

UiO : **Department of Informatics**
University of Oslo

An R2RML Mapping Management API in Java

Making an API Independent of its Dependencies

Marius Strandhaug
Master's Thesis Spring 2014



Abstract

When developing an Application Programming Interface (API), there is often a need to use other libraries to implement some of the functionality. However, using an external library will make the API dependent on it. There may be several libraries that implement the needed functionality. A problem arises if the user needs to use a different library than what the API supports. The user should be able to choose which underlying library the API will use. To achieve this, the API will have to be made in a generic way, so that it can work with any library that implements the needed functionality.

This thesis discusses approaches that can make an API independent of its dependencies. We will see that a combination of several approaches are needed in order to achieve the best result. The approaches are tested on an R2RML mapping management API, written in Java. The API was made for the Optique project, led by the LogID group at the University of Oslo. The thesis will first give an introduction to Ontology Based Data Access (OBDA), with a focus on mappings. It will then discuss how to design an independent API. Towards the end, it will discuss the design and implementation of the R2RML API in detail.

Acknowledgements

First of all, I would like to thank my supervisor Martin Giese for all the help with this thesis. I also want to thank Arild Waaler and the members of Optique for letting me be a part of the project. Thanks to the LogID research group for many interesting seminars.

Thanks to Timea Bagosi, who wrote the code for parsing R2RML mappings, as well as being of great help during the development of the API. Also, many thanks to Marco Ruzzi and Riccardo Mancini for writing unit tests for the API.

Finally, I want to thank my family and friends for all the motivation and support.

Contents

1	Introduction	1
1.1	Motivation and Approach	1
1.2	Contributions	2
2	Ontology Based Data Access	3
2.1	What is an Ontology?	3
2.2	Description Logics and OWL	4
3	Optique	7
4	Mapping Relational Databases to RDF	11
4.1	D2RQ	14
5	R2RML	17
5.1	Background	18
5.2	Mapping a Database to RDF with R2RML	20
5.3	The R2RML Data Model	21
5.3.1	Triples Map	21
5.3.2	Logical Table	21
5.3.3	Predicate Object Map	22
5.3.4	Term Maps	22
5.3.5	Referencing Object Map	24
5.4	Advantages and Disadvantages of R2RML	26
5.5	An example mapping	27
6	API Design	29
6.1	API Requirements	30
7	Design Patterns	33
7.1	Singleton	34
7.2	Abstract Factory Pattern	36
8	Library Independence	39
8.1	Dependency Injection	41
8.1.1	Library Configuration	43
8.2	Generics	46
8.3	Class Objects	48
8.4	C++ Templates	50

9 R2RML API Design	53
9.1 Requirements	53
9.2 API Architecture	56
10 Supported Libraries	63
10.1 Jena	63
10.2 Sesame	64
10.3 OWL API	65
11 R2RML API Implementation	69
11.1 Tools	69
11.1.1 Eclipse	69
11.1.2 Subversion	69
11.2 Implementation of the R2RML API	70
11.3 Code examples	71
12 Evaluation and Conclusion	75
12.1 Conclusion	76
12.2 Future Work	77
A R2RML API Code and Documentation	79

List of Figures

3.1	An overview of the Optique architecture[30]	9
4.1	An example relational database.	13
5.1	The process of mapping an RDB to RDF.	20
5.2	An overview of the R2RML data model.	21
5.3	An example relational database.	27
7.1	An example using the Abstract Factory Pattern.	37
8.1	<code>GUIHandler</code> is dependent on <code>WindowsFrame</code> and <code>WindowsButton</code> .	41
8.2	<code>GUIHandler</code> is dependent on the abstractions <code>Frame</code> and <code>Button</code> .	42
8.3	Overview of dependencies using a library configuration.	44
9.1	Class diagram of the <code>LogicalTable</code> .	57
9.2	Class diagram of the central access point to the API.	59
9.3	Illustration of how the bridge module makes the R2RML API work with an external library.	61
9.4	The bridge module is dependent on both the core module and the library.	61
10.1	The structure of RDF statements in Jena.	64
10.2	The structure of RDF statements in OpenRDF Sesame.	65
10.3	The structure of RDF statements in OWL API.	66

