tracker.</td>
</tr>
<tr>
<td>GetTrackerName</td>
<td>The name of the peer tracker is returned to the calling peer.</td>
</tr>
<tr>
<td>Connect</td>
<td>Using the engine details provided as a parameter, the calling engine is added to the list of known DOL engines.</td>
</tr>
<tr>
<td>Disconnect</td>
<td>Removes the calling engine, identified by the given Guid, from the list of known engines.</td>
</tr>
</tbody>
</table>

Table 4.4: List of operations found exposed by the PeerTrackerService class, as defined by the IPeerTracker contract.

4.4.4 Applications

In order to present and interact with the data and operations of the various libraries and services presented above, three applications have been developed. Note that these applications are by no means complete and are only constructed with the goal to evaluate the framework.

All of these applications offer a graphical user interface for interacting with the various class and service libraries, discussed in the previous sections. None of the applications implement additional functionality not yet discussed earlier.

**Gellikx.DolManager**

This application allows users to interact with and manage the service of the DOL engine and/or DOL peer tracker installed locally on their machine. Also the user interface provides the user with a Gellish table viewer, and a query builder, which allow for the construction of a Gellish query.

Within the tree view found in the left pane of this manager application (see figure 4.10), nodes for the peer tracker and engine service are displayed. These nodes represent the peer tracker and engine installed on the local machine. When the peer tracker and or engine services have not been installed, these nodes will be grayed out.

Using the nodes found in the tree view a user can invoke the actions found on the engine...
Chapter 4. Implementation

Figure 4.11: (left) Main interface of the demo definition viewer, used to search for objects. (right) Definition window presenting a single integrated result from a distributed query based on the selected object.

and peer tracker discussed earlier. These actions are summarized below, more details on the actions can be found in the quick start manual found in appendix E.

- Start/stop the engine and peer tracker services
- Change settings like port, endpoint address, name, etc.
- View connected engine (peer tracker only)
- Add or remove trackers, engine, publications and or subscriptions (engine only)
- View subscription on publications and approve/cancel them (engine only)

Gellixx.DemoDefinitionViewer

The second application is an example of a Gellish liaison application, the goal of which is offer the user an interface which translates between Gellish and a particular domain of interest. In case of this definition viewer, the domain of interest is the definition of objects found in the distributed library.

Figure 4.11 shows two screenshots taken of the definition viewer. On the left of the figure the main window can be seen, which enables the user to search for objects based on their name. Once a search string is provided, and the search button is clicked, the application constructs a query to look for objects matching the query. Within this application objects can be found as left objects within Gellish facts containing the specialization or synonym relation type (or any specializations of these types). The Gellish query constructed is forwarded to the local engine.

Once a result is returned from the local engine, the application extracts the left objects from the Gellish facts contained in the answer. The objects are then displayed in the list shown at the bottom of the main definition viewer application window. Apart from displayed object with a matching name, the application is also capable of displaying a specialization tree, displaying all subtypes of the selected object.

A second window available in the application is the definition window. Once an object is selected from the list, and the definition window is opened, the application constructs a new query which is used to retrieve various details about the selected object. One piece of information requested is the supertype of the object, which can be found in the specialization
4.4. Constructing the Framework

![Table]

<table>
<thead>
<tr>
<th>Name of left hand object</th>
<th>Name of relation type</th>
<th>Name of right hand object</th>
</tr>
</thead>
<tbody>
<tr>
<td>selected object</td>
<td>is specialization of</td>
<td>supertype</td>
</tr>
<tr>
<td>supertype</td>
<td>can have as aspect</td>
<td>what aspect</td>
</tr>
<tr>
<td>qualifier</td>
<td>is a qualification of</td>
<td>what aspect</td>
</tr>
<tr>
<td>qualifier</td>
<td>is a qualifying aspect</td>
<td>selected object</td>
</tr>
</tbody>
</table>

Figure 4.12: Example query for retrieving the discriminating aspect compared to a supertype of a selected object.

![Screenshot]

Figure 4.13: Screenshot of demo DOL setup application, used to configure and run multiple engines on the same machine.

fact where the selected object is found on the left side. Other pieces of information are the compositions which contain or are connected to the selected object, found by using the can be part of relation type and can be connected to relation type.

A more complex part of the Gellish query, used to find the object’s definition, is the properties or aspects on which an object discriminates itself from it supertype(s). This piece of the query requires four different relation types, together with two unknown objects. Figure 4.12 display the query which is used to retrieve the discriminating aspect of the selected object.

Once an answer is returned from the engine, the various parts of the definition window are refreshed with the information taken from the facts retrieved.

Gellixx.DemoDolSetup

Finally the demo DOL setup application can be used to create additional instances of the EngineService on the local machine. The primary goal of this application is to run tests, without needing more than one machine (see chapter 5). All engine functionalities, available in the DOL manager application, are also made available within the DOL setup application.

While creating a new engine, the user can select a location where all settings for the engine are saved. Newly created engines can be started directly after they have been constructed, and will operate as long as the DOL setup application is running. The created file can also be used to restore an engine, which was started during a previous execution of the application.

4.4.5 Assembly Versions

Before evaluating the framework using the various applications presented above, this last part of this chapter will give the most recent version number of each of the libraries and/or applications, together with the release date. These number and dates are shown in table 4.5.

59
<table>
<thead>
<tr>
<th>Assembly Name</th>
<th>Version</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gellikx</td>
<td>1.2.0</td>
<td>2008-9-18</td>
</tr>
<tr>
<td>Gellikx.Gellish</td>
<td>2.6.0</td>
<td>2008-9-18</td>
</tr>
<tr>
<td>Gellikx.Dol.Engine</td>
<td>1.1.0</td>
<td>2008-9-18</td>
</tr>
<tr>
<td>Gellikx.Dol.PeerTracker</td>
<td>2.2.0</td>
<td>2008-9-17</td>
</tr>
<tr>
<td>Gellikx.Dol.Windows</td>
<td>1.0.0</td>
<td>2008-9-18</td>
</tr>
<tr>
<td>Gellikx.DolManager</td>
<td>1.0.4</td>
<td>2008-9-18</td>
</tr>
<tr>
<td>Gellikx.DemoDefinitionViewer</td>
<td>1.0.0</td>
<td>2008-10-19</td>
</tr>
<tr>
<td>Gellikx.DemoDolSetup</td>
<td>1.0.0</td>
<td>2008-10-19</td>
</tr>
</tbody>
</table>

Table 4.5: Version numbers, and dates of the most recent releases of libraries and applications.
Chapter 5

Evaluation of Framework

This chapter will take the constructed distributed object library framework, and the support applications, for a test drive. Section 5.1 will give an overview of the test setup, and the construction of the Gellish data stores used during the tests. Following this introduction, section 5.2 will run a number of tests using this setup. Based on the results, the final section of this chapter will highlight limitations of the framework and applications.

5.1 Evaluation Setup

Before discussing the outcome and results from the tests, this section will illustrate how the various engines and peer trackers are set up, and which datasets are used to test the framework and applications. Three different datasets will be used, one of which will also be separated into three different parts, which can then be distributed across the engines found in the distributed network.

5.1.1 Gellish Datasets

Currently the number of available Gellish datasets is increasing. As part of this project datasets found on the project page of Gellish\(^1\), have been converted. During this conversion the Gellish facts, stored in their respective Excel sheets, have been imported into a Gellish SQL Database and also Gellish XML files. The datasets selected from Sourceforge are the Gellish upper ontology, containing the grammar and top of the Gellish specialization hierarchy, and one of the datasets concerned with rotating equipment, containing pumps, compressors, and other objects.

As part of the work, outside the scope of this project, a dataset has also been created containing Gellish facts concerned with pipes, drains, sewage installations, and more. The library is used within a website developed for Rioned and CROW by Idoro. To test the distribution of facts across several engines, all 1209 facts found in the GWB Riolering (GWBR) dataset are separated into three different datasets, resulting in non overlapping sets of facts. The separation is based on the relation type found in each of the facts. Table 5.1 shows the result of this separation.

<table>
<thead>
<tr>
<th>Relation Type</th>
<th>Number of Facts</th>
</tr>
</thead>
<tbody>
<tr>
<td>specialization</td>
<td>400</td>
</tr>
<tr>
<td>synonym</td>
<td>200</td>
</tr>
<tr>
<td>others</td>
<td>609</td>
</tr>
</tbody>
</table>

The first dataset resulting from this separation only contains the specialization and synonym relation types, identified by the UIDs 1146 and 1981. Since this dataset will contain

\(^{1}\text{http://www.sourceforge.net/projects/gellish}\)
Chapter 5. Evaluation of Framework

<table>
<thead>
<tr>
<th>Dataset</th>
<th>UID</th>
<th>Relation Type Name</th>
<th># Facts</th>
</tr>
</thead>
<tbody>
<tr>
<td>basis</td>
<td>1146</td>
<td>is a specialization of</td>
<td>313</td>
</tr>
<tr>
<td></td>
<td>1981</td>
<td>is a synonym for</td>
<td>44</td>
</tr>
<tr>
<td>compositions</td>
<td>1191</td>
<td>can be part of</td>
<td>152</td>
</tr>
<tr>
<td></td>
<td>1407</td>
<td>can be connected to</td>
<td>124</td>
</tr>
<tr>
<td></td>
<td>4989</td>
<td>must be part of</td>
<td>35</td>
</tr>
<tr>
<td>aspects</td>
<td>1726</td>
<td>is a qualification of</td>
<td>147</td>
</tr>
<tr>
<td></td>
<td>2069</td>
<td>can have as aspect</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td>2070</td>
<td>is a qualifying aspect of</td>
<td>188</td>
</tr>
<tr>
<td></td>
<td>4956</td>
<td>must have as aspect</td>
<td>135</td>
</tr>
</tbody>
</table>

Table 5.1: Results from separating the GWB Riolering (GWBR) dataset into three non-overlapping datasets, based on the relation type found in each Gellish fact.

the specialization hierarchy of objects found in the GWBR dataset, together with all synonyms of these objects, this dataset will be called GWBR "basis". Total number of facts present in this dataset is 357.

All facts containing one of the composition relation types will be placed in the GWBR "compositions" dataset. These relation types are either the optional or required assembly relation, having 1191 and 4989 as their UID, or the connection relation type with UID 1407. This second dataset has a total of 311 facts.

Finally the GWBR "aspects" dataset will contain the qualification and aspect relation types, and will consist of 541 facts. The relation types needed for encoding aspect information are: qualification (1726), optional and required possession of aspect by an individual thing (2069 and 2070), and possession of qualitative aspect (2070).

5.1.2 Network Layout

All tests in this chapter will be either performed using a single engine, or using the distributed setup shown in figure 5.1. This setup is designed to test the distribution of Gellish facts across multiple engines, each of which are known to at least one peer tracker. However not all engines are connected to the same peer tracker, the goal of which is to test the forwarding of a Gellish request along more than one connection.

Figure 5.1 shows a distribution of three engines and two peer trackers. Engines $E_0$ through $E_2$ are all connected to peer tracker $T_A$, engine $E_2$ is also connected to tracker $T_B$ together with engine $E_3$. The three separated datasets, the construction of which has been discussed above, will be placed at the various engines, and made available through publications. Initially all these publications will be made public, however in order to test the restriction of protected publication, some publication will also be protected during the tests of section 5.2.4.

The GWBR "basis" data store will be located at engine $E_1$, the "aspects" found in the GWBR dataset will be placed at engine $E_2$, and GWBR "compositions" will be situated at engine $E_3$. Apart from the Gellish TOPini dataset, engine $E_0$ will also have one of the datasets found in the Part 4 collection from SourceForge, containing 5362 facts about rotating equipment, including electrical machines, transport equipment, (un)loading facilities, and solids handling.
5.2 Testing the Framework

The focus during the tests discussed in this section will be on validating the capabilities of the distributed object library framework for sharing information in a distributed fashion, as well as identifying possible shortcomings. Note that none of these tests will focus on performance characteristics of the framework and applications, although important, these test will not be done as part of determining the feasibility of a distributed object library.

5.2.1 Validating Distribution

This first test will validate if all information, or more precisely, all Gellish facts originally found in the GWBR dataset can be successful retrieved from the setup depicted in figure 5.1.

