Bootstrapping ontology-based data access specifications from relational databases

Bachelor thesis
by
stud. inf. Philipp Martis

realized at the
Institute for Parallel and Distributed Systems,
University of Stuttgart

Stuttgart, in May 2016

Abstract

Nach der Titelseite des Berichtes und dem Aufgabenblatt soll das Wesentliche aus dem Inhalt der Arbeit in wenigen Sätzen zusammengefasst werden. Diese Übersicht soll keine Formeln und möglichst keine Literaturhinweise enthalten.

Kurzfassung

Nach der Titelseite des Berichtes und dem Aufgabenblatt soll das Wesentliche aus dem Inhalt der Arbeit in wenigen Sätzen zusammengefasst werden. Diese „Übersicht“ soll keine Formeln und möglichst keine Literaturhinweise enthalten.

Contents

List of Figures

List of Tables

1 Introduction

Sie führt in die Problematik ein, skizziert die Motivation und Zielsetzung sowie das geplante Vorgehen und die angestrebten Ergebnisse und sollte ca. 1 - 2 Seiten umfassen.

1.1 Motivation

As estimated in 2007 [HPZC07], publicly available databases contained up to 500 times more data than the static web and roughly 70 % of all websites were backed by relational databases back then. As hardware has become cheaper yet more powerful, open source tools have become more and more widespread and the web has gotten more and more dynamic and interactive, it's likely that these numbers have even increased since then. This makes the publication of available data in a structured, machine-processable form and its retrieval with eligible software (Ontology based data access, OBDA) an interesting topic. This vision emerged as early as TODO and was entitled with the term "semantic web" by Tim Berners-Lee [BLHL01]. Definitely, the automatic translation of relational databases to RDF or similar representations of structured information is an integral part of the success of the semantic web [HPZC07]. This automatic translation process is commonly called "bootstrapping".

Early work regarding the development of bootstrapping systems includes TODO. Today, the pure translation process is a relatively well understood topic, ranging from the rather simple direct mapping approach [W3C12] to TODO. On the other hand, the handling of the complexity introduced by these approaches and the use of sophisticated tools to perform various related tasks meanwhile has become a significant challenge in its own right [SGH+15]. Besides the parametrization of the tools in use, this includes the management of the several kinds of artifacts accruing during the process, possibly needed in different versions and formats for the use of different tools and output formats, while also taking changing input data into account [SGH+15]. Skjæveland and others therefore suggested an approach using a declarative description of the data to be mapped, concentrating in one place all the information needed to coordinate the bootstrapping process and to drive the entire tool chain [SGH+15].

1.2 Approach

This thesis describes the development of a specification language to serialize the declarative specification of the bootstrapping process and of a software to in turn bootstrap it from a relational database schema. After the tasks they accomplish, the specification language was called "OBDA Specification Language" ("OSL") and the software bootstrapping the specification was called "db2osl".

Using a declarative specification makes the entire bootstrapping process a two-step-procedure: First, the OBDA specification is derived from the database schema using DB2OSL. It specifies the actual bootstrapping process in a very general way, so it only has to be recreated when the database schema changes. The second step is to use the OBDA specification to coordinate and drive the actual bootstrapping process. The development of a software that uses the OBDA specification to perform this second step currently is subject to ongoing work. It will be able to be parameterized accordingly to support different output formats, tools, tool versions and application ranges.

TODO: illustration of overall process

1.3 Requirements and goals

The final system shall be able to cleanly fit into existing bootstrapping systems while being easy to use, taking the burden of dealing with OSL specifications manually from its users instead of adding even more complexity to the process. To achieve the former goal, use of existing tools, languages and conventions was made wherever possible. To fit into the environment used in the OPTIQUE project TODO it is ultimately part of, Java was used for the bootstrapping software. Care was taken to design it to be modular and flexible, making it usable not only as a whole but also as a collection of independent components possibly serving as the basis for a program library in the future. To further support this aim TODO and to make the software more easily understandable and extensible, it was documented carefully and thoroughly.

As the software will be maintained by diverse people after its development and will likely be subject to changes, general code quality was also an issue to consider. Following good object-oriented software development practice TODO, real world artifacts like database schemas, database tables, columns, keys, OBDA specifications et cetera were modeled as software objects, provided with a carefully chosen set of operations to manipulate them and make them collaborate. Scarce, informative comments were inserted at places of higher complexity and to expose logical subdivisions, but care was taken to use “speaking code” in favor of rampant comments. Complex, misleading and hard-to-use interfaces were avoided wherever possible. External software libraries employed were chosen to be stable, specific, well structured, publicly available and ideally in wide use.

2 Background and related work

Bei experimentellen Untersuchungen sind die benutzte Versuchsanordnung sowie Messanordnungen und Messverfahren ausführlich zu beschreiben. Analog sind die Grundlagen eventuell verwendeter oder erweiterter Berechnungsprogramme zusammenfassend darzustellen. Bei übernommenen Formeln, Bildern, Tabellen und bei Zitaten ist stets die Quelle durch den Namen des Verfassers und die zugehörige Nummer im Literaturverzeichnis anzugeben (z.B. [SGH+15])

Nam eu dolor a nisl faucibus suscipit. Nulla interdum sapien id lectus. Curabitur fringilla pulvinar nibh. Aenean porta luctus purus. Cras dictum mauris quis velit. Nullam pharetra pede at risus. Nullam orci sapien, porttitor eu, iaculis et, bibendum ultricies, ipsum. Mauris eget justo. Donec semper auctor tortor. Mauris a ante et magna facilisis mollis. Proin sem turpis, interdum quis, fermentum aliquet, faucibus scelerisque, quam. In mi nibh, facilisis eu, euismod sed, luctus ut, sapien. Etiam ut dui eget libero dapibus elementum.