Listings

4.1	The direct mapping of the relational database in Figure 4.1	13
4.2	An example D2RQ mapping that can be used on the database in Figure 4.1 on page 13.	15
5.1	A R2RML mapping that generates the graph in Listing 5.2 on page 28.	27
5.2	The resulting RDF graph given by applying the mapping in Listing 5.1 on page 27 to the database in Figure 5.3.	28
7.1	A simple (but unsafe) singleton implementation.	36
7.2	A modern singleton implementation.	36
7.3	Usage of the Abstract Factory Pattern.	38
8.1	A simple implementation of constructor injection.	42
8.2	An implementation of interface injection.	43
8.3	A simplified example of how the library configuration (<code>LibConfig</code>) can be used.	45
8.4	The Java <code>List</code> interface.	46
8.5	An example showing how the type parameters propagate to other classes.	47
8.6	Usage of an API with Java Generics.	48
8.7	Example of a set-method using class objects.	49
8.8	Usage of an API with class objects.	50
8.9	An example of a class template.	51
8.10	Extended class template.	52
11.1	The mapping used in the code examples.	71
11.2	How to create a Jena mapping with the R2RML API.	72
11.3	How to import a Jena mapping with the R2RML API.	72
11.4	How to serialize a Jena mapping with the R2RML API.	73
11.5	An example implementation of the <code>LibConfiguration</code> inter- face.	74
11.6	Using a custom RDF library with the R2RML API.	74

Chapter 1

Introduction

1.1 Motivation and Approach

Being able to access *Big Data* in an effective manner, is a great challenge for many companies. The size and complexity of such data makes it very hard to access the relevant data. E.g. an oil company like Statoil generates several terabytes of data every day. Engineers need to use this data, but querying it requires detailed knowledge about the data sources. The engineers often have to ask an IT expert to construct queries for them. Ideally, the engineers should be able to query the data themselves.

Ontology Based Data Access (OBDA) is a data access paradigm that has the potential to solve this problem. It uses an ontology as a semantic layer on top of the underlying data sources. The Optique project aims to bring a paradigm shift to data accessing using an OBDA approach.

An essential part of OBDA is the *mappings*. Relational databases are widely used to store data, but in order to access these through an ontology, it has to be mapped to another data model called *RDF*. This can be done with a mapping language such as *R2RML*.

An R2RML mapping specifies the relation between tables in the databases, and the RDF. In order to map a database to RDF, several of these mappings are needed. The mappings have to be managed by an IT expert in order to keep them up to date with the needs of the end-users. However, the mappings are serialized in RDF, so manipulating them manually is not very convenient. Optique therefore needed an R2RML mapping management API in Java. This API should provide an abstraction of the R2RML data model, above its RDF serialization.

During the development, we decided to make the API available to the public, in the hope of establishing a de-facto standard Java implementation of R2RML. To do this, the API has to be made in a way that makes it appealing for as many developers as possible. We therefore wanted to make the API independent of any specific external library. The first Optique version was

dependent of the OpenRDF Sesame API to handle RDF functionality. We wanted the API to work with any RDF library that a user would want to use.

There are several potential approaches for making an API independent of an underlying library (see Chapter 8 on page 39). There may be a combination of several approaches that is needed to get the best result. The approach should not increase the complexity of the API too much. Ideally, using the independent API should not be very different from using the dependent version of the API.

1.2 Contributions

This thesis will discuss how to make an API that is independent of its dependencies. The goal is to establish an approach that can be used in a variety of situations. The approach will be used on the R2RML API. These are the contributions that this thesis will make:

- Design an R2RML mapping management API in Java.
- Discuss possibilities for making an API independent of its dependencies.
- Apply the approach to the R2RML API.

The R2RML API will provide programmatical access to R2RML mappings with Java. It will give an abstraction of the R2RML mappings language above the level of its RDF serialization. It will be independent of any specific library for handling RDF. The user will be able to choose which library the API will use, or even be able to extend the support for another library.

Chapter 2

Ontology Based Data Access

Ontology Based Data Access (OBDA) is a data access paradigm where ontologies are used to classify data, verify data and more. The actual data can be stored in relational databases, in RDF triple stores, or in some other way. The ontology adds a semantic layer on top of the underlying data storage. It provides a common vocabulary that is familiar for the end-users. The vocabulary abstracts away the details of the data sources. The end-user can then query the system using this vocabulary, without thinking about the structure and syntax of the underlying data storage. The OBDA system will automatically transform the queries over the vocabulary into queries over the data sources.