In order to perform this test, a console application is implemented, having references to the Gellykx.Gellish and Gellykx.Dol.Engine libraries. This allows the console application to read the complete GWBR dataset stored in a Gellish XML file. The console application also creates an instance of the EngineController based on the settings of engine $E_0$.

With all of the facts retrieved from the complete GWBR dataset file, the application loops through all of these 1209 facts. For each fact a new DolRequest is created. The query within this request is a copy of the current fact, with one exception: the intention is changed from "true" to "question". Once all other properties of the request have been filled in, the instance of the EngineController is used to send this DolRequest to all engines known to engine $E_0$.

Upon receiving a response back from engine $E_0$, containing a result of either zero or more facts, the console application checks if the original fact is present in this result. If the original fact is indeed found within the received response, that fact has successfully been found in the distributed object library. During the execution of this process a counter is
Chapter 5. Evaluation of Framework

being incremented by one every time a original fact is successfully found in a response.

After the above console application is done executing, the counter indicating the number of times a fact was successfully found in the received result, was equal to the number of facts found in the actual GWBR dataset. This leads to the conclusion that all off the GWBR facts were successfully distributed, and retrieved, through the use of the DOL framework.

5.2.2 Exploding Down

When searching for a particular type of object, any subtype of this object type is also a valid answer. This also applies to the relation types being used in the query, i.e. while searching for facts containing the relation type "common alias for a term for an object" all subtypes of this relation are also valid relation types, and thus the query provider should extend the original query with these objects and relation types.

The first test for retrieving an answer which is exploded down in this manner, is performed only on the E0 engine, and in particular on the dataset containing facts about rotating equipment. For this test the implemented DOL manager application is used. Using the query builder found in this manager application, a query is constructed with the left object UID being equal to 1 (signaling an unknown), a relation type UID equal to 1146 (specialization relation) and the right object UID equal to 130206 (the UID of a "pump" object).

The result returned by the application, after clicking on the "Execute" button, is displayed in the upper part of the manager application’s window. This results consists of 328 facts. After reviewing these facts, they all contain a specialization relation between a subtype of "pump", and either "pump" or a subtype of "pump". In this scenario queries in which explode down is needed, can be successfully performed. However, as will be shown next, a problem occurs within the current DOL implementation if the Gellish TOPini is required for determining inheritance of the relation types, which are currently all defined in this TOPini dataset.

This test is performed on the GWBR dataset, whereby composition, or assembly parts of a "hemelwaterstelsel"2 are looked up. The query used has a left object UID equal to 1, and a right object UID equal to 110000006 ("hemelwaterstelsel"). Since we are looking for parts, the relation type 1191 is used ("can be part of"-relation), however because the relation type 4989 ("must be part of"-relation) is a subtype of the 1191 relation type, facts containing this relation type should also be included in the answer.

After executing this query however, only the 14 facts containing the 1191 relation type are found; even though there exists a fact within the GWBR ("compositions") dataset having "hemelwaterstelsel" as a right object and the "must be part of" as the relation type. After reviewing this case, the reason for this fact not being included in the result, appears to be due to the fact that the specialization relation with the required assembly relation as a left object (4989), and the optional assembly relation type as a right object (1191), are not present at the E3 engine.

A similar problem occurs with our initial explode down example, in which subtypes of a pump are returned, if part of the specialization hierarchy is removed from the Part 4 dataset. For instance, when the single specialization fact between a "dynamic pump" and "pump", the former being the left object and latter the right, is removed from the Part 4 dataset, all subtypes of a "dynamic pump" will also not be found by exploding down from "pump". The removal of part of the hierarchy simulates the situation in which parts of a specialization hierarchy are distributed. This limitation of the current implementation is further discussed in section 5.3.

2Installation designed to collect rain water, and transport this water solely by means of the Earth’s gravity.
5.2.3 Distributed Queries

This third test will use the demo definition viewer application. All previous tests have been concerned with one or more relation types found exclusively at a single data store. However as described in section 4.4.4, a definition requires one or more relation types found in each separated GWBR dataset. For instance the facts found in the GWBR "basis" dataset are needed for finding object having a particular name, whereas facts found in the GWBR "aspects" and "composition" datasets are needed to construct a definition, which will be displayed in the application.

Using the definition viewer all object containing the string "riool" are first looked up. Next the object "gesloten rioolleiding" is selected and the definition window is opened (see figure 4.11). Each part of this window requires facts from a different separated GWBR data store. The result of this distributed query is equal to the result shown on the right side of figure 4.11, which displays the capability of combining the distributed information, retrieved through the framework, within the in a single transparent manner.

5.2.4 Protected versus Public

All tests have so far being executed on information exposed through public publications. However, in order to allow users to restrict the access to (part of) their information the restricted publication type can be used.

By changing the publication type from public to protected on the GWBR "basis" publication, located at engine $E_1$, the query string "riool", together with a partial identical case insensitive commonality, no longer returns any matching objects. In contrast to the 20 objects, which were returned in the previous test using the same input parameters.

Using the demo DOL setup application, engine $E_0$ engine can request a subscription to the restricted GWBR "basis" publication. Once this request has been sent to engine $E_1$, the demo DOL setup application is again used on engine $E_1$ to grant access to engine $E_0$ to the GWBR "basis" publication. A lookup of objects having names partially identical to "riool", again returns the 20 objects, however now this publication is restricted and engine $E_0$ is granted access to the information found in the dataset exposed through the GWBR "basis" publication.

5.3 Limitations Prototype

As a result of the test cases discussed above, a number of problems or limitations have been discovered. These limitations are separated into two sections. The first section presents the limitations of the protocol used by engines and peer tracker services. Other limitations discovered, are closely related to the distribution of Gellish facts and/or the Gellish query provider.

5.3.1 Improved Protocol

One problem currently is that no check is performed if an engine is actually up and running, and thus is accepting requests and capable of answering. When an engine service is stopped during the execution of the above tests, the engine initiating a query will attempt to forward the query to the stopped engine, only when a timeout occurs will the requesting engine Another challenge is the determination of possible very large answers being returned, although this should be possible, applications should also be capable of asking precise questions and only if the answer does not satisfy should another (again precise) question be asked.
Chapter 5. Evaluation of Framework

Also the need to explicitly add an engine is something that needs to be revisited. When the discovery process is hidden to the user and only performed just before a query is executed, the distributed character can be hidden more effectively from the user.

Finally the framework does not implement any means of security and/or encryption. In order to protect possible sensitive information from being intercepted by the wrong parties, these issues should be implemented.

5.3.2 Gellish Interpretation

Apart from the limitations of the protocol a number of limitations have become apparent related to the distribution of Gellish facts. One of these problems is the placements of the upper ontology, or TOPini. Since a number of basic Gellish facts are required to take advantage of the inheritance for instance, these facts need to be present on all engines.

Also another issue is that a number of queries can only be satisfied if and only if enough facts are present on a particular engine. Take the example executed in the above setup, in which the definition of a particular object is retrieved. The discriminator information needs a number of relation types to be present (see figure 4.12 in previous chapter). All four of these relation types should be found in a single data store if an answer is to be returned. A similar problem occurs if the parts of a specialization hierarchy are not located at a particular engine, needing this information to determine subtypes.
Chapter 6
Conclusions and Future Work

The next section will revisit the research questions set forth at the start of this project. Based on the tests results gathered within 5, in section 6.2 recommendations will be given on how they might be resolved in the future. This chapter will be concluded with identifying future work on the distributed object library framework and applications.

6.1 Conclusions

At the outset of this project two research questions were posed, which will be revisited here in light of building a framework on which distributed object libraries can be constructed using Gellish.

The first question is related to the possibility of classifying ontology modeling languages. The conceptualization graph, presented in chapter 2, allows ontology modeling languages to be compared, based on the subject and structure of their conceptualization. Using the conceptualization graph, statements can be made about the possibility of translating knowledge, modeled in one language, into a representation of that knowledge in another language. When translating from one language into a language of less structure, some of the knowledge will not be able to be represented by this second language. Similarly, translation will not be complete, when translating from a language with a more generic subject of conceptualization, into a language which does not contain possibilities for representing the generic constructs found in the original modeling language.

The candidate languages, presented in section 2.3, are all used by actors within the building and construction industry to conceptualize building objects, found in the real world during the stages of their lifecycle. By placing these languages within the conceptualization graph, a picture could be drawn about the relative structure and subject of their conceptualization. Because translation between different modeling languages, is outside the scope of this project, a single modeling language needed to be chosen for conceptualizing the knowledge stored throughout the distributed data stores. This lead to the selection of Gellish in section 2.4, based on the facts that Gellish supports an expressive structures of conceptualization, but also has a generic subject of conceptualization.

Based on the requirements presented in section 3.1, and by taking into a number of design goals, the distributed object library framework and applications were designed. The goal of the framework and applications built on top of it, is to enable its users to share information in a transparent and coherent fashion.

All information is stored in either Gellish XML files, having the .gml extension, or
within SQL databases. The Gellish XML schema and resulting .gml files, as well as the Gellikx.Gellish assembly, are currently also being used in other (web)applications and web services developed by Idoro, where they form the basis of storing and retrieving Gellish information.

Implementation of the distributed object library framework resulted in two services: the DOL peer tracker and DOL engine services. The peer tracker is used to create a distributed network, by functioning as the white pages to engines wishing to participate in the distributed object library. Engines on the other hand are the actual workers, since they keep track of the trackers they are connected to, maintain the data stores of information, offer publications, allow subscription to be acquired. However the most important function of the engine, is the capability to: accept queries for a particular piece of information, forward them to any engine who has not yet received the query, and return results of these activities back to the requesting application.

From the results of the tests, performed during the evaluation of the framework and application in chapter 5, it is clear that the framework is capable of storing and retrieving Gellish facts, distributed across a network. Also the query provider, which is part of the engine, can answer questions about unknowns, take into account inheritance by exploding down, as well as answer composed questions of more than one Gellish fact.

All libraries and applications are created as separate libraries and follow the .NET naming guidelines, which makes extending them in the future easier, since .NET developers will be familiar with these guidelines. Although the framework and applications have been implemented using the Microsoft .NET framework, and released to run on the Windows OS, projects exist, like Mono\(^1\), which allow .NET and Winforms applications to run on other operating systems, like Linux and Mac OS. Since all communication from and to the framework is done via SOAP messages, which are platform independent, other platforms will be capable of interacting with the framework.

### 6.2 Recommendations

Based on the limitations presented in section 5.3 the developed distributed object library framework and applications needs to be improved on a number of aspects. These improvements are concerned with the protocol used, like automatic discovery, the ability to detect when peers are offline, and additional implementation of security and encryption. All these improvements could be easily implemented in future versions of distributed object library framework and applications.

Apart from improvements needed for the protocol, further research is also needed in the area of resolving conflicts from differing or contra dictionary facts. The problem of not having a particular set of facts present at a engine, which is needed to compose an answer, illustrated in section 5.2, also needs to be addressed. One solution could be to remove the query provider from remote engine, and let the requesting engine generate new questions every time part of the hierarchy is present, until no responses are returned. This solution will however increase the number of requests which will be sent between engines, and as a consequence negatively influence performance.

Finally the availability of a upper ontology, or Gellish TOPini, needs to be addressed, since this dataset contains the most generic concepts. This problem could be solved by either automatically downloading the latest version from other engines, or by have a dedicated engine making the TOPini available (the former of which is to be preferred in a truly distributed application).

---

\(^1\)http://www.mono-project.com
6.3 Future Work

Once these issues have been dealt with, the next phase of development should focus on analyzing the performance of the distributed object library framework.

6.3 Future Work

Although the results so far look promising, the recommendations given in the previous section make clear that further research and development is needed to improve the abilities of the distributed object library.

Further developments will be split into two separate areas. One of these areas is the Gellish .NET framework, contained within the Gellikx.Gellish assembly. After further improving this Gellish .NET framework, this library will be made freely available by Idoro to other developers of applications, based on Gellish libraries. All improvements made to this framework will be inline with the Gellish@work methodology developed by Andries van Renssen and Matthé van Koetsveld.