Nulla ut felis et libero tempus luctus. Praesent vitae velit. Vivamus pharetra pharetra sem. Morbi id mauris. Ut sem mauris, fermentum non, interdum eu, nonummy non, nunc. Aenean a sem et odio ornare dictum. Phasellus fermentum justo quis justo. Aenean et felis. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Curabitur faucibus ornare augue. Nullam blandit pellentesque odio. Vestibulum tempor tempor ante. Etiam scelerisque elementum diam. Vestibulum enim sem, dictum et, rhoncus vitae, ullamcorper ut, dolor. Sed diam.

Sed dignissim diam vel erat. Pellentesque ac lacus sed dui vehicula tristique. Curabitur justo sapien, convallis in, faucibus nec, hendrerit eu, sapien. Quisque imperdiet lacus vitae lacus. Vestibulum aliquet rutrum enim. Fusce et leo. Ut id dui non felis rhoncus laoreet. Nullam ipsum. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Cras et dolor. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

2.1 Background

2.2 Related work

3 The OBDA Specification Language (OSL)

As described in [SGH+15], an OBDA specification consists of several types of maps, all containing data entries and links to other maps. This fits perfectly into the environment of ontologies and OWL, with data properties being the obvious choice to represent contained data entries and object properties being the obvious choice to represent links between maps. Also, a potential user probably to some degree is familiar with this environment, since this is what the bootstrapping process at the end amounts to.

Therefore, an ideal base for the OBDA Specification Language is OWL, being a solid framework for data and constraint representation with a high degree of software support, while imposing only a minimum of introductory preparation to the user.

Another advantage of this approach is that the specification is kept compact and focused on the entities that the language has to represent rather than primarily dealing with technical details. In particular, many of those details can be formulated as OWL restrictions in a header ontology demanded to be imported by documents conforming to the OSL specification. Thus, they are not only specified precisely but they are also stipulated in a machine-readable form for which tools are widely available, enabling the user to check many aspects of an OSL document for conformity with minimal effort.

3.1 Specification

[1] An OSL document is a valid OWL 2 document (as described in [W3C12]) containing individuals and data that represent the OBDA Specification, as well as OWL properties that connect them. The individuals and OWL properties are recognized and mapped to their roles by their IRIs.

[2] An OSL document may contain more OWL entities (with IRIs not defined in this specification), which are ignored.

[3] An OSL document has to declare all individuals having different IRIs as different from each other (except those which are ignored, see paragraph 2).
It is recommended to use the `owl:AllDifferent` OWL statement for this purpose.

[4] Unless stated otherwise, IRIs mentioned in the following are IRIs relative to a base IRI chosen by the user being empty (which makes the IRIs absolute [W3C09]) or ending with a hash character ('#').

Map type	OWL IRI
Entity map	<*class URI*>`__ENTITY_MAP`
Attribute map	<*property URI*>`__ATTRIBUTE_MAP`
Identifier map	<*class URI*>`__IDENTIFIER_MAP`
Relation map	<*property URI*>`__RELATION_MAP`
Subtype map	<*class URI*>`__SUBTYPE_MAP`
Translation table of attribute map	<*property URI*>`__ATTRIBUTE_MAP__TRANSLATION_TABLE`
Translation table of subtype map	<*class URI*>`__SUBTYPE_MAP__TRANSLATION_TABLE`

Table 3.1: OWL individual IRIs in OSL

Map type	OWL class IRI
Entity map	`osl:EntityMap`
Attribute map	`osl:AttributeMap`
Identifier map	`osl:IdentifierMap`
Relation map	`osl:RelationMap`
Subtype map	`osl:SubtypeMap`
Translation table	`osl:TranslationTable`

Table 3.2: Class membership of map representations in OSL

It is recommended to use that base IRI as `xml:base` XML attribute.
IRIs prefixed with `osl:` are IRIs relative to the IRI
http://w3studi.informatik.uni-stuttgart.de/~martispp/ont# .

[5] An OSL document has to import the following ontology (referred to as "the OSL header" in the following):
http://w3studi.informatik.uni-stuttgart.de/~martispp/ont/db2osl.owl

[6] The OWL individuals described by the OSL document representing the certain types of OBDA maps must have the IRIs specified in table 3.1 (for base IRIs, see paragraph 4). Here, <*class URI*> refers
to the `OWL class URI` field of the respective entity map for entity maps,
to the `OWL class URI` field of the associated entity map for identifier maps,
to the `OWL class URI` field of the associated entity map for subtype maps and
to the `OWL class URI` field of the entity map associated with the respective subtype map for translation tables of subtype maps.
Similarly, <*property URI*> refers
to the `OWL property URI` field of the respective attribute map for attribute maps (or, if it is empty, the value that would have been generated for it if it weren't empty),
to the `OWL property URI` field of the respective relation map for relation maps and
to the `OWL property URI` field of the respective attribute map for translation tables of attribute maps (or, if it is empty, the value that would have been generated for it if it weren't empty).

[7] The OWL individuals described by the OSL document representing the certain types of OBDA maps must be of the OWL types specified in table 3.2 (for base IRIs, see paragraph 4).

[8] The OWL properties described by the OSL document representing the fields of the certain

OBDA maps must have the IRIs specified in table 3.3 (for base IRIs, see paragraph 4).

[9] The following OWL properties in the OSL document refer to lists of elements:

`osl:rm__sourceColumns`

`osl:rm__targetColumns`

`osl:tt__sourceValues`

`osl:tt__rdfRessources`

Therefore, they have the OWL class `osl:StringListNode` as their range, as is required by the OSL header. They must connect the respective individual to an `osl:StringListNode` individual in every case. This "root node" must *not* have an `osl:hasValue` property.
If the represented list is not empty, the list elements are represented by other `osl:StringListNode` individuals connected seriatim by the property `osl:nextNode`, with the first individual being connected to the root node. The node representing the last list element must not have an `osl:nextNode` property.
All nodes except the root node *may* have an `osl:hasValue` property connecting them to their values. The actual list consists of exactly these values, thus, nodes without values are ignored.
It is recommended to enumerate the node IRIs, using 0 for the root node.