To be able to rewrite the queries over the ontology to queries over the data sources, a set of mappings is needed. The mappings define the relationship between the ontology and the underlying data sources. Mappings are made using a declarative mapping language. This will be thoroughly described in Chapter 4 on page 11.

2.1 What is an Ontology?

The term *ontology* was originally used in philosophy. It comes from the Greek word *ontos*, which means “being” or “that which is” [37]. In philosophy, ontology is the study of being and existence. It deals with questions about what entities can exist, and how entities can be grouped and related within a hierarchy [35].

In computer science, an ontology describes knowledge. The ontology defines concepts, properties, and relations between those concepts and properties. It describes what kind of things that can exist and not exist, and how the things relate to each other. For example, we can say that a child is a person with age less than 18, or that no one can be both a person and a horse.

The ontology formalizes the vocabulary that people would use to talk about a domain. It is created completely independent of how the data is stored.

Querying the data can therefore be done using the vocabulary that the users are familiar with. The query to the ontology must be rewritten into a query over the data sources in order to get the results.

There are several existing ontologies that are widely used. An example is the *Friend of a Friend* (FOAF) [14] ontology. FOAF defines a common vocabulary to describe people, groups and organizations, and relationships between them. Another example is the *SNOMED-CT* ontology, which is a large ontology that describes medical terms. SNOMED-CT actually contains over 311,000 active concepts [1].

An ontology is described using an ontology language. The components of an ontology language are quite similar in every language. Some of the most important components in an ontology language are the following [36]:

- Individuals
- Classes
- Attributes
- Relations

Individuals are the concrete objects in the world. This could be a concrete person, car, plant, a specific case of a disease (in SNOMED-CT) or something else.

Classes are what individuals can be grouped in. A class is interpreted as a set of individuals. Examples of classes are persons, cars, plants, diseases etc. Classes can also be more complex, for example adults, which can be defined as all persons over the age of 18.

Attributes (or roles) are used to describe individuals or classes. A person may have a name, or have another person as a father. A class of animals may be endangered. Using attributes on a class may cause it to be treated both as a class and as an individual. There are often two kinds of attributes, object attributes and data attributes. Object attributes are used to connect individuals to other individuals, while data attributes connects an individual to some data (like a name).

Relations describe how the classes, individuals and properties relate to each other. A class may for example be a subclass of another class. The class of women is a subclass of person, because all women are persons. Individuals may be a member of a class, and a property may be a subproperty of another property.

2.2 Description Logics and OWL

The far most common ontology language, is *OWL*. OWL contains several constructs to describe different kinds of axioms. OWL is based on a group of logics called *Description Logic* (DL), which is a formal logic used

for knowledge representation. OWL has several profiles that is based on different kinds of description logics. The differences in these profiles are the logical constructs they support. Some have a low expressivity and a low complexity, while others have a high expressivity and a high complexity. This makes the profiles suitable for different scenarios. For example, OWL QL is a profile that is suitable for accessing data stored in relational databases.

OWL QL is based on the description logic $DL\text{-}Lite_R$ (see [24], Section 3). This description logic has a quite low expressivity, but it is very efficient for query answering. $DL\text{-}Lite_R$ has a property called first-order rewritability. This means that a query made over a $DL\text{-}Lite_R$ ontology, can be rewritten to a first-order query (which has a very good data complexity¹). These queries can then be translated to SQL, which is used on the relational databases. This makes the OWL QL profile very suitable for an OBDA approach.

An ontology consists of a *TBox* and an *ABox* (see Table 2.1 on the next page for an example). The *TBox* contains terminological axioms that describe the domain of the data. These axioms describe relationships between different concepts and roles. For example that a man is a person ($Man \sqsubseteq Person$), or that a mother is a woman that has a child ($Mother \equiv Woman \sqcap \exists hasChild$). How complex these axioms can be, is decided by the description logic that is used. $DL\text{-}Lite_R$ allows all axioms in Table 2.1, except the one that defines *Mother*². The *ABox* contains assertions that describe the actual data. For example that Maria is a woman ($Woman(Maria)$), and that Jesus is the child of Maria ($hasChild(Maria, Jesus)$).