As for the engine and peer tracker services found within the framework, together with the DOL manager application, future development will be concerned with tackling the shortcomings discussed above. These improvements will initially be done by Idoro, but may be contracted out in later stages if required.
Bibliography


[Che08] Cheobs, CROWN. Ontwikkeling Basismodule Cheobs - PvE, June 2008.


Chapter 6. BIBLIOGRAPHY


Appendices
Appendix A

Gellish Table Column Definitions

This table, taken from [VR08], describes the Gellish columns found in a Gellish table, and also defines the valid value space, whether the column is obligatory, and their respective SQL and XML datatypes. See the section on persistent data (4.3) for further elaboration.

<table>
<thead>
<tr>
<th>UID</th>
<th>Name</th>
<th>Value Space</th>
<th>Optional</th>
<th>SQL Type</th>
<th>XML Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Presentation key</td>
<td>textual (non-Unicode)</td>
<td>obligatory</td>
<td>int</td>
<td>token</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A presentation key (or sequence number) indicates a position or field in a presentation structure, such as a spreadsheet or a list of lines (or sequence number). It can support sorting the content of a Gellish table. It has no contribution to the meaning of the facts represented on the line. The presentation key does not effect the meaning of the lines. This column can be arbitrarily filled-in for use in a specific context.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>69</td>
<td>Unique language identifier</td>
<td>integer (64 bit)</td>
<td>optional</td>
<td>bigint</td>
<td>uid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The unique identifier of the language in which the name of the left hand object (see column 101) and the name of the relation type (see column 3) is spelled and, if present, in which the definition (see column 63 and 4) is spelled. The language is a context for the origin of the referencing relation between the UID and the string that is the name.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>Name of language of left object name</td>
<td>textual (Unicode)</td>
<td>optional</td>
<td>nvarchar(255)</td>
<td>token</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The name of the language of the left hand object name indicates the name of the language for which a UID is given in column 69 and that is a context for the name of the left hand object (see column 101) and the name of the relation type (see column 3). If the relation type name is not available in that language, it may be given in English. The allowed values for the language name are the names defined in the Gellish Dictionary (or your private extension). Currently the dictionary contains names of natural languages and of (artificial) programming languages.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Unique left hand object identifier</td>
<td>integer (64 bit)</td>
<td>obligatory</td>
<td>bigint</td>
<td>uid</td>
</tr>
</tbody>
</table>
Appendix A. Gellish Table Column Definitions

A unique left hand object identifier is the identifier of the main object about which the line defines a fact. That main fact is an association between two objects mentioned in column 2 and 15. The external identifier (name) of the object in column 2 can be given in column 56 with its text attribute in column 101 'name of left hand object'.

A UID is an artificial sequence number, provided it is unique in a managed context. For example, the UID 4724 is a reference number of a telephone extension in the context of my company in The Hague. An identical number may refer to a different object in a different context, such as the extension with UID 4724 in the context of your company. The uniqueness context is given in column 16 (subject area). Such a context itself is defined on a separate line in a Gellish database table.

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>71</td>
<td>Uniqueness context identifier for left name</td>
</tr>
<tr>
<td></td>
<td>integer (64 bit)</td>
</tr>
<tr>
<td></td>
<td>The uniqueness context identifier for left hand object name, also called the identifier of the language community, provides the context within which the left hand object name in column 101 is a unique reference to the object id in column 2, in addition to the language context (see column 69 and 54). The context is superfluous (and is for human clarification only) on all lines other than lines with a specialization, a qualification a classification or an alias relation and their subtypes, because only there the left hand objects, identified by their UID, are defined to have a name. If no context is given on a definition line, then the name for the left hand object is unique in the whole (natural) language (column 54) and no homonyms are then allowed (in the Gellish Dictionary).</td>
</tr>
<tr>
<td>16</td>
<td>Uniqueness context name for left name</td>
</tr>
<tr>
<td></td>
<td>textual (Unicode)</td>
</tr>
<tr>
<td></td>
<td>The uniqueness context name for left hand object name is the name for the uniqueness context of which the identifier is given in column 71. The name is optional (and is for human clarification only) because the context UID in column 71 shall be a reference to a context that is defined on another line, where its UID and name appears in columns 2 and 101 respectively.</td>
</tr>
<tr>
<td>38</td>
<td>Left hand object type name</td>
</tr>
<tr>
<td></td>
<td>textual (Unicode)</td>
</tr>
<tr>
<td></td>
<td>An object type of the left hand object (with the UID in column 2) indicates the name of the entity type of the left hand object in a particular data model about which the line defines the main fact. This column is superfluous in Gellish as it can be inferred via inheritance from the mapping of the appropriate object or its classifying class in the Gellish specialization hierarchy to the entity in appropriate data model.</td>
</tr>
<tr>
<td>39</td>
<td>Reality</td>
</tr>
<tr>
<td></td>
<td>textual (Unicode)</td>
</tr>
<tr>
<td></td>
<td>The reality is a classification of the left hand object, being either imaginary or materialized (= real). This indicates that the object is either a product of the mind or an object whose existence is based in the physical world, either as natural or as artificial object. If not specified, then the reality shall be interpreted from the context or from a explicit classification fact. For example, during design a pump will be an imaginary (although realistic) object, when fabricated a pump will be a materialized object. Note that an object cannot be imaginary and materialized. An installation relation relates an imaginary object to a materialized object. Note: Classes are always imaginary.</td>
</tr>
<tr>
<td>44</td>
<td>Left hand object cardinalities</td>
</tr>
<tr>
<td></td>
<td>textual (non-Unicode)</td>
</tr>
<tr>
<td></td>
<td>For common associations between classes this column contains the simultaneous cardinalities for the left hand object class. This means that it indicates the minimum and maximum number of members of the class that can be associated with a member of the right hand object class at the same time. The cardinalities may be specified by: a comma separated list of two integers that indicate the lower and upper limit cardinalities. The upper limit may be the character 'n' to indicate that the upper limit is unlimited.</td>
</tr>
<tr>
<td>101</td>
<td>Left hand object name</td>
</tr>
<tr>
<td></td>
<td>textual (Unicode)</td>
</tr>
</tbody>
</table>
Appendix A. Gellish Table Column Definitions

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Type</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>72</td>
<td>Identifier of left hand role</td>
<td>integer (64 bit) optional bigint uid</td>
<td>An identifier of left hand role identifies the role that is played by the left hand object in column 2. This role is implicitly classified or is implicitly a subtype of the first or second kind of role that is required by the kind of relation in column 60.</td>
</tr>
<tr>
<td>73</td>
<td>Name of left hand role</td>
<td>textual optional nvarchar(255) token</td>
<td>A name of left hand role is the name of the role in column 72.</td>
</tr>
<tr>
<td>43</td>
<td>Intention</td>
<td>textual (Unicode) optional varchar(255) intention</td>
<td>An intention indicates the extent to which the main fact is the case or is the case according to the author of a proposition. An intention includes also a level of truth. If a line expresses a proposition or communication fact, then the intention qualifies the proposition. If a line expresses a fact, then the intention indicates whether the relation of the type is true or false (not) or questionable (maybe). For example, the intention may indicate that a proposition is an affirmative request (question), confirmation, promise, declination, statement, denial, probability or acceptance. Default = &quot;true&quot;, which means a qualification of the statement: this fact &quot;is the case&quot;.</td>
</tr>
<tr>
<td>19</td>
<td>Unique identifier of validity context for main fact</td>
<td>integer (64 bit) optional bigint uid</td>
<td>The unique identifier of validity context for main fact identifies the context within which the fact id, given in column 1, represents a valid fact. If not given, the fact is valid in all contexts.</td>
</tr>
<tr>
<td>18</td>
<td>Validity context name</td>
<td>textual (Unicode) optional nvarchar(255) token</td>
<td>The validity context name provides a name of the context that is identified in column 19.</td>
</tr>
<tr>
<td>1</td>
<td>Unique identifier of main fact</td>
<td>integer (64 bit) obligatory bigint uid</td>
<td>A unique main fact identifier is an identifier of the main fact that is represented on the line (such as an association or possession relationship). This main fact is of the type as indicated in column 3 &quot;relation type name&quot;.</td>
</tr>
<tr>
<td>60</td>
<td>Relation type ID</td>
<td>integer (64 bit) obligatory bigint uid</td>
<td>A relation type ID is unique ID for the class that qualifies the fact in column 1, whereas a name of the type of relation is given in Gellish in column 3.</td>
</tr>
<tr>
<td>3</td>
<td>Relation type name</td>
<td>textual (Unicode) obligatory nvarchar(255) token</td>
<td>A relation type name (or fact type name) is a name of one of the subtypes of relation or class of relation expressed in Gellish English.</td>
</tr>
<tr>
<td>74</td>
<td>Identifier of right hand role</td>
<td>integer (64 bit) optional bigint uid</td>
<td>An identifier of right hand role identifies the role that is played by the right hand object in column 15. This role is implicitly classified or is implicitly a subtype of the first or second kind of role that is required by the kind of relation in column 60.</td>
</tr>
<tr>
<td>75</td>
<td>Name of right hand role</td>
<td>textual (Unicode) optional nvarchar(255) token</td>
<td>A name of right hand role is the name of the role in column 74.</td>
</tr>
<tr>
<td>15</td>
<td>Right hand unique object identifier</td>
<td>integer (64 bit) obligatory bigint uid</td>
<td>A name of right hand role is the name of the role in column 74.</td>
</tr>
</tbody>
</table>
Appendix A. Gellish Table Column Definitions

A right hand unique object identifier is the UID of the object associated with the object in column 2. The name of this right hand object can (optionally) be given as right hand term in column 201. The name of an object that has a name is defined only on a line where the fact type indicates a referencing association to the object. On other lines a filled in name is only meant to support human readability. For dates in column 15/201 there is a Gellish convention that the UID for dates between the year 1500 and 3000 is an integer number that is a concatenation of four digits for the year, two for the month and two for the day, whereas two zero's are used for the month when a whole year is meant and two zero's for the day when a whole month is meant. For example, January 2006 has UID 20060100.

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Type</th>
<th>Length</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>Right hand object cardinalities</td>
<td>textual (non-Unicode)</td>
<td>optional</td>
<td>varchar(32)</td>
</tr>
<tr>
<td></td>
<td>For common associations between classes this column contains the simultaneous cardinalities for the right hand object class. This means that it indicates the minimum and maximum number of members of the class that can be associated with a member of the left hand object class at the same time. The cardinalities may be specified in the same way as for the left hand object.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>Description of main fact (template text)</td>
<td>textual (Unicode)</td>
<td>optional</td>
<td>nvarchar(255)</td>
</tr>
<tr>
<td></td>
<td>A description of the main fact (column 1) is meant to be presented to a user. The text is intended as an aid for interpretation of the meaning of the main fact in its context and may imply an instruction to a user for what should be filled in as a value for the right hand term or what should be selected from a pick list in order to finalise a fact or group of facts. The text might appear on a user interface (e.g. a fill-in-the-blanks form or data sheet) and supports human understanding of the meaning of the fact(s) and the intention of the object in column 15 and 201 and optionally the UoM in column 7. For example: the text “temperature of the fluid at inlet” suggests that a value and a unit of measure should be supplied.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>201</td>
<td>Right hand object name</td>
<td>textual (Unicode)</td>
<td>obligatory</td>
<td>nvarchar(255)</td>
</tr>
<tr>
<td></td>
<td>A right hand object name is a string or value which is a textual name of the object identified in column 15, and which is associated with the object in column 2 with a name in column 101. For example, a tag name or code, numeric value, class name or free text description.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>Partial description</td>
<td>textual (Unicode)</td>
<td>optional</td>
<td>ntext</td>
</tr>
<tr>
<td></td>
<td>A partial description is a description that together with the relation type name (column 3) and the right hand object name (column 201) forms a full definition as presented in column 4.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Full definition</td>
<td>textual (Unicode)</td>
<td>optional</td>
<td>ntext</td>
</tr>
<tr>
<td></td>
<td>A full definition is a textual description of the characteristics that identify the left hand object or members of the left hand object class. Typically this is a concatenation of the term “is a(n)”, the right hand object name and the text in column 63 (partial description).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66</td>
<td>Unit of measure identifier</td>
<td>integer (64 bit)</td>
<td>optional</td>
<td>bigint</td>
</tr>
<tr>
<td></td>
<td>The unit of measure identifier identifies the scale used for interpretation of the numeric value of a property in column 201. In case column 201 contains a concept of property name, the indicated UoM UID in column 66 indicates the default.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Unit of measure name</td>
<td>textual (Unicode)</td>
<td>optional</td>
<td>nvarchar(255)</td>
</tr>
<tr>
<td></td>
<td>The unit of measure name is the name of the scale used for interpretation of the numeric value of a property in column 201. In case column 201 contains a concept of property name, the indicated UoM in column 7 is a name of the default.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>76</td>
<td>Accuracy of mapping UID</td>
<td>integer (64 bit)</td>
<td>optional</td>
<td>bigint</td>
</tr>
</tbody>
</table>
Appendix A. Gellish Table Column Definitions