Map type	Field label	Field name	OWL IRI
Entity map	E1	Table name	osl:em__tableName
Entity map	E2	Label	osl:em__label
Entity map	E3	Identifier map	osl:em__identifierMap
Entity map	E4	Attribute maps...	osl:em__attributeMaps
Entity map	E5	OWL class URI	osl:em__owlClassURI
Entity map	E6	Description	osl:em__description
Attribute map	A1	Column name	osl:am__columnName
Attribute map	A2	SQL datatype	osl:am__sqlDatatype
Attribute map	A3	Mandatory	osl:am__mandatory
Attribute map	A4	Label	osl:am__label
Attribute map	A5	OWL property URI	osl:am__owlPropertyURI
Attribute map	A6	Property type	osl:am__propertyType
Attribute map	A7	Translation	osl:am__translation
Attribute map	A8	URI pattern	osl:am__uriPattern
Attribute map	A9	RDF language	osl:am__rdfLanguage
Attribute map	A10	XSD datatype	osl:am__xsdDatatype
Attribute map	A11	Description	osl:am__description
Identifier map	I1	Entity map	osl:im__entityMap
Identifier map	I2	Attribute maps...	osl:im__attributeMaps
Identifier map	I3	URI pattern	osl:im__uriPattern
Relation map	R1	Source entity map	osl:rm__sourceEntityMap
Relation map	R2	Source column	osl:rm__sourceColumns
Relation map	R3	Target entity map	osl:rm__targetEntityMap
Relation map	R4	Target column	osl:rm__targetColumns
Relation map	R5	OWL property URI	osl:rm__owlPropertyURI
Subtype map	S1	Entity Map	osl:sm__entityMap
Subtype map	S2	Column Name	osl:sm__columnName
Subtype map	S3	OWL superclass URI	osl:sm__owlSuperclassURI
Subtype map	S4	Prefix	osl:sm__prefix
Subtype map	S5	Suffix	osl:sm__suffix
Subtype map	S6	Translation	osl:sm__translation
Translation table	T1	Source value...	osl:tt__sourceValues
Translation table	T2	RDF ressource...	osl:tt__rdfRessources

Table 3.3: OWL property IRIs in OSL

4 The db2osl software

Besides the conception of the "OBDA Specification Language" (OSL), the design and implementation of the DB2OSL software was an important part of this work. The program itself and its creation process are described in the following sections: Section 4.1 – Functionality describes the functionality the program offers. Section 4.2 – Interface describes how this functionality is exposed to the program environment. Section 4.3 – Tools employed explains what tools where used to create the program. Section 4.4 – Architecture describes the program architecture both on a coarse and a fine level. Section 4.5 – Code style describes concepts and decisions that where implemented on the code level to yield clean code. Section 4.6 – Numbers and statistics mentions some figures about the program. Section 4.7 – Versioning gives a brief timewise TODO overview over the program development and describes important milestones.

Except the last section, this chapters' sections present the information in a functionally-structured fashion: the concepts and decisions are described along with the topics they are linked to and the problems that made them arise. However, the last section, besides giving an overview about the program versions, tries to give an insight about development succession.

Unless stated differently, program version 1.0 is described (for details, see section 4.7 – Versioning).

4.1 Functionality

As described in the introduction, the DB2OSL software basically is a program which automatically derives an OBDA specification from a relational database schema, which then can be used by other tools to drive the actual bootstrapping process. Its functionality is described in the next section (while leaving out self-evident features) and is then listed completely in the section after that.

4.1.1 Description

The database schema is retrieved by connecting to an SQL database and querying its schema information. Parsing SQL specifications TODO or SQL dumps currently is not supported. The databases to derive information from can be specified by regular expressions, while there are also options to use other databases than specified or even other database servers, taken from a list of hard-coded strings. While these features may not seem to carry real benefit at the first glance, they proved to be very useful for testing purposes, especially since the retrieval of a database schema can take some time (see TODO). For the same purpose, DB2OSL allows the processing of a hard-coded example database schema.

In addition to OSL output, a low-level output format containing information on all fields of

the underlying objects is supported, which is useful for debugging (however, this feature has to be enabled via one slight change in the source code). To allow for some customization, the insertion of an own OSL header is supported (for more information on the OSL header, see the specification of the OSL language in section 3.1). If the standard OSL header is used, it is by default loaded from a hard-coded copy, so bootstrapping information from a database server running locally or from the hard-coded example schema requires no Internet connection (simply inserting the `owl:imports` statement of course would not anyway, but the generated underlying ontology is always checked for consistency with the OSL header to prevent the generation of invalid output).

The DB2OSL software can be used both in an interactive and in a non-interactive mode, while skipping a database or a database server or aborting the entire bootstrapping process is possible in either mode. Multiple database servers can be specified for a bootstrapping operation, which then are checked in order for a matching database, allowing to make use of mirrors or fallback servers. Additionally, multiple bootstrapping operations can be specified to be performed in sequence with one invocation of DB2OSL, while all features and settings previously described are enabled, disabled or set per operation. Finally, a help text can be displayed which describes the usage of DB2OSL including the description of all command-line arguments.