To be able to answer queries to this ontology, we need to take into account both the *ABox* and the *TBox*. We want the query answering to be complete with regards to the ontology. That is, if we make a query, the answer must both check the *ABox* for the right assertions, but also check the *TBox* for ways to get more answers. For example, if the query asks for all mothers ($Mother(x)$), we will first need to check the *ABox* to see if there are any assertions stating that someone is a mother. In our example, there are none such assertions. However, by looking at the *TBox*, we can see that a mother is defined as a woman with a child. The query therefore needs to be reformulated, in order to use the *TBox* as well. In this case, we need to find everyone who is a mother, or those who are both women and have a child. This is called the *perfect reformulation* [28, p. 147]. Answering this query on the *ABox* only, is the same as answering the original query on both the *TBox* and the *ABox*.

¹It is in the AC_0 complexity class.

² $DL\text{-}Lite_R$ does not allow intersection in *equivalent class axioms*.

Natural Language	Logic
<i>Ontology (TBox)</i>	
A man is a person	$Man \sqsubseteq Person$
A woman is a person	$Woman \sqsubseteq Person$
A mother is a woman that has a child	$Mother \equiv Woman \sqcap \exists hasChild$
<i>Data (ABox)</i>	
Maria is a woman	$Woman(Maria)$
Maria has the child Jesus	$hasChild(Maria, Jesus)$

Table 2.1: An example ontology and data

Chapter 3

Optique

The Optique project aims to bring a paradigm shift to data accessing, using an OBDA approach [26]. Current state-of-the-art OBDA approaches have limitations when nontechnical end-users want to access the large amounts of data. Some of the limitations are (quoted from page 212 in [16]):

1. The *usability* is hampered by the need to use a formal query language that makes it difficult for end-users to formulate queries, even if the vocabulary is familiar.
2. The *prerequisites* of OBDA, namely the ontology and mappings, are in practice expensive to obtain.
3. The *scope* of existing systems is too narrow: they lack many features that are vital for applications.
4. The *efficiency* of both the translation process and the execution of the resulting queries is too low.

Classical OBDA approaches have several problems with how the end-users are accessing data. Usually, the end-user gets access to the data through specialized applications that retrieves the data through a predefined pool of queries. These queries are usually specialized toward some specific data source with a specific structure. This limits the complexity of the queries that the end-user can make, but the time it takes to get an answer is kept low. The situation can be illustrated like this [29]:

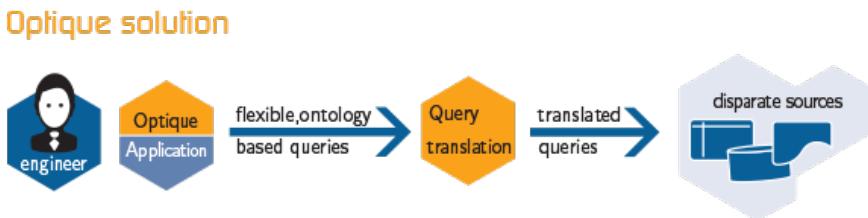
Simple case:



Problems arise when the user needs to make more complex queries. This may require in-depth knowledge of the domain, as well as knowledge of how the data is stored. In this case, the end-user would have to ask an IT expert for help to formulate the query. This can take a very long time to complete, depending on how much time the IT expert has available. This situation can be illustrated like this [29]:



The time to get the answer to a complex query like this, is increased from minutes to days. This is a major bottleneck. The goal is therefore to enable the end-users to query the data themselves, without the help of the IT experts [16, p. 207]. Optique's solution to this problem, can be illustrated as follows [29]:



With the Optique solution, the IT experts can instead focus on the ontology and mapping management. New vocabulary may be added to the ontology if it is needed by the user queries, and mappings must be added when there are new data sources. Building an ontology from scratch may be very expensive. It will therefore be built using a bootstrapping mechanism that generates an initial ontology. The ontology can then be extended to fit the needs of the end-users. The mappings can initially be constructed based on, for instance, the database schemas. However, these mappings need to be adjusted and new mappings may need to be added. Some mappings may even become obsolete if the ontology or the data sources are changed. [16, p. 219-220]

In order for end-users to construct their own queries, they need some kind of user interface. Constructing custom SPARQL or SQL queries are not something we can assume that the typical end-user knows how to do. The user interface must therefore simplify this process, so that the end-user is able to create these queries anyway. This is one of the areas that the Optique project will explore. Figure 3.1 on the facing page gives an overview of the Optique architecture.