<table>
<thead>
<tr>
<th>Column No.</th>
<th>Column Name</th>
<th>Data Type</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>77</td>
<td>Accuracy of mapping name</td>
<td>textual (Unicode)</td>
<td>optional</td>
<td>The accuracy of mapping name is a name for the concept in column 76.</td>
</tr>
<tr>
<td>70</td>
<td>Picklist UID (Domain UID)</td>
<td>integer (64 bit)</td>
<td>optional</td>
<td>The unique identifier for the collection of objects from which values for instances of the right hand term may be selected in the context of an instance of the left hand term. Note, this column (together with column 20) is meant as a short-cut for subtyping a (right hand) aspect type in the context of the left hand object and adding an additional line which defines that the value for a subtype &quot;shall be one of the&quot; picklist collection of aspect values.</td>
</tr>
<tr>
<td>20</td>
<td>Picklist name</td>
<td>textual (Unicode)</td>
<td>optional</td>
<td>The name of a picklist or domain identified by the Picklist UID in column 70. The name of the picklist shall be unique in the same context as the context for the right hand term (column 201) as defined in column 16 on the line where the right hand term is defined and occurs as a left hand term.</td>
</tr>
<tr>
<td>14</td>
<td>Remarks</td>
<td>textual (Unicode)</td>
<td>optional</td>
<td>A remarks field is intended for comments related to the fact or the existence of the left hand object, its definition or status.</td>
</tr>
<tr>
<td>8</td>
<td>Approval status of main fact</td>
<td>textual (non-Unicode)</td>
<td>obligatory</td>
<td>An approval status indicates the status of the main fact. The status of the other facts on a line can be derived from the status of the main fact. A status can be any of the qualifications of 'approval status' in STEPlib. For example: proposed, issue, deleted, proposed to be deleted, ignore, agreed, accepted, accepted association (= only the main fact is accepted), or replaced (see also the &quot;Guide on STEPlib&quot;). The status 'replaced' indicates that the main fact is deleted and that a succeeding fact (see column 64) exists. The reason of the status may be clarified in the remarks column (see column 14).</td>
</tr>
<tr>
<td>67</td>
<td>UID of successing fact</td>
<td>integer (64 bit)</td>
<td>optional</td>
<td>The UID of the fact by which this line, and especially the main fact which UID is given in column 1, is replaced when the status in column 8 is &quot;replaced&quot;. It indicates that there exists a succession relation between the two facts. Note: If the relation type is the last classification relation or specialization relation for the left hand object, then the life of the left hand object is terminated and replaced by the left hand object of the succeeding relation.</td>
</tr>
<tr>
<td>9</td>
<td>Date of start of validity</td>
<td>date/time</td>
<td>obligatory</td>
<td>A date of start of life is the moment of the begin of the validity of the main fact. It is implicitly associated with the main fact via a &quot;valid since&quot; relation. The '1900 date system' enables very accurate timestamps, for example for the recording of moments of measurement.</td>
</tr>
<tr>
<td>10</td>
<td>Date of latest change</td>
<td>date/time</td>
<td>obligatory</td>
<td>A date of latest change indicates the latest change of one of the auxiliary facts. If the status in column 8 is &quot;deleted&quot;, &quot;replaced&quot; or &quot;history&quot;, then the data of latest change indicates the moment of the end of the validity of the main fact. Then it is assumed to be related to the main fact by a &quot;valid until&quot; relation.</td>
</tr>
<tr>
<td>12</td>
<td>Author of latest change</td>
<td>textual (Unicode)</td>
<td>optional</td>
<td></td>
</tr>
</tbody>
</table>
## Appendix A. Gellish Table Column Definitions

The person who is the originator of the proposition or of the expression of the fact and who has (limited) responsibility for the content of the line; especially its latest change. It is good practice to provide this information, although strictly speaking it is optional.

<table>
<thead>
<tr>
<th>Reference or source</th>
<th>textual (Unicode)</th>
<th>optional</th>
<th>nvarchar(255)</th>
<th>token</th>
</tr>
</thead>
</table>

One or more organisations, persons or positions in organisations or (parts of) documents that act as the source or point of reference for the main fact. It is good practice to provide this information, although strictly speaking it is optional. It may include URI strings.

<table>
<thead>
<tr>
<th>Line identifier</th>
<th>integer (64 bit)</th>
<th>optional</th>
<th>bigint</th>
<th>uid</th>
</tr>
</thead>
</table>

A line id (UID-5) is the identifier for a single row in a Gellish database table. It indicates the collection of facts (or 'cloud' of related things) in which the main fact and the auxiliary facts on one single line in a Gellish database table are included. It may be used to distinguish different expressions of the same fact (with the same fact UID (see column 1)). For example, to distinguish the same fact expressed in different languages.

<table>
<thead>
<tr>
<th>Unique plural fact identifier</th>
<th>integer (64 bit)</th>
<th>optional</th>
<th>bigint</th>
<th>uid</th>
</tr>
</thead>
</table>

A unique plural fact id is a unique identifier of a collection of facts in which the fact as identified in column 1 is included. This column is intended to indicate a collection of which the elements are facts that are identified by the above mentioned unique main fact identifiers (UID-1). A plural fact identifier is typically used as an identifier of a model or (sub) template or view.

When a plural fact identifier is filled-in, it implies the existence of an inclusion relation (.. is an element of ..) between the main fact on this line identified in column number 1 and the collection of facts identified in column number 50. The name of the collection may be given in column 68. Collections may appear as left hand or right hand objects in main facts, for example in the definition of larger collections.

<table>
<thead>
<tr>
<th>Name of collection of facts</th>
<th>textual (Unicode)</th>
<th>optional</th>
<th>nvarchar(255)</th>
<th>token</th>
</tr>
</thead>
</table>

The name of collection of facts indicates the collection of main facts in a Gellish database table, which collection is identified by the UID in column 50. The facts in the collection might be managed together. The main fact on the line is an element of the collection. The collection may indicate for example an area of responsibility of a peer group, the content of a table or the facts on a data sheet.

<table>
<thead>
<tr>
<th>Left hand string commonality</th>
<th>textual (non-Unicode)</th>
<th>optional</th>
<th>varchar(64)</th>
<th>stringCommonality</th>
</tr>
</thead>
</table>

A left hand string commonality specifies for a query in which way a search string has or shall have commonality with a target string that is a left hand object name. A string commonality may have one of the allowed values that are specified as qualifications of a string commonality in the Gellish Dictionary. For example: case (in)sensitive identical, case (in)sensitive partially identical, case (in)sensitive front end identical, case (in)sensitive different, equal, unequal, less than or equal, greater than or equal.

The default in a query is 'case sensitive partially identical'.

<table>
<thead>
<tr>
<th>Right hand string commonality</th>
<th>textual (non-Unicode)</th>
<th>optional</th>
<th>varchar(64)</th>
<th>stringCommonality</th>
</tr>
</thead>
</table>

A right hand string commonality specifies for a query in which way a search string has or shall have commonality with a target string that is a right hand object name.

<table>
<thead>
<tr>
<th>Reference or source</th>
<th>textual (Unicode)</th>
<th>optional</th>
<th>nvarchar(255)</th>
<th>token</th>
</tr>
</thead>
</table>

One or more organisations, persons or positions in organisations or (parts of) documents that act as the source or point of reference for the main fact. It is good practice to provide this information, although strictly speaking it is optional. It may include URI strings.

<table>
<thead>
<tr>
<th>Line identifier</th>
<th>integer (64 bit)</th>
<th>optional</th>
<th>bigint</th>
<th>uid</th>
</tr>
</thead>
</table>

A line id (UID-5) is the identifier for a single row in a Gellish database table. It indicates the collection of facts (or 'cloud' of related things) in which the main fact and the auxiliary facts on one single line in a Gellish database table are included. It may be used to distinguish different expressions of the same fact (with the same fact UID (see column 1)). For example, to distinguish the same fact expressed in different languages.

<table>
<thead>
<tr>
<th>Unique plural fact identifier</th>
<th>integer (64 bit)</th>
<th>optional</th>
<th>bigint</th>
<th>uid</th>
</tr>
</thead>
</table>

A unique plural fact id is a unique identifier of a collection of facts in which the fact as identified in column 1 is included. This column is intended to indicate a collection of which the elements are facts that are identified by the above mentioned unique main fact identifiers (UID-1). A plural fact identifier is typically used as an identifier of a model or (sub) template or view.

When a plural fact identifier is filled-in, it implies the existence of an inclusion relation (.. is an element of ..) between the main fact on this line identified in column number 1 and the collection of facts identified in column number 50. The name of the collection may be given in column 68. Collections may appear as left hand or right hand objects in main facts, for example in the definition of larger collections.

<table>
<thead>
<tr>
<th>Name of collection of facts</th>
<th>textual (Unicode)</th>
<th>optional</th>
<th>nvarchar(255)</th>
<th>token</th>
</tr>
</thead>
</table>

The name of collection of facts indicates the collection of main facts in a Gellish database table, which collection is identified by the UID in column 50. The facts in the collection might be managed together. The main fact on the line is an element of the collection. The collection may indicate for example an area of responsibility of a peer group, the content of a table or the facts on a data sheet.

<table>
<thead>
<tr>
<th>Left hand string commonality</th>
<th>textual (non-Unicode)</th>
<th>optional</th>
<th>varchar(64)</th>
<th>stringCommonality</th>
</tr>
</thead>
</table>

A left hand string commonality specifies for a query in which way a search string has or shall have commonality with a target string that is a left hand object name. A string commonality may have one of the allowed values that are specified as qualifications of a string commonality in the Gellish Dictionary. For example: case (in)sensitive identical, case (in)sensitive partially identical, case (in)sensitive front end identical, case (in)sensitive different, equal, unequal, less than or equal, greater than or equal.

The default in a query is 'case sensitive partially identical'.

<table>
<thead>
<tr>
<th>Right hand string commonality</th>
<th>textual (non-Unicode)</th>
<th>optional</th>
<th>varchar(64)</th>
<th>stringCommonality</th>
</tr>
</thead>
</table>

A right hand string commonality specifies for a query in which way a search string has or shall have commonality with a target string that is a right hand object name.

<table>
<thead>
<tr>
<th>Reference or source</th>
<th>textual (Unicode)</th>
<th>optional</th>
<th>nvarchar(255)</th>
<th>token</th>
</tr>
</thead>
</table>

One or more organisations, persons or positions in organisations or (parts of) documents that act as the source or point of reference for the main fact. It is good practice to provide this information, although strictly speaking it is optional. It may include URI strings.

<table>
<thead>
<tr>
<th>Line identifier</th>
<th>integer (64 bit)</th>
<th>optional</th>
<th>bigint</th>
<th>uid</th>
</tr>
</thead>
</table>

A line id (UID-5) is the identifier for a single row in a Gellish database table. It indicates the collection of facts (or 'cloud' of related things) in which the main fact and the auxiliary facts on one single line in a Gellish database table are included. It may be used to distinguish different expressions of the same fact (with the same fact UID (see column 1)). For example, to distinguish the same fact expressed in different languages.

<table>
<thead>
<tr>
<th>Unique plural fact identifier</th>
<th>integer (64 bit)</th>
<th>optional</th>
<th>bigint</th>
<th>uid</th>
</tr>
</thead>
</table>

A unique plural fact id is a unique identifier of a collection of facts in which the fact as identified in column 1 is included. This column is intended to indicate a collection of which the elements are facts that are identified by the above mentioned unique main fact identifiers (UID-1). A plural fact identifier is typically used as an identifier of a model or (sub) template or view.