4.1.2 Summary

The functionality of the DB2OSL software can be summarized as follows:

- Bootstrap one or more OBDA specifications from a database schema by connecting to an SQL database server
- Specify a custom port, login and password for the database server
- Ask for passwords interactively (before starting any bootstrapping operation), hide them if desired
- Specify database names by regular expressions
- Process an arbitrary database if the specified database could not be found or unconditionally
- Connect to a database server containing example databases without having to specify any further details
- Process a hard-coded example database schema without having to specify any further details
- Use the OSL format described in section 3 – The OBDA Specification Language (OSL) or a detailed low-level format for output (the latter is for debugging purposes and has to be enabled in the source code)
- Write to standard output or to a file
- Insert a custom OSL header (see the specification of the OBDA Specification Language (OSL) in section 3.1 for details)
- Consistency check against a custom OSL header

- Consistency check against the standard OSL header without internet connection
- Act interactively or non-interactively
- Skip currently retrieved database (and try next on server), skip current server or abort the overall process at any time, even in non-interactive mode
- Define multiple database servers to check in order for the specified database
- Specify multiple bootstrapping operations to perform in order
- Configure the features described in the above notes per bootstrapping operation
- Display a help text describing the usage of DB2OSL, including the description of all command-line arguments

4.2 Interface

This section describes the interface to the operating system and the user interface. For information on programming interfaces, see section 4.4 – Architecture.

4.2.1 User interaction and configuration

Basic usage

Currently, the only user interface of DB2OSL is a command-line interface. Since the program is supposed to bootstrap the OBDA Specification automatically and thus there is little interaction, but a lot of output, this was considered ideal. Because of its ability to write to the standard output (which is also the default behavior), it is easy to pipe the output of DB2OSL directly to a program that handles it in a Unix-/POSIX-like fashion TODO:

```
db2osl myserver.org | osl2onto myserver.org
```

(supposed OSL2ONTO is a tool that reads an OSL specification from its standard input and uses it to bootstrap an ontology from the database specified on its command line).
This scheme is known as "filter-pattern" TODO.

By inserting additional "filters", the bootstrapping process can be customized without changing any of the involved programs:

```
db2osl mydatabase.org | customize_uris.sh | osl2onto mydatabase.org
```

(supposed CUSTOMIZE_URIS.SH is a shell script that modifies the URIs contained in an OSL specification in the manner the user desires).

Command-line arguments

The behavior of DB2OSL itself can be adjusted via command-line arguments. The syntax for their application follows the POSIX standard TODO. Most features can be configured via

Option(s)	Description, taken from the help page of DB2OSL
`--database, -d`	database name (Java regular expression) databases have to match to be p
`--echo-password`	echo input when prompting for SQL password – must be specified before
`--help, -h,`	show this help and exit
`--interactive, -i`	be interactive when chosing database
`--login, -L`	SQL login
`--loose-database-match`	if no database matching the regex specified with `--database` is found on
`--osl-header`	use the specified custom (non-standard) OSL header, implies `--remote-c`
`--output-file, -o`	use the specified output file (for the standard output, specify "-")
`--password, -P`	SQL password; use `--password-prompt` to get a password prompt (if you
`--password-prompt, -p`	prompt for SQL password; a password set via `--password` is ignored
`--remote-osl-header, -R`	don't use hard-coded version of the OSL header for verification
`--remote-test`	try to retrieve a database schema from a hard-coded list of servers and ta
`--test`	use hard-coded test database schema, ignore given servers (note: give a d

Table 4.1: Command-line arguments in DB2OSL

short options (as, for example, `-P`). To allow for enhanced readability of DB2OSL invocations, each feature can (also) be configured via a long option (like `--password`).

The command-line arguments DB2OSL currently supports and their effects are described in table 4.1. There currently is no switch to set the output format, since the only supported output format, besides OSL, is a low-level output format for debugging purposes. Because of this and since the change that has to be made in the source code to enable it only involves changing one token, it was preferred not to offer a command-line option for this, thus not unnecessarily complicating the command-line interface for the normal, non-debugging, user.

The sole invocation of DB2OSL, without any arguments, does not initiate any processing but displays the usage directions instead, in addition to an error message.

Multiple bootstrapping operations or multiple servers

To perform multiple bootstrapping operations with one invocation of DB2OSL, it is sufficient to concatenate the command-line arguments for each operation, separated by blanks, to get the final command line. However, when combining a test job with other operations, some arbitrary string has to be inserted as dummy server to allow distinguishing the different jobs and assigning each command-line argument to the appropriate job.

On the other hand, to check several servers in order for the database to be used for one bootstrapping operation, these servers have to be concatenated, separated by blanks. Again, the distinction of the different bootstrapping jobs has to be possible, so all but the first operation have to have at least one command-line argument (which is no practical problem, since to enforce this, a default argument simply can be stated explicitly without changing the behavior of the invocation).

All settings are configured per operation, so, when using a shell that separates batched commands by ';',

```
db2osl --database employees --password itsme sql.myemployer.com
```

```
--database test myserver.org backup.myserver.org
```

is equivalent to

```
db2osl --database employees --password itsme sql.myemployer.com;
db2osl --database test myserver.org backup.myserver.org
```