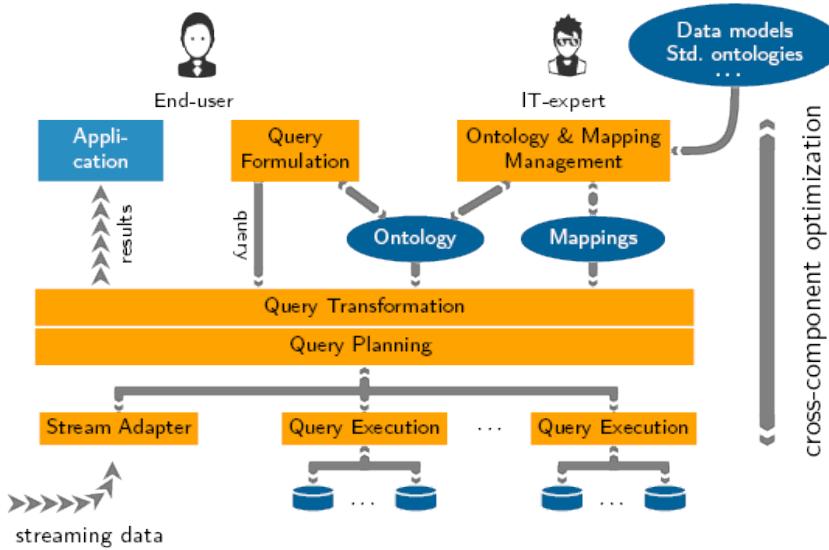


Figure 3.1: An overview of the Optique architecture[30]

Statoil and Siemens are two use-case providers for the project. Statoil's use-case provides a very large dataset that contains geological data. The issues here regard mainly scalability, as well as how to handle geospatial data. Query answering on such large datasets need to be efficient. Siemens' use-case contains streamed sensor data from turbines and other equipment. The growth rate of such data is very large. Because the data is streamed, the system needs to be able to handle time-stamped data. The issues here is therefore related to time and streams.

Another issue that must be taken care of, is distributed query execution. The data is very often distributed across several heterogeneous data sources. In both of the use-cases, the data is distributed over several sources. A single query might require data from several sources in order to give the correct answer. How to do this effectively with such large amounts of data, is something that needs to be investigated. The queries need to be optimized to run on such a highly distributed system. The lower part of Figure 3.1 illustrates the distributed data sources.

Chapter 4

Mapping Relational Databases to RDF

The majority of today's data is stored in relational databases. To make this data available on the Semantic Web, or to be able to access the data using an OBDA approach, it needs to be mapped to RDF. Mapping the relational data to RDF can be done in several ways. A simple approach is to use a direct mapping that maps the data directly, without any possibility to customize the RDF. Another approach is to use a mapping language that allows the user to specify the relation between the two data models. This makes it possible to customize the output RDF.

There are several ways to access the mapped RDF data. One way is to materialize the data and store it in, for example, a triple store. This can be used to store a dump of the database as RDF, but it is not very practical if the RDF has to be kept up to date with the database. It will also be impractical if the database is very large. What is often done instead, is to only create an RDF view of the relational data which can be accessed through a SPARQL endpoint. The SPARQL queries can be translated to SQL, with the help of the mappings. The data is stored as it is, in the database. This means that the database can be updated normally, because the mappings are evaluated at query-time.

Mapping languages can be classified into four categories [18, p. 29]. These are: Direct mappings, read-only general purpose mappings, read-write general purpose mappings and special purpose mappings. The special purpose mapping languages are made for specific use-case scenarios. An example is the mapping language called Triplify, which is a light-weight application for publishing Linked Data from relational databases.

A direct mapping maps the relational database directly to RDF, without any user interaction. This means that the RDF will reflect the exact data model of the relational data, rather than the domain of the data. A direct mapping usually works by mapping each table to a class. Each row in the table will be mapped to