When a plural fact identifier is filled-in, it implies the existence of an inclusion relation (.. is an element of ..) between the main fact on this line identified in column number 1 and the collection of facts identified in column number 50. The name of the collection may be given in column 68. Collections may appear as left hand or right hand objects in main facts, for example in the definition of larger collections.

<table>
<thead>
<tr>
<th>Name of collection of facts</th>
<th>textual (Unicode)</th>
<th>optional</th>
<th>nvarchar(255)</th>
<th>token</th>
</tr>
</thead>
</table>

The name of collection of facts indicates the collection of main facts in a Gellish database table, which collection is identified by the UID in column 50. The facts in the collection might be managed together. The main fact on the line is an element of the collection. The collection may indicate for example an area of responsibility of a peer group, the content of a table or the facts on a data sheet.

<table>
<thead>
<tr>
<th>Left hand string commonality</th>
<th>textual (non-Unicode)</th>
<th>optional</th>
<th>varchar(64)</th>
<th>stringCommonality</th>
</tr>
</thead>
</table>

A left hand string commonality specifies for a query in which way a search string has or shall have commonality with a target string that is a left hand object name. A string commonality may have one of the allowed values that are specified as qualifications of a string commonality in the Gellish Dictionary. For example: case (in)sensitive identical, case (in)sensitive partially identical, case (in)sensitive front end identical, case (in)sensitive different, equal, unequal, less than or equal, greater than or equal.

The default in a query is 'case sensitive partially identical'.

<table>
<thead>
<tr>
<th>Right hand string commonality</th>
<th>textual (non-Unicode)</th>
<th>optional</th>
<th>varchar(64)</th>
<th>stringCommonality</th>
</tr>
</thead>
</table>

A right hand string commonality specifies for a query in which way a search string has or shall have commonality with a target string that is a right hand object name.

Table A.1: Gellish column definitions, including the SQL and XML datatypes and additional constraints (taken from Van Renssen’s *Definition of Gellish Databases* [VR08]).
Appendix B

SQL Script

This appendix presents the SQL script used to create an empty SQL table, containing the columns as specified in section 4.3.

```sql
CREATE TABLE [dbo].[EmptyGellishTable](
    [Sequence] [int] NULL,
    [Uid] [bigint] NOT NULL,
    [LineUid] [bigint] NULL,
    [SuccessorUid] [bigint] NULL,
    [ValidityContextUid] [bigint] NULL,
    [ValidityContextName] [nvarchar](255) NULL,
    [LanguageUid] [bigint] NULL,
    [LanguageName] [nvarchar](255) NULL,
    [LeftNameContextUid] [bigint] NULL,
    [LeftNameContextName] [nvarchar](255) NULL,
    [LeftObjectUid] [bigint] NULL,
    [LeftObjectName] [nvarchar](255) NOT NULL,
    [LeftObjectTypeUid] [nvarchar](255) NULL,
    [LeftObjectRoleUid] [bigint] NULL,
    [LeftObjectRoleName] [nvarchar](255) NULL,
    [LeftObjectCardinalities] [varchar](32) NULL,
    [Reality] [nvarchar](255) NULL,
    [Intention] [varchar](255) NOT NULL,
    [RelationTypeUid] [bigint] NOT NULL,
    [RelationTypeName] [nvarchar](255) NOT NULL,
    [RightObjectUid] [bigint] NULL,
    [RightObjectName] [nvarchar](255) NOT NULL,
    [RightObjectRoleUid] [bigint] NULL,
    [RightObjectRoleName] [nvarchar](255) NULL,
    [RightObjectCardinalities] [nvarchar](32) NULL,
    [Description] [ntext] NULL,
    [PartialDefinition] [ntext] NULL,
    [FullDefinition] [ntext] NULL,
    [Remarks] [ntext] NULL,
    [UnitOfMeasureUid] [bigint] NULL,
    [UnitOfMeasureName] [nvarchar](32) NULL,
);
Appendix B. SQL Script

[MappingAccuracyUid] [bigint] NULL,
[MappingAccuracyName] [nvarchar](255) NULL,
[CollectionUid] [bigint] NULL,
[CollectionName] [nvarchar](255) NULL,
[PicklistUid] [bigint] NULL,
[PicklistName] [nvarchar](255) NULL,
[ApprovalStatus] [nvarchar](50) NOT NULL,
[EffectiveFrom] [datetime] NOT NULL,
[LatestUpdate] [datetime] NOT NULL,
[Author] [nvarchar](255) NULL,
[Reference] [nvarchar](255) NULL
)}
Appendix C

Gellish XML Schema

The following XML Schema describes how a valid Gellish XML file (or GML file) should be formatted and which values ranges are to be used. All parts of this schema are discussed extensively in section 4.3.

```xml
<?xml version="1.0" encoding="utf-8"?>
<xs:schema targetNamespace="http://gellikx.com/2008/ns/1.0/GellishSchema"
 xmlns:gellish="http://gellikx.com/2008/ns/1.0/GellishSchema"
 xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="unqualified">

<!-- Simple types -->
<xs:simpleType name="uid">
  <xs:restriction base="xs:long">
    <xs:minInclusive value="0"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="cardinalities">
  <xs:restriction base="xs:token">
    <xs:pattern value="((0|1−9|0−9)∗|0|1−9|0−9)∗|(0|1−9|0−9)∗n)"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="status">
  <xs:restriction base="xs:token">
    <xs:enumeration value="proposed"/>
    <xs:enumeration value="issue"/>
    <xs:enumeration value="deleted"/>
    <xs:enumeration value="proposed to be deleted"/>
    <xs:enumeration value="ignore"/>
    <xs:enumeration value="agreed"/>
    <xs:enumeration value="accepted"/>
    <xs:enumeration value="accepted association"/>
    <xs:enumeration value="replaced"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="reality">
  <xs:restriction base="xs:NMTOKEN">
    <xs:enumeration value="imaginary"/>
    <xs:enumeration value="materialized"/>
    <xs:enumeration value="real"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="intention">
  <xs:restriction base="xs:NMTOKEN">
    <xs:enumeration value="true"/>
    <xs:enumeration value="false"/>
    <xs:enumeration value="request"/>
    <xs:enumeration value="question"/>
    <xs:enumeration value="confirmation"/>
    <xs:enumeration value="promise"/>
    <xs:enumeration value="declination"/>
    <xs:enumeration value="statement"/>
  </xs:restriction>
</xs:simpleType>
```
Appendix C. Gellish XML Schema

```xml
<x:simpleType name="stringCommonality">
  <xs:restriction base="xs:token">
    <xs:enumeration value="case sensitive identical"/>
    <xs:enumeration value="case insensitive identical"/>
    <xs:enumeration value="case sensitive partially identical"/>
    <xs:enumeration value="case insensitive partially identical"/>
    <xs:enumeration value="case sensitive front end identical"/>
    <xs:enumeration value="case insensitive front end identical"/>
    <xs:enumeration value="case sensitive different"/>
    <xs:enumeration value="case insensitive different"/>
    <xs:enumeration value="equal"/>
    <xs:enumeration value="unequal"/>
    <xs:enumeration value="less than or equal"/>
    <xs:enumeration value="greater than or equal"/>
  </xs:restriction>
</xs:simpleType>

<!-- Derived types from entity -->
<x:complexType name="objectName">
  <xs:simpleContent>
    <xs:extension base="xs:token">
      <xs:attribute name="StringCommonality" type="gellish:stringCommonality" use="optional"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>

<x:complexType name="leftObject">
  <xs:all>
    <xs:element name="Name" type="gellish:objectName" minOccurs="0" maxOccurs="1"/>
    <xs:element name="NameContext" type="gellish:context" minOccurs="0" maxOccurs="1"/>
    <xs:element name="Role" type="gellish:role" minOccurs="0" maxOccurs="1"/>
    <xs:element name="Cardinalities" type="gellish:cardinalities" minOccurs="0" maxOccurs="1"/>
    <xs:element name="TypeName" type="xs:token" minOccurs="0" maxOccurs="1"/>
  </xs:all>
  <xs:attribute name="Uid" type="gellish:uid" use="required"/>
  <xs:attribute name="Reality" type="gellish:reality" use="optional"/>
</xs:complexType>

<x:complexType name="rightObject">
  <xs:all>
    <xs:element name="Name" type="gellish:objectName" minOccurs="0" maxOccurs="1"/>
    <xs:element name="Role" type="gellish:role" minOccurs="0" maxOccurs="1"/>
    <xs:element name="Cardinalities" type="gellish:cardinalities" minOccurs="0" maxOccurs="1"/>
  </xs:all>
  <xs:attribute name="Uid" type="gellish:uid" use="required"/>
</xs:complexType>

<x:complexType name="relationType">
  <xs:simpleContent>
    <xs:extension base="xs:token">
      <xs:attribute name="Uid" type="gellish:uid" use="required"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>

<x:complexType name="language">
  <xs:simpleContent>
    <xs:extension base="xs:token">
      <xs:attribute name="Uid" type="gellish:uid" use="optional"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

86
Appendix C. Gellish XML Schema

```xml
<xs:complexType name="context">
  <xs:simpleContent>
    <xs:extension base="xs:token">
      <xs:attribute name="Uid" type="gellish:uid" use="optional" />
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>

<xs:complexType name="role">
  <xs:simpleContent>
    <xs:extension base="xs:token">
      <xs:attribute name="Uid" type="gellish:uid" use="optional" />
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>

<!−− Actual Table −−>
<xs:element name="Table">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Fact" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:all>
            <xs:element name="LeftObject" type="gellish:leftObject" minOccurs="1" maxOccurs="1" />
            <xs:element name="RelationType" type="gellish:relationType" minOccurs="1" maxOccurs="1" />
            <xs:element name="RightObject" type="gellish:rightObject" minOccurs="0" maxOccurs="1" />
            <xs:element name="Language" type="gellish:language" minOccurs="0" maxOccurs="1" />
            <xs:element name="ValidityContext" type="gellish:context" minOccurs="0" maxOccurs="1" />
            <xs:element name="Author" type="xs:token" minOccurs="0" maxOccurs="1" />
            <xs:element name="Reference" type="xs:token" minOccurs="0" maxOccurs="1" />
            <xs:element name="Description" type="xs:string" />
          </xs:all>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

87
Appendix C. Gellish XML Schema

```xml
minOccurs="0" maxOccurs="1" />
</xs:element name="PartialDefinition" type="xs:string"
minOccurs="0" maxOccurs="1" />
</xs:element name="FullDefinition" type="xs:string"
minOccurs="0" maxOccurs="1" />
</xs:element name="Remarks" type="xs:string"
minOccurs="0" maxOccurs="1" />
</xs:element name="UoM" type="gellish:uom"
minOccurs="0" maxOccurs="1" />
</xs:element name="Collection" type="gellish:collection"
minOccurs="0" maxOccurs="1" />
</xs:element name="Picklist" type="gellish:picklist"
minOccurs="0" maxOccurs="1" />
</xs:element name="MappingAccuracy" type="gellish:mappingAccuracy"
minOccurs="0" maxOccurs="1" />
</xs:element name="Picklist" type="gellish:picklist"
minOccurs="0" maxOccurs="1" />
</xs:element name="MappingAccuracy" type="gellish:mappingAccuracy"
minOccurs="0" maxOccurs="1" />
</xs:element>
</xs:schema>
```
Appendix D

Sample Gellish XML Document

The following example shows what a valid Gellish XML file (or GML file) should look like. The table represented in this file contains four Gellish facts. Since this file is a valid GML file, it corresponds to the Gellish XML schema, discussed in section 4.3, and presented in appendix C.