Thus, currently

4.2.2 Integration into systems

TODO: modularity -> reusability TODO: logger TODO: operating system -> Java

4.3 Tools employed

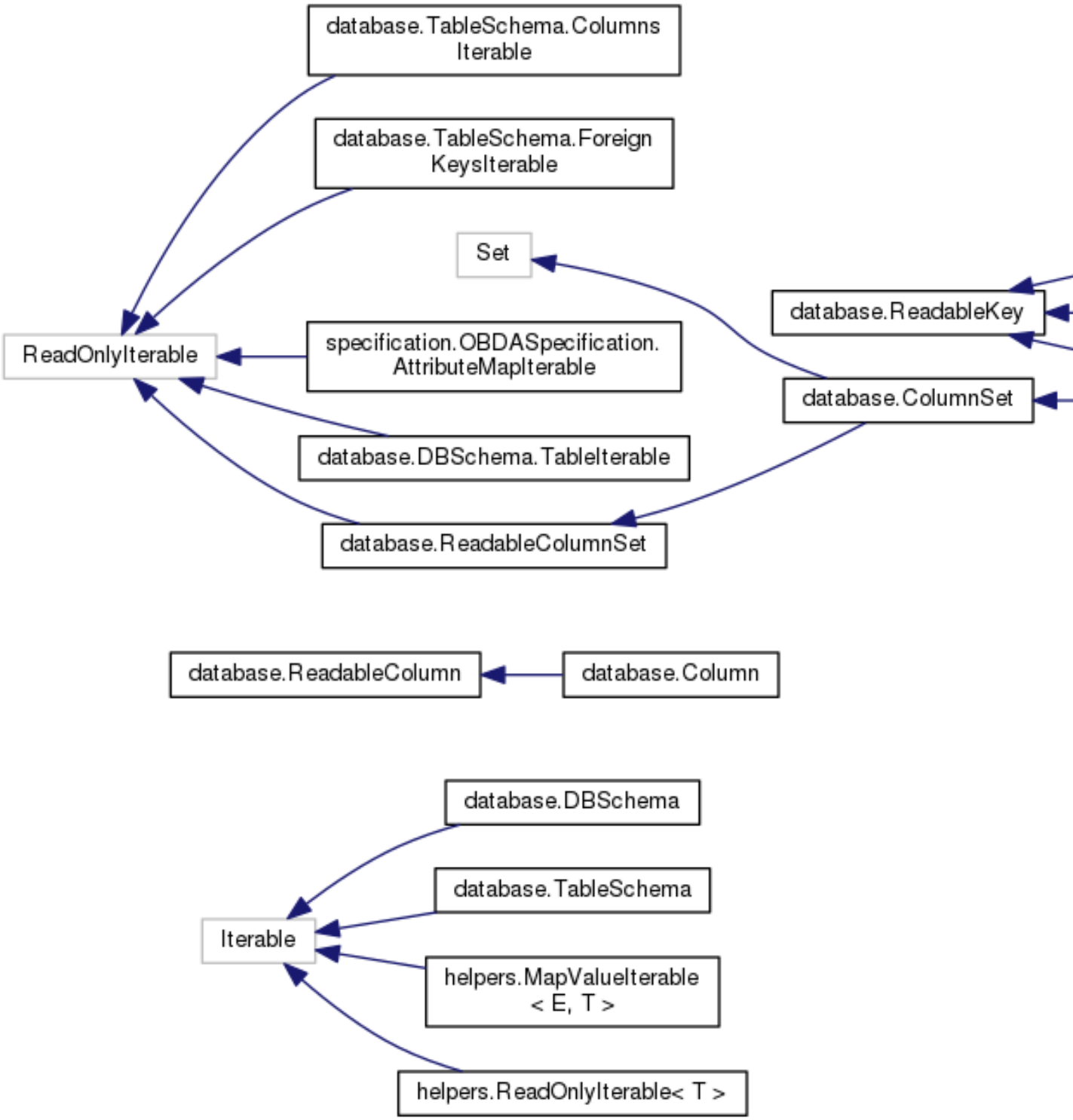
4.4 Architecture

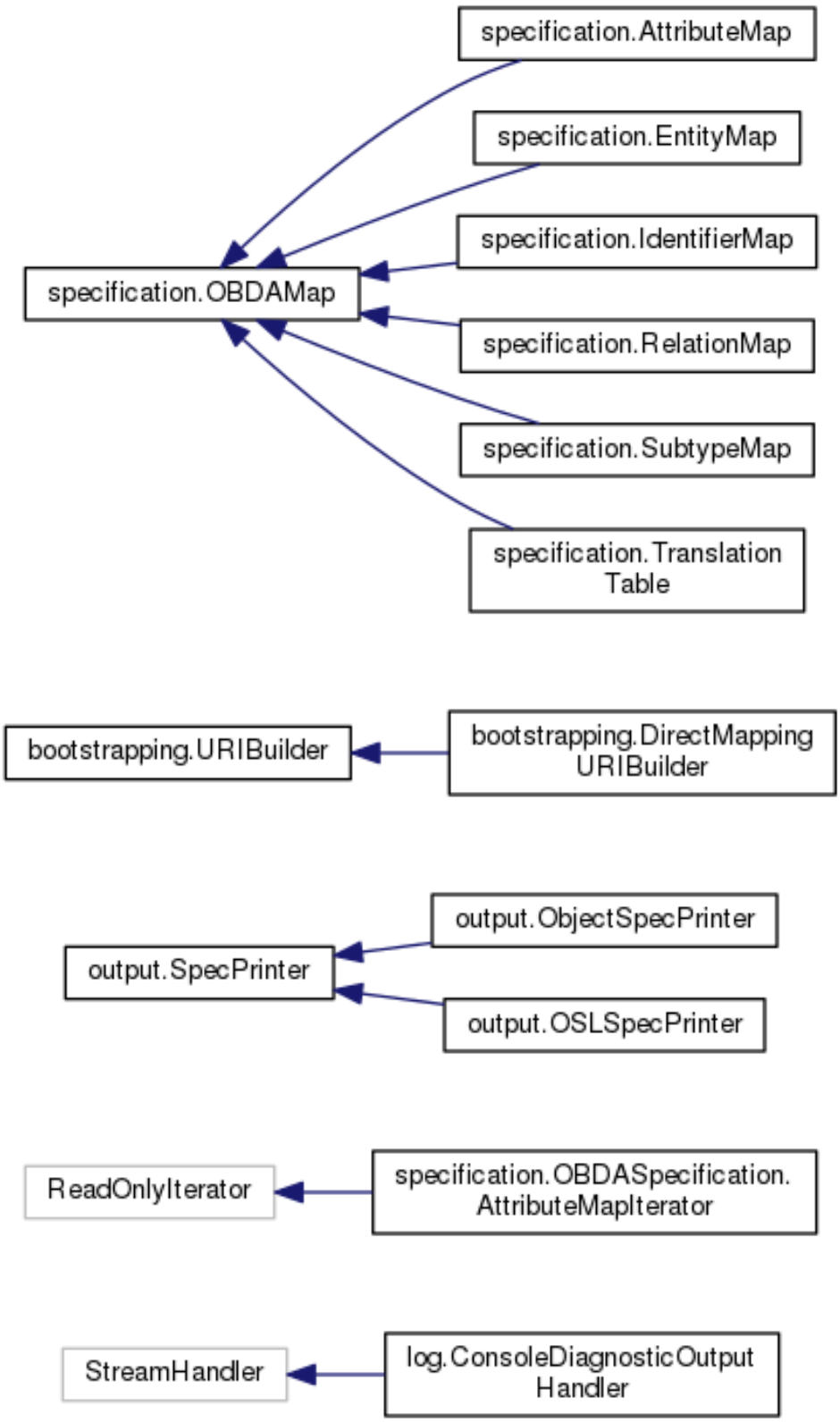
4.4.1 Libraries used

4.4.2 Coarse structuring

TODO: overall description, modularity, extendability, ex: easy to add new in-/output formats TODO: mapping profiles (maybe better in next subsection) TODO: package description TODO: package interaction description

4.4.3 Fine structuring




specification.AttributeMap
specification.EntityMap
specification.IdentifierMap
specification.OBDAMap
specification.RelationMap
specification.SubtypeMap
specification.Translation Table
bootstrapping.URIBuilder
bootstrapping.DirectMapping URIBuilder
output.ObjectSpecPrinter
output.SpecPrinter
output.OSLSpecPrinter
ReadOnlyIterator
specification.OBDASpecification. AttributeMapIterator
StreamHandler
log.ConsoleDiagnosticOutput Handler

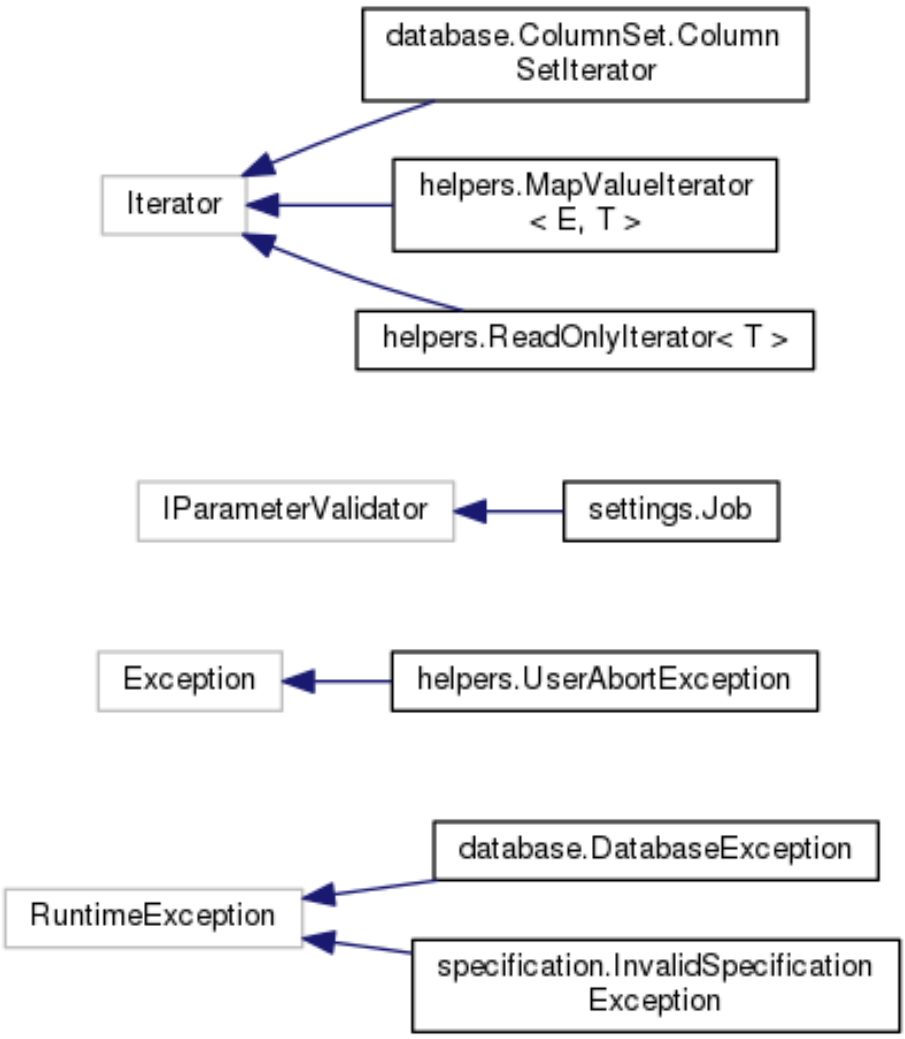


Figure 4.1: Class hierarchies in DB2OSL

- `main.Main`
- `database.Helpers`
- `database.RetrieveDBSchema`
- `database.SQLType`
- `database.Table`
- `specification.OBDASpecification`
- `osl.OSLSpecification`
- `bootstrapping.Bootstrapping`
- `cli.CLIDatabaseInteraction`
- `log.GlobalLogger`
- `test.CreateTestDBSchema`
- `test.GetSomeDBSchema`

Table 4.2: Standalone classes in DB2OSL

4.5 Code style

TODO: Conventions, ex.: iterators As the final system hopefully will have a long living cycle TODO and will be used and refined by many people, high code quality was an important aim. Beyond architectural issues this also involves cleanness on the lower level, like the design of classes and the implementation of methods. Common software development principles were followed TODO and the unfamiliar reader was constantly taken into account to yield clean, usable and readable code.

4.5.1 Comments

Comments were used at places ambiguities or misinterpretations could arise, yet care was taken to face such problems at their roots and solve them wherever possible instead of just eliminating the ambiguity with comments.

Consider the following method in CLIDatabaseInteraction.java:

```
public static void promptAbortRetrieveDBSchemaAndWait
     (final FutureTask<DBSchema> retriever) throws SQLException
```

It could have been called `promptAbortRetrieveDBSchema` only, with the waiting mentioned in a comment. However, the waiting is such an important part of its behavior, that this wouldn't have been enough, so the waiting was included in the function name. Since the method is called at one place only, the lengthening of the method name by 7 characters or about 26 % is really not a problem.

More generally, "speaking code" was used wherever possible, as described in section 4.5.2 – Speaking code, which rendered many uses of comments unnecessary. In fact, the number of (plain, e.g. non-Javadoc) comments was consciously minimized, to enforce speaking code and avoid redundancy. This technique is known TODO.

An exception of course from this is the highlighting of subdivisions. In class and method implementations, comments like

```
//********************** Constructors **********************TODO
```

were deliberately used to ease navigation inside source files for unfamiliar readers, but also to enhance readability: independent parts of method implementations, for example, were optically separated this way. Another alternative would have been to use separate methods for this code pieces, as was done in other cases, but this would then have introduced additional artifacts with either long or non-speaking names. Additionally, it would have increased complexity, because these methods would have been callable at least from everywhere in the source file, and would have interrupted the reading flow. This technique is known TODO, while TODO

Wherever possible, appropriate Javadoc comments were used in favor of plain comments, for example to specify parameters, return types, exceptions and links to other parts of the documentation.

4.5.2 Speaking code

As mentioned in section 4.5.1 – Comments, the use of "speaking code" as introduced TODO renders many uses of comments unnecessary. In particular, the following aspects are commonly considered when referring to the term "speaking code" TODO:

- Variable names
- Control flow

Variable names

A very important part of speaking code

4.5.3 Robustness against incorrect use

Care was taken to produce code that is robust to incorrect use, making it suitable for the expected environment of sporadic updates by unfamiliar and potentially even unpracticed programmers who very likely have their emphasis on the concepts of bootstrapping rather than details of the present code.

In fact, carefully avoiding the introduction of technical artifacts to mind, preventing programmers from focusing on the actual program logic, is an important principle of writing clean code TODO.

In modern programming languages, of course the main instruments for achieving this are the type system and exceptions. In particular, static type information should be used to reflect data abstraction and the "kind" of data, an object reflects, while dynamic type information should only be used implicitly, through dynamically dispatching method invocations[**str4**]. Exceptions on the other hand should be used at any place related to errors and error handling, separating error handling noticeably from other code and enforcing the treatment of errors, preventing the programmer from using corrupted information in many cases.

An example of both mechanism, static type information and exceptions, acting in combination, while cleanly fitting into the context of dynamic dispatching, are the following methods from Column.java:

```
public Boolean isNonNull()
public Boolean isUnique()
```

There return type is the Java class `Boolean`, not the plain type `boolean`, because the information they return is not always known. In an early stage of the program, they returned `boolean` and were accompanied TODO by two methods `public boolean knownIsNonNull()` and `public boolean knownIsUnique()`, telling the caller whether the respective information was known and thus the value returned by `isNonNull()` or `isUnique()`, respectively, was reliable.

They were then changed to return the Java class `Boolean` and to return null pointers in case the respective information is not known. This eliminated any possibility of using unreliable

data in favor of generating exceptions instead, in this case a `NullPointerException`, which is thrown automatically by the Java Runtime Environment if the programmer forgets the null check and tries to get a definite value from one of these methods when the correct value currently is not known.

Comparing two unknown values – thus, two null pointers – also yields the desired result, `true`, since the change, even when the programmer forgets that he deals with objects. However, when comparing two return values of one of the methods in general – as opposed to comparing one such return value against a constant –, errors could occur if the programmer writes `col1.isUnique() == col2.isUnique()` instead of `col1.isUnique().booleanValue() == col2.isUnique().booleanValue()`.
TODO: Java rules.

TODO: more, summary

4.5.4 Classes

Following the object-oriented programming paradigm, classes were heavily used to abstract from implementation details and to yield intuitively usable objects with a set of useful operations [**obj**].

Identification of classes

To identify potential classes, entities from the problem domain were – if reasonable – directly represented as Java classes. The approach of choosing "the program that most directly models the aspects of the real world that we are interested in" to yield clean code, as described and recommended by Stroustrup [**str3**], proved to be extremely useful and effective. As a consequence, the code declares classes like `Column`, `ColumnSet`, `ForeignKey`, `Table`, `TableSchema` and `SQLType`. As described in section 4.5.2 – Speaking code, class names were chosen to be concise but nevertheless expressive TODO. Java packages were used to help attain this aim, which is why the previously mentioned class names are unambiguous (for details about package use see section 4.5.5 – Packages).

Care was taken not to introduce unnecessary classes, thereby complicating code structure and increasing the number of source files and program entities. Especially artificial classes, having little or no reference to real-world objects, could most often be avoided. On the other hand of course, it usually is not the cleanest solution to avoid such artificial classes entirely.

Const correctness

Specifying in the code which objects may be altered and which shall remain constant, thus allowing for additional static checks preventing undesired modifications, is commonly referred to as "const correctness" TODO.

Unfortunately, Java lacks a keyword like C++'s `const`, making it harder to achieve const correctness. It only specifies the similar keyword `final`, which is much less expressive and doesn't allow for a similarly effective error prevention [**final**]. In particular, because `final`

is not part of an object's type information, it is not possible to declare methods that return read-only objects TODO – placing a `final` before the method's return type would declare the method `final`. Similarly, there is no way to express that a method must not change the state of its object parameters. A method like `public f(final Object obj)` is only liable to not assigning a new value to its parameter object `obj` [**java**] (which, if allowed, wouldn't affect the caller anyway [**java**]). Methods changing its state, however, are allowed to be called on `obj` without restrictions [**java**].