```xml
<?xml version="1.0" encoding="utf-8"?>
<gellish:Table Name="ASampleTable"
xmlns:gellish="http://gellikx.com/2008/ns/1.0/GellishSchema">
<!−− Minimal required sample fact −−>
<Fact Uid="1234" LatestUpdate="2008-08-28T11:23:00.12354"
EffectiveFrom="2008-08-28T11:23:00.12354"
Status="proposed" Intention="true">
<LeftObject Uid="2345"/>
<RelationType Uid="1146"/>
<RightObject Uid="3456"/>
</Fact>

<!−− Completely filled sample fact −−>
<Fact Uid="3456" EffectiveFrom="2008-08-28T11:25:00"
LatestUpdate="2008-08-28T11:26:12" Sequence="1" Status="proposed"
Intention="true" LineUid="12233" SuccessorUid="13345">
<LeftObject Uid="45673" Reality="materialized">
<Name StringCommonality="case sensitive identical">
reciprocating pump
</Name>
</LeftObject>
<RelationType Uid="1146">
is a specialization of a
</RelationType>
<RightObject Uid="65432">
<Role Uid="99873" supertype="Role">
<Name StringCommonality="less than or equal">
pump
</Name>
</Role>
<Cardinalities>1,n</Cardinalities>
</RightObject>
</Fact>
</gellish:Table>
```

89
Appendix D. Sample Gellish XML Document

45  <Remarks>
46    <! [CDATA[additional remarks about this fact]]>
47  </Remarks>
48  </UoM Uid="44332">km/h</UoM>
49  </Fact>
50
51  <!-- Actual fact taken from a Sourceforge Gellish table (Part4 Rotating e-Machines) -->
52  <Fact Intention="true" Sequence="4482" Uid="113020">
53    <Language Uid="910036">English</Language>
54    <LeftObject Uid="131062">
55      <Name>vertical turbine discharge barrel pump</Name>
56    </LeftObject>
57    <RelationType Uid="1146">is a specialization of</RelationType>
58    <RightObject Uid="130058">
59      <Name>centrifugal pump</Name>
60    </RightObject>
61    <Author>Jaap Rozeboom</Author>
62    <Reference>Jaap Rozeboom</Reference>
63    <PartialDefinition>
64      <! [CDATA[that is provided with a discharge barrel [casing] below grade and a nozzle head
65        with suction- and discharge flange above grade, impeller inlet below grade. The nozzle head
66        serves as motor stool. Pump is a dry pit pump.]]>
67    </PartialDefinition>
68  </Fact>
69
70  <!-- Actual fact taken from the Gellish TOPini -->
71  <Fact Uid="1001143" Status="accepted" Intention="true">
72    <Language Uid="910036">English</Language>
73    <LeftObject Uid="1147">
74      <Name>exception of an aspect from a composition</Name>
75      <NameContext Uid="193259">ontology</NameContext>
76    </LeftObject>
77    <RelationType Uid="1146">is a specialization of</RelationType>
78    <RightObject Uid="1334">
79      <Name>association</Name>
80      <Language Uid="910036">English</Language>
81      <ValidityContext Uid="492014">Gellish</ValidityContext>
82      <Reference>AP221 modelling team</Reference>
83      <PartialDefinition>
84        <! [CDATA[that indicates the excepted aspect that is an element of the part collection but
85          is excepted from being an element of the composed whole aspect.]]>
86    </PartialDefinition>
87  </Fact>
88
89  <Collection>facts about rotating equipment</Collection>
90  <Fact>
91
92  <! [CDATA[that is provided with a discharge barrel [casing] below grade and a nozzle head
93    with suction- and discharge flange above grade, impeller inlet below grade. The nozzle head
94    serves as motor stool. Pump is a dry pit pump.]]>
95
96  <FullDefinition>
97    <! [CDATA[is a centrifugal pump that is provided with a discharge barrel [casing] below
98      grade and a nozzle head with suction- and discharge flange above grade, impeller inlet below
99      grade. The nozzle head serves as motor stool. Pump is a dry pit pump.]]>
100  </FullDefinition>
101  </Fact>
102
103  <Collection>facts about upper ontological relations between individual things</Collection>
104  </gellish:Table>
Appendix E

Installatie en Basis Handleiding

The following manual was written in Dutch for users of the DOL prototype application(s). The first section covers the requirements needed to and process to install the various applications. Following these installation instructions a quick-start user manual will be given for each of the prototype/demo applications.

E.1 Installatie DOL Prototype

Deze handleiding beschrijft o.a. het proces waarmee de verschillende componenten van het prototype Distributed Object Library (afgekort DOL) kunnen worden geïnstalleerd. Voor informatie over het gebruik van de verschillende componenten en/of applicaties zie de quick start handleidingen verderop in dit document.

Alle services en applicaties hier beschrijven kunnen zowel worden geïnstalleerd op Windows XP desktop PC’s als ook server waarop Windows 2003 Server draait.

E.1.1 Microsoft .NET Framework Vereist

Voor het kunnen installeren en gebruiken van deze applicaties is de Microsoft .NET Framework nodig. Het gaat hierbij om de versies 2.0, 3.0 en 3.5. Alle drie deze gratis versies moeten geïnstalleerd zijn op uw computer, voor het goed functioneren van de verschillende componenten. Bezoek voor het downloaden de .NET Framework Developer Center van Microsoft1.

E.1.2 DOL Engine Service

Het kloppend hart van het gedistribueerde netwerk is de DOL Engine (service). Dit component wordt gebruikt om Gellish bronnen aan te bieden middels publicaties, het aanvragen van abonnementen op dergelijke publicaties en het uitzetten en beantwoorden van (gedistribueerde) Gellish query.

De DOL Engine Service is eenvoudig te installeren via de meegeleverde setup. Eenmaal gestart krijgt u een welkomsscherm zoals in figuur E.1. Door twee keer op 'Next' te klikken belandt u op het scherm zoals afgebeeld in E.2. Geef hier op waar u de DOL Engine Service wil hebben geïnstalleerd (standaard 'C:\Program Files\Gellix\DOL Engine Service') en of de service onder alle gebruikersaccount van uw Windows installatie wordt geïnstalleerd of

alleen uw huidige account. Klik nogmaals op 'Next' om bij het scherm te komen waarop u de port kan opgeven waarop de Engine service zal gaan luisteren.

Door gebruik te maken van het scherm zoals hieronder is afgebeeld, kunt u de port wijzigen waarop de service standaard begint met luisteren (standaard is dit port 7464). Wanneer u zich achter een firewall bevindt, en u wilt verbindingen van buitenaf accepteren, dient u uw port-forwarding instellingen op deze firewall aan te passen (zie paragraaf E.1.7).

Door nu nogmaals op de 'Next' knop te klikken start de installatie van de service. Eenmaal afgerond zult u een scherm krijgen met de boodschap dat de installatie succesvol is afgerond. U sluit de installatie wizard door op de 'Close' knop te klikken.

**E.1.3 DOL Peer Tracker Service (optioneel)**

De DOL Peer Tracker Service is niet noodzakelijk voor het kunnen functioneren van een DOL Engine. Het doel van de peer tracker is om een lijst bij te houden van aangemelde engines, met daarbij het adres waarop deze engines te bereiken zijn. Als test is een DOL Peer Tracker geïnstalleerd op een het volgende adres:


Net als de DOL Engine Service is de Peer Tracker Service eenvoudig te installeren via de setup, welke op de CD te vinden is. De stappen zijn zo goed als hetzelfde, zoals het aangeven van het installatie pad, gebruikersaccounts waaronder de service wordt geïnstalleerd, en de standaard port waarop de service luistert naar verbindingen (in dit geval standaard 6120).

**E.1.4 Activeren van een Windows Service**

Eenmaal geïnstalleerd moeten de DOL Engine- en DOL Peer Tracker Services gestart worden. Het is mogelijk om dit via de DOL Manager applicatie te doen, echter zal deze paragraaf
E.1. Installatie DOL Prototype

Figure E.2: Stap in DOL Engine setup waarmee de lokatie op uw computer kan worden opgegeven en de gebruiker accounts voor wie de installatie van toepassing is.

Figure E.3: DOL Engine setup stap voor het opgeven van de port waarop de service zal luisteren.
Allereerst opent u het Services venster via 'Start \ Control Panel \ Administrative Tools \ Services' (in het Nederlands is dit 'Start \ Configuratiescherm \ Systeembeheer \ Services'). Een andere optie om bij Services beheer te komen is door eerst 'Start \ Run...' (Nederlands: 'Start \ Uitvoeren...') te selecteren en vervolgens het commando 'service.msc' uit te voeren. Een scherm zoals figuur E.4, toont vervolgens een overzicht van alle op uw computer geïnstalleerde services.

Zoek in deze lijst met services naar de 'DOL Engine Service'. Als het goed is staat er achter de beschrijving van deze service nog geen status. Door links van de lijst van service op de 'Start' te klikken start de DOL Engine Service. Doordat deze service bij het opstarten een port opent om requests op te vangen, kan het zijn dat eventuele beveiligings programma’s, zoals Norton etc., om bevestiging van deze actie vragen. Het stoppen van de service kan door op de 'Stop' link te klikken.

De werkwijze voor het starten (en stoppen) van de DOL Engine Service is gelijk aan die van de DOL Peer Tracker Service.

E.1.5 DOL Manager (optioneel)

De DOL Manager applicatie biedt u de mogelijkheid om uw lokale DOL Engine en/of Peer Tracker te beheren. Hierbij moet u denken aan het starten en stoppen van de services (zoals hierboven beschreven), het toevoegen, verwijderen en bekijken van Gellish datastores, het beheren van eventuele publicaties en abonnementen (subscriptions), en het uitvoeren van (gedistribueerde) Gellish queries. Details over deze acties zijn te vinden verderop in dit
E.1. Installatie DOL Prototype

Documenten.
Installatie van de DOL Manager is een kwestie van starten van de setup en aangeven op welke locatie het programma moet worden geïnstalleerd (standaard C:\Program Files\Gellikx\DOL Manager). Deze procedure is identiek aan de procedure voor het installeren van de DOL Engine Service.

Eenmaal geïnstalleerd vindt u onder 'All Programs → Gellikx → DOL Manager' een snelkoppeling naar de DOL Manager applicatie.

E.1.6 Demo Applicaties

Op de CD met programma's zijn ook een tweetal applicaties te vinden die puur en alleen voor demo doeleinden zijn: de DOL Environment Setup en Demo Definition Viewer applicaties. Deze eerste applicatie kan gebruikt worden om meer dan één lokale DOL Engine te managen. Naast de standaard geïnstalleerde DOL Engine Service kunnen via deze applicatie lokaal meerdere engines gestart worden (uiteraard om een verschillend adres).

De tweede applicatie kan gebruikt worden om via de lokaal geïnstalleerde DOL Engine gedistribueerde queries uit te zetten, waarbij gezocht wordt om de naam van een object (zie de handleiding aan het einde van dit document voor meer informatie over het gebruik). Beide applicaties zijn te installeren via de meegeleverde setups.

E.1.7 Port-forwarding Instellen

Om ook andere engines toegang te geven tot uw lokaal geïnstalleerde DOL Engine (of tot een lokale Peer Tracker) dient u, indien dit van toepassing is, uw firewall aan te passen. Deze bevindt zich tussen uw computer en het internet, of wordt door Windows afgehandeld. Raadpleeg voor het instellen van deze firewalls de daarvoor behorende handleiding.

E.1.8 Deïnstallatie Componenten

Aangezien de installatie niet een standaard deïnstallatie programma meelevert, zal hier kort uitgelegd worden hoe de services en/of applicaties eenvoudig kunnen worden verwijderd van uw computer.

Optie 1: Setup gebruiken

Indien u nog beschikt over de originele setup, kunt u deze nogmaals uitvoeren. Zodra de setup wordt opgestart op een computer waar de service of applicatie al geïnstalleerd is, krijgt u de vraag of u een 'Repair' (herstel) of 'Remove' (verwijder) actie wilt uitvoeren. Wanneer u deze laatste optie selecteert en op de 'Finish' knop drukt, zal de desbetreffende applicatie worden gedeïnstalleerd.

Optie 2: Via Windows Add/Remove Programs

Wanneer u niet meer beschikt over de originele setup bestanden, kunt u altijd via de Windows interface voor het toevoegen en verwijderen van applicaties deze componenten alsnog verwijderen. In figuur E.6 ziet u het scherm wat bij deze Windows functionaliteit hoort.

Nadat de lijst met geïnstalleerde programma's is opgebouwd kunt u naar beneden scrollen op zoek naar de desbetreffende applicaties (zoals bijvoorbeeld de DOL Engine Service). Eenmaal gevonden, selecteert u de applicatie en klikt op de 'Remove' knop. Wanneer het deïnstallatie process is afgerond kunt u de Add or Remove interface van Windows sluiten.
Appendix E. Installatie en Basis Handleiding

Figure E.5: Setup gebruiken voor het deinstalleren van DOL services.

Figure E.6: Windows venster voor het toevoegen of verwijderen van applicaties.
Verwijderen van Data Map

Om te kunnen voorzien in eventuele updates wordt bij het deinstalleren de map met daarin applicatie specifieke data niet verwijderd. Het gaat hier om map 'Data', welke te vinden is in de mappen voor de DOL Engine Service en DOL Peer Tracker Service (standaard te vinden in de map ‘C:\Program Files\Gellikx\’). Wanneer u deze map niet zou weggooien en later nogmaals de service installeert, dan zullen alle instellingen en data van de service automatisch worden hersteld uit de vorige installatie.

E.2 DOL Manager Quick Start Guide

E.2.1 Schermopbouw

Wanneer u de DOL Manager opstart zult u een scherm zien zoals hieronder is afgebeeld. Dit hoofdvenster kent een aantal verschillende onderdelen, welke elk kort hieronder zullen worden toegelicht.

De **Navigatie-Boom** bevindt zich aan de linkerkant van het venster (blauw omkaderd in figuur E.7) en is vormgegeven als een boom, welke als wortel element de DOL Configuration Manager kent. Afhankelijk van de geïnstalleerde service zijn onder de DOL Configuration Manager node een tweetal nodes te vinden: één voor het beheren van de lokale Peer Tracker, en één voor het beheren van de lokale Engine.

Via deze node kunt u respectievelijk de service starten en stoppen door een rechter muisklik op de desbetreffende node te geven en de correcte menu optie te kiezen.

Het **Hoofd-Venster** bevindt zich rechts van de navigatie-boom (rood omkaderd in figuur) en kan een aantal tabbladen met verschillende Gellish fact viewers bevatten.

De **Toolbar** bevindt zich boven de navigatie-boom (groen omkaderd figuur) en bevat twee functionele knoppen (de overige knoppen zijn op dit moment slechts dummy knoppen en hebben verder geen functie):
Appendix E. Installatie en Basis Handleiding

E.2.2 Peer Tracker Beheren

Via de DOL Manager kunnen de volgende acties op de lokale Peer Tracker Service worden uitgevoerd:

- Starten en stoppen van Peer Tracker diensten
- Tracker hernoemen
- Bekijken en verwijderen van aangemelde engines

De Peer Tracker node bevat tevens de huidige naam en status van de Peer Tracker service. De naam die tussen de haakjes te vinden is, is standaard gelijk aan de unieke sleutel van de Peer Tracker en ziet eruit als een reeks van 32 cijfers en letters. Alle bovenstaande acties zijn bereikbaar via het snelmenu dat zichtbaar wordt als u een rechtermuisklik geeft op de Peer Tracker node.

**Starten en Stoppen Service**

Het is mogelijk om via de DOL Manager de DOL Peer Tracker Service, welke lokaal geïnstalleerd is, te starten en te stoppen. Wanneer een service gestopt is kunnen de diensten die de service biedt niet meer worden aangeroepen, dit geldt voor zoowel lokale- als externe aanroepen.

Wanneer u een rechtermuisklik geeft op de Peer Tracker node, zult u een snelmenu zien zoals weergegeven in figuur E.8. Afhankelijk van de huidige status van de service kunt u ervoor kiezen om de service te starten of te stoppen, met resp. de 'Start' en 'Stop' opties.

**Tracker Hernoemen**

Het Peer Tracker menu bevat, naast de 'Start' en 'Stop' opties ook een 'Rename Tracker' optie. Hiermee kunt u de naam aanpassen, welke zichtbaar is in uw DOL Manager als ook te zien is als engines zich bij uw tracker aanmelden.
E.2. DOL Manager Quick Start Guide

Nadat u op de 'Rename Tracker' optie geklikt heeft, krijgt u een venster zoals weergeven in figuur E.9. De nieuwe naam voor de Peer Tracker Service kunt u opgeven in het tekstveld, waarna u deze wijziging via de 'OK' knop kunt bevestigen.

Aangemelde Engines

Onder de Peer Tracker node vindt u een map genaamd 'Connected Engines'. Hierin komen alle engines te staan die zich bij uw lokale peer tracker hebben aangemeld. Om deze lijst met engines te verversen kunt u gebruik maken van de 'Refresh' optie die te vinden in het menu onder de rechter muisknop.

Tevens is het mogelijk om reeds aangemelde engines te verwijderen uit deze lijst, door de 'Delete' optie te kiezen in het snelmenu, wat zichtbaar wordt na het geven van een rechter muisklik op de desbetreffende engine.

E.2.3 Engine Beheren

Onder het beheren van de lokale DOL Engine Service vallen de volgende taken:

- Starten en stoppen van de Engine diensten
- Engine hernoemen
- Toevoegen en verwijderen van peer trackers
- Toevoegen en verwijderen van andere engines
- Toevoegen, bekijken en verwijderen van Gellish datastores
- Aanmaken, bekijken, beheren en verwijderen van publicaties en geabonneerde engines
- Aanvragen en opheffen van abonnementen naar publicaties van andere engines
Appendix E. Installatie en Basis Handleiding

De Engine Service node bevat vijf mappen met daarin de peer trackers, andere engine, datatstores, publicaties en aangevraagde subscriptions (ofwel abonnementen op publicaties van andere engines).

Starten en Stoppen Service

De lokale Engine Service kan worden gestart en gestopt op dezelfde wijze als de Peer Tracker Service. Door een rechter muisklik te geven op de Engine node opent zich eenzelfde menu zoals in het geval van de Peer Tracker. Afhankelijk van de status van de engine kunt u via dit menu de Engine Service starten en stoppen.

Engine Hernoemen

Door de optie 'Rename Engine' te gebruiken is het mogelijk om de naam van de lokale Engine Service te wijzigen. Deze naam is zichtbaar voor andere engine die zoeken naar onbekende engines via een peer tracker.

Peer Trackers

Zonder een peer tracker kan het voor een engine erg alleen zijn. De peer tracker kan gezien worden als een telefoonboek met daarin de adressen van andere engines. Elke engine kan meerdere tacker bevatten, elk van deze trackers wordt vervolgens gebruikt bij het zoeken naar nieuwe engines.

Via het snelmenu, dat zich opent na het geven van een rechter muisklik op het mapje 'Trackers', is het mogelijk om middels de optie 'New Tracker' een tracker aan uw engine toe te voegen. Om een tracker toe te voegen maakt u gebruik van het venster zoals te zien in figuur E.10.

Allereerst vindt u in dit venster een tekstveld bedoeld voor het adres, of ookwel URI genoemd. Het formaat van deze URI’s is als volgt: 'protocol://host:port/path'. Het protocol zal in alle gevallen 'http' zijn, en geeft aan wat de basis vorm van communicatie zal zijn. De host is gelijk aan het adres waarop de server te vinden is. Dit adres kan een domeinnaam zijn, zoals google.com, of een IP adres. Een domeinnaam die altijd naar uw lokale computer verwijst is bijvoorbeeld 'localhost' (het IP adres voor deze naam is gelijk aan '127.0.0.1'). De port kan gezien worden als de brievenbus waar een brief bezorgd moet worden, of in ons geval een aanvraag om opgenomen worden in de trackers' telefoonboek.
De waarde van een port zijn we ook eerder al tegengekomen bij de installatie van zowel de Peer Tracker- als Engine Services (standaard bevindt een tracker zich op port '6120').

Het path is een extra verwijzing die leeg kan blijven als dat gewenst is; echter wordt standaard een Peer Tracker Service gestart binnen het path '/Dol/PeerTrackerService'. Als u bijvoorbeeld een tracker wilt aanmelden die u lokaal geïnstalleerd heeft, dan zou u gebruik kunnen maken van het volgende volledige adres: 'http://localhost:6120/Dol/PeerTrackerService'.

Onder het tekstveld voor het adres (of URI) vindt u een 'Test' knop, hiermee wordt gekeken of er inderdaad een peer tracker draait op het opgegeven adres. Mocht u verbinding maken met een lokale peer tracker let er dan wel op dat deze Peer Tracker Service reeds gestart is.

Middels het vinkje 'Connect to Tracker upon completion', kunt u na het sluiten van het venster automatisch uw lokale engine registreren bij de tracker, zodat uw engine door andere engines wordt gevonden als deze een zoekopdracht uitvoeren bij de desbetreffende tracker.

Nadat een tracker is toegevoegd aan de lijst van trackers zult u een tracker icoontje voor deze nieuwe tracker aantreffen in de map 'Trackers' van de Engine Service node. Dit icoontje kent twee versies waarmee de status van de tracker wordt aangegeven (zie tabel hieronder).

Ook is het mogelijk om via het snelmenu, wat zich opent na het geven van een rechter muisklik op een tracker, uw registratie bij deze tracker ongedaan maken ('Disconnect') of juist een registratie door te voeren ('Connect'). Onder deze opties ziet u ook de 'Delete' optie, waarmee de tracker wordt verwijderd uit de lijst van trackers die uw engine kent.

Als laatste bevat het snelmenu, waaraan hierboven gerefereerd wordt, ook een 'Properties' optie. Hiermee kunt u meer details bekijken van de tracker die u op dat moment geselecteerd heeft.

Engines

Omdat in dit prototype geen automatische ontdekking van andere engines is ingebouwd dient u zelf te zoeken naar andere engines, welke u wilt gebruiken in het uitzetten van een gedistribueerde query. Zowel bij het uitzetten van een gedistribueerde query als bij het zoeken naar publicaties wordt gebruik gemaakt van deze lijst van engines.

Om engines toe te voegen aan de lijst, geeft u een rechter muisklik op het map icoontje genaamd 'Engines'. In het hierop volgende snelmenu selecteert u de optie 'Discover New'. In het bovenste gedeelte van het venster (figuur E.11), ziet u alle trackers welke u reeds heeft toegevoegd. Door een vinkje voor een tracker te zetten, neemt u de tracker mee in het ontdekkingsproces. Wanneer u klaar bent om alle aangevinkte trackers te bevragen klikt u op de 'Start Discovery' knop.

Tijdens het ontdekkingsproces bevraagt uw engine de opgegeven trackers voor engines welke nog niet bekend zijn bij uw lokale engine. Het resultaat van dit proces wordt in het onderste gedeelte van het scherm weergegeven. Als het proces is afgerond zult u een boodschap hiervan krijgen.

Net als bij het instellen van de tracker hierboven, kunt u een vinkje inschakelen voor de nieuw ontdekte engines. Elk aangevinkte engine zal, na het klikken op de 'Add' knop, toegevoegd worden aan de lijst van engines van uw lokale engine. Middels de 'Close’ knop kunt u vervolgens het Discovery venster sluiten.

U kunt ten alle tijden de lijst van engines verwerven door wederom het snelmenu te openen, door een rechter muisklik te geven op het 'Engines' mapje, en de optie 'Refresh' te kiezen. Ook is het mogelijk om via hetzelfde menu engine te verwijderen, door de 'Delete' optie, of om nadere info over de engine op te vragen, kies hiervoor de 'Properties' optie.
Figure E.11: Venster voor het ontdekken van nieuwe engines.
Datastores

Zonder Gellish data is er weinig te bequeryen, deze paragraaf zal dus nader in gaan op het toevoegen, verwijderen en bekijken van Gellish datastores. Deze datastore kunnen in XML formaat of vanaf een SQL Server database worden aangeroepen.


Kies vervolgens het type datastores, ofwel een XML bestand, ofwel een SQL database. In het geval van een XML bestand, moet de locatie van het bestand worden opgegeven in het tekstveld onderaan het venster (Location). Eventueel kan middels de blader (ellipsis) knop gezocht worden op uw computer.

Wanneer u ervoor kiest om gebruik te maken van een SQL database zult u i.p.v. de bestandslocatie de connectie gegevens van de SQL database moeten opgeven. Naast de username en password (resp. gebruikersnaam en wachtwoord) zult u ook de server, database en command text moeten opgeven. Neem contact op met uw systeem beheerder om deze waarden te achterhalen. Nadat u alle gegevens heeft ingevuld, voegt u de datastore toe door op de 'OK' knop te klikken.

Via de 'Delete' optie in het Datastores' snelmenu kunt ook uit datastores verwijderen van uw lokale engine. LET OP! Door het verwijderen van de datastore, zal ook elke publicatie die gekoppeld is aan deze datastore worden verwijderd!

Ook is het mogelijk om de feiten in datastores te bekijken door in het snelmenu de 'View Facts' optie te kiezen. De beschrijving van de Gellish fact viewer wordt verderop in dit document behandeld.

Publicaties

Publicaties zijn nodig om de feiten, welke opgeslagen zijn in één of meer van uw datastores, aan te bieden aan andere engines. Voordat een publicatie kan worden aangemaakt, zullen er één of meer datastores moeten zijn toegevoegd aan uw engine.
Appendix E. Installatie en Basis Handleiding

Figure E.13: Nieuwe publicatie toevoegen.

Om een publicatie toe te voegen aan uw engine selecteert u de 'New Publication' vanuit het snelmenu. Op het venster, gelijk aan figuur E.13, geeft u de mogelijkheid om naast de naam (Name) van de publicatie ook een beschrijving (Description) toe te voegen. Deze beschrijving is zichtbaar voor andere gebruikers/engines wanneer zij een subscription (abonnement) willen aanvragen voor uw publicatie. Naast de naam en beschrijving dient u ook aan te geven welke datastore via uw publicatie wordt aangeboden.

Als laatste kunt u kiezen of de inhoud van de publicatie voor iedereen toegankelijk is ('Public') of dat gebruikers pas, na het door u accederen van het aangevraagde abonnement, toegang hebben tot de inhoud van de publicatie ('Restricted'). Eenmaal alle velden correct ingevuld, kunt u door op de 'OK' knop te klikken de nieuwe publicatie aanmaken en toevoegen aan uw lijst van publicaties.

Via het snelmenu onder publicaties kunt u naast het verversen van de lijst en het bekijken van de eigenschappen ('Properties' opties), ook de feiten uit de publicatie bekijken in de Gellish fact viewer (zie verderop dit document).

Ook ziet u, indien engines een subscription hebben aangevraagd, een plus-tekentje voor een publicatie staan. In dit geval kunt u, door op dit plusje te klikken, het overzicht van deze abonnees openen. Door gebruik te maken van de 'Approve' en 'Cancel' opties in en snelmenu van deze abonnees, kunt u resp. de abonnee toegang geven tot de publicatie of juist de toegang ontzeggen tot de publicatie.

Subscriptions

De laatste map in de navigatie boom ('Subscriptions'), geeft een lijst weer van subscriptions (abonnementen), welke u heeft aangevraagd bij andere engines.

Om een nieuwe subscription aan te vragen bij een andere engine, kiest u de 'Request Subscription' optie uit het subscriptions snelmenu. Het venster wat vervolgens wordt geopend (figuur E.14) geeft een lijst weer van alle publicaties die worden aangeboden door de engines.
E.2. DOL Manager Quick Start Guide

Figure E.14: Venster voor het aanvragen van abonnementen op publicaties.

in uw lokale lijst met engines. In deze lijst ziet u naast de naam van de publicatie, en het toegankelijkheids type, ook de beschrijving van de publicatie.

Door voor een publicatie een vinkje te zetten en op de 'Request' knop te klikken, zal de subscription voor de publication worden toegevoegd aan uw lijst van subscriptions. Tevens wordt de request verzonden naar de eigenaar van de publicatie, welke vervolgens kan kiezen om u toegang te verlenen tot de publicatie of niet.

Op dit moment is het aanvragen van een subscription voor publiekelijke publicaties niet nodig. Immers worden bij het ontvangen van een Gellish query altijd alle publiekelijke publicaties meegenomen in het zoeken naar een antwoord. Echter zullen alleen publicaties met beperkte toegankelijkheid worden meegnomen, als de maker van de query ook een (goedgekeurd) abonnement heeft aangevraagd voor de betreffende publicatie.

E.2.4 Gellish Feiten Bekijken

De voorgaande pagrafen hebben beschreven hoe u de verschillende elementen van uw peer tracker en engine kunt instellen en beheren. Hieronder zal kort worden beschreven hoe de Gellish fact viewer kan worden gebruikt.

Middels de Gellish fact viewer kunt u een overzicht krijgen van de Gellish feiten welke zijn opgeslagen in uw lokale datastore(s) en publicatie(s). Tevens is het mogelijk om via deze interface Gellish queries te bouwen (zie verderop in dit document) waarvan het antwoord op dezelfde wijze kan worden bekeken. Het hoofd venster van de DOL Manager applicatie scherm ziet er ongeveer hetzelfde uit als figuur E.17.

Het grootste gedeelte van de venster bevat een tabel met daarin de feiten uit de betreffende datastore of publicatie. Als u klikt op een cell in deze tabel wordt automatisch de rij geselecteerd waarin de cell zich bevindt. Deze rij vertegenwoordigt één enkel Gellish feit. Bij het selecteren van een feit wordt automatisch de inhoud van het eigenschappen venster
Appendix E. Installatie en Basis Handleiding

Figure E.15: Sorteer mogelijkheden Gellish fact viewer.

(rechts van de tabel) aangepast, zodat de eigenschappen van het feit beter te bekijken zijn. Het onderste gedeelte van het venster bevat de Gellish query builder. Deze query builder zal in het volgende gedeelte van dit document besproken worden.

Het is mogelijk één of meer van de drie gedeelten van de Gellish fact viewer te verbergen. Hiervoor kunt u onder andere gebruik maken van de onderstaande knoppen. Deze knoppen zijn te vinden boven de tabel met Gellish feiten.

Naast de knoppen voor het verbergen van de verschillende gedeelten, zijn de eerste twee knoppen in de toolbar bedoeld voor het terugspringen naar de vorige set aan Gellish feiten en het verwijderen van de huidige set van Gellish feiten (nogmaals: met verwijderen wordt slechts de weergave van feiten bedoeld, de inhoud van uw datastores kan niet middels de DOL Manager worden verwijderd).

Zoals beschreven in het System en Design Document kent Gellish een aantal subsets van Gellish kolommen, nl. atomic, taxonomy, business model, etc.. De Subset Selector, zoals beschreven in bovenstaande tabel, stelt u in staat om de kolommen die worden weergeven te wijzigen. Ook is het mogelijk om de sortering van de feiten o.b.v. deze kolommen aan te passen met de Sort knop. Na het klikken op deze knop ziet u het sorteer venster, zoals weergegeven in figuur E.15, via dit venster kunt u tot vier Gellish kolommen op- of aflopend sorteren.

Tevens is het mogelijk om direct op één kolom sorteren, door op de naam van de kolom te klikken. Door nogmaals op de kolom header te klikken kunt u de sortering veranderen van aflopend naar oplopend en vice versa.

E.2.5 Gellish Queries Uitvoeren

De Gellish fact viewer is niet uitsluitend voor het bekijken van de Gellish feiten. Zoals hierboven beschreven bevat het scherm van de Gellish fact viewer naast een tabel- en eingeschappe gedeelte ook een basis query builder.

Voordat u een query kunt bouwen moeten eerst één of meer velden, ofwel Gellish kolommen, worden toegevoegd. Dit kunt u doen door een enkel veld te selecteren uit de lijst met
Figure E.16: Selectie menu met daarin de Gellish subset varianten die worden ondersteund in de DOL Manager fact viewer.

Figure E.17: Gellish fact viewer venster.
Fields en dan op de Add Selected Field(s) knop te klikken. Ook kunt u een aantal velden in één keer toevoegen door op het pijltje rechts van de Add Selected Field(s) te klikken, waardoor een tweetal opties gekozen kunnen worden. Bijvoorbeeld de 'Basic Fields' optie voegt resp. de Gellish velden intention, left object UID, left object name, relation type UID, relation type name, right object UID en right object name toe aan uw query. Intention moet altijd gevuld zijn met de waarde 'question' als het gaat om een query. Vervolgens kunt een Gellish query bouwen conform de Gellish Query definities. Middels de Targets knop kunt u opgeven welke Gellish bronnen worden bequeried. Momenteel kan alleen worden aangegeven of de query moet worden gedistribueerd (Distributed) moet worden uitgevoerd of alleen op de Gellish tabel die op dat moment zichtbaar is in de Gellish fact viewer (Current View). Deze selectie kunt u opgeven door het vinkje voor de betreffende optie aan of uit te zetten.

Om nu de query te starten klikt op op de Execute Query knop. Het resultaat van deze query wordt getoond in het tabel gedeelte van de Gellish fact viewer.

E.3 Demo Environment Setup

Deze applicatie is bedoeld om naast de lokale engine meer engines te draaien. Indien u een engine lokaal heeft geïnstalleerd zal de applicatie deze bovenin de DOL Environment boom plaatsen.

Daarnaast kunt u door op New Engine knop een nieuwe engine lokaal starten. Deze engine is dan actief, zolang u de applicatie heeft draaien. Na het afsluiten van de applicatie zullen alle engine worden gestopt, m.u.v. de lokaal geïnstalleerde engine.

Na het klikken op de New Engine knop, opent zich het Create Engine venster (figuur E.19). In dit venster kunt u naast een naam voor de nieuwe engine ook het adres opgeven van deze engine, zie de installatie handleiding aan het begin van dit document voor een uitleg over de verschillende onderdelen van het adres. Let wel op dat het adres afwijkt van uw lokaal geïnstalleerde engine; elk uniek adres kan namelijk slechts behoren tot één engine.
Het onderste gedeelte van dit venster is bedoeld voor het Data bestand van de engine. In dit bestand slaat de engine naast het adres en de naam bijvoorbeeld op welke tracker de engine gebruikt, welke andere engine ontdekt zijn, welke datastores gebruikt moeten worden, etc.. Via de ellipsis knop kunt u browsen naar de juiste lokatie.

Als u al eerder een engine heeft aangemaakt, en u wilt de instellingen en data ophalen uit het Data bestand, zet u een vinkje voor de 'Read settings from file' optie. Let op, de naam en het adres worden dan ook uit dit bestand gehaald, en dus worden de ingevulde waarden in het venster genegeerd.

Het vinkje 'Automatically start service' zorgt ervoor dat, na het sluiten van het Create Engine venster, de service wordt opgestart. Ook nu kan het zijn dat eventuele beveiligingsprogramma’s vragen om een bevestigen van het starten van de service.

Verder beheer van de engine verloopt op dezelfde wijze als het beheer van een engine in de DOL Manager applicatie. Zie dus de handleiding van de DOL Manager, eerder in dit document, voor het beheren van de nieuwe engine(s).

**E.4 Demo Definition Viewer**

De laatste applicatie die in bijgevoegd is een demo applicatie die gebruik maakt van de gedistribueerde Gellish datastores. Om de applicatie te kunnen draaien moet u in ieder geval een engine lokaal installeren, aangezien de demo applicatie deze gebruikt voor het uitzetten van een query en ontvangen van een antwoord.

Deze applicatie is gebaseerd op de Object Browser web applicatie ontwikkeld door Idoro en laat u zoeken naar objecten op basis van een term. Deze zoekopdracht kunt u verder uitbreiden door een string commonality en hoofdletter gevoeligheid op te geven.

Het scherm van deze applicatie ziet er uit zoals figuur E.20. Het bovenste gedeelte van dit venster stelt u in staat om uw zoek term en aanvullende criteria, zoals string commonality, op te geven. Vervolgens start de Gellish query welke zoekt naar objecten die voldoen aan
deze naam door op de Search knop te klikken. Deze query wordt vervolgens gedistribueerd uitgezet en het antwoord in de vorm van Gellish feiten wordt geïnterpreteerd door de applicatie en eventuele objecten worden in de lijst onderin het scherm weergegeven.

Vervolgens kunt u, na het selecteren van een object in de lijst, klikken op de Specializations knop of de Definitions knop. De Specializations knop zet opnieuw een query uit welke als resultaat een soortenboom toont (figuur 23). De Definitions knop zet ook een gedistribueerde query uit en haalt feiten op die betrekking hebben op de definitie van het object (figuur E.22).

Figure E.20: Hoofdvenster Demo Definition viewer.
Figure E.21: Voorbeeld van een soortenboom.

Figure E.22: Definition venster Demo Definition viewer.