Several possibilities were considered to address this problem:

- Not implementing const correctness, but stating the access rules in comments only
- Giving the methods which modify object states special names like `setName--USE_WITH_CARE`
- Delegating changes of objects to special "editor" objects to be obtained when an object shall be altered
- Deriving classes offering the modifying methods from the read-only classes

Not implementing const correctness at all of course would have been the simplest possibility, producing the shortest and most readable code, but since incautious manipulation of objects would possibly have introduced subtle, hard-to-spot errors which in many cases would have occurred under additional conditions only and at other places, for example when inserting a `Column` into a `ColumnSet`, this method was not seriously considered.

Using intentionally angular, conspicuous names also was not considered seriously, since it would have cluttered the code for the only sake of hopefully warning programmers of possible errors – and not attempting to avoid them technically.

So the introduction of new classes was considered the most effective and cleanest solution, either in the form of "editor" classes or derived classes offering the modifying methods directly. Again – as in the identification of classes –, the most direct solution was considered the best, so the latter form of introducing additional classes was chosen and classes like `ReadableColumn`, `ReadableColumnSet` et cetera were introduced which offer only the read-only functionality and usually occur in interfaces. Their counterparts including modifying methods also were derived from them and the implications of modifications were explained in their documentation, while the issue and the approach as such were also mentioned in the documentation of the `Readable...` classes. The `Readable...` classes can be converted to their fully-functional counterparts via downcasting (only), thereby giving a strong hint to programmers that the resulting objects are to be used with care.

Java interfaces

In Java programming, it is quiet common and often recommended, that every class has at least one `interface` it `implemen,ts`, specifying the operations the class provides. TODO If no obvious `interface` exists for a class or the desired interface name is already given to some other entity, the interface is often given names like `ITableSchema` or `TableSchemaInterface`.

However, for a special purpose program with a relatively fixed set of classes mostly representing real-world artifacts from the problem domain, this approach was considered overly cluttering, introducing artificial code entities for no benefit. In particular, as explained in

section TODO, all program classes either are standing alone TODO or belong to a class hierarchy derived from at least one interface. So, except from the standalone classes, an interface existed anyway, either "naturally" (as in the case of `Key`, for example) or because of the chosen way to implement const correctness. In some cases, these were interfaces declared in the program code, while in some cases, Java interfaces like `Set` were implemented (an obvious choice, of course, for `ColumnSet`). Introducing artificial interfaces for the standalone classes was considered unnecessary at least, if not messy.

4.5.5 Packages

As mentioned in section 4.5.4 – Classes, class names were chosen to be concise but nevertheless expressive. This only was possible through the use of Java `package`s, which also helped structure the program.

For the current, relatively limited, extent of the program which currently comprises 45 (`public`) classes, a flat package structure was considered ideal, because it is simple and doesn't stash source files deep in subdirectories (in Java, the directory structure of the source tree is required to reflect the package structure TODO). Because also every class belongs to a package, each source file is to be found exactly one directory below the root program source directory, which in many cases eases their handling.

The following 11 packages exist in the program (their purpose and more details about the package structure are described in section TODO):

- `boostrapping`
- `cli`
- `database`
- `helpers`
- `log`
- `main`
- `osl`
- `output`
- `settings`
- `specification`
- `test`

TODO: two columns

Each package is documented in the source code also, particularly in a file package-info.java residing in the respective package directory. This is a common scheme supported by the ECLIPSE IDE as well as the documentation generation systems JAVADOC and DOXYGEN TODO (all of which were used in the creation of the program, as described in section TODO).

4.6 Numbers and statistics

TODO: Retrieval times, (NC)LOC and other statistics

4.7 Versioning

5 Summary and future work

Für den eiligen Leser ist die Vorgehensweise zusammen mit den wesentlichen Ergebnissen am Schluss in einer “Zusammenfassung” klar herauszustellen. Diese soll ausführlicher sein als die “Übersicht” am Anfang der Arbeit. Auch diese Zusammenfassung soll möglichst keine Formeln enthalten.

Appendix

Hierher gehören zur Dokumentation Tabellen, Messprotokolle, Rechnerprotokolle, Konstruktionszeichnungen, kurze Programmausdrucke und Ähnliches.

Bibliography

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. "The Semantic Web". In: *Scientific American* 284.5 (May 2001), pp. 34–43. URL: http://www.sciam.com/article.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21 (cit. on p. 1).

[HPZC07] Bin He, Mitesh Patel, Zhen Zhang, and Kevin Chen-Chuan Chang. "Accessing the deep web." In: *Commun. ACM* 50.5 (2007), pp. 94–101. URL: http://dblp.uni-trier.de/db/journals/cacm/cacm50.html#HePZC07 (cit. on p. 1).

[SGH+15] Martin G. Skjæveland, Martin Giese, Dag Hovland, Espen H. Lian, and Arild Waaler. "Engineering ontology-based access to real-world data sources". In: *Web Semantics: Science, Services and Agents on the World Wide Web* 33 (2015), pp. 112–140 (cit. on pp. 1, 3, 4).

[W3C09] W3C XML Core Working Group. *XML Base (Second Edition)*. https://www.w3.org/TR/xmlbase/. [Accessed: 2016-04-02]. 2009 (cit. on p. 4).

[W3C12] W3C OWL Working Group. *OWL 2 Web Ontology Language, Document Overview (Second Edition)*. https://www.w3.org/TR/owl2-overview/. [Accessed: 2016-04-02]. 2012 (cit. on p. 4).

[W3C12] TODO. *A Direct Mapping of Relational Data to RDF*. https://www.w3.org/TR/rdb-direct-mapping/. [Accessed: 2016-04-06]. 2012 (cit. on p. 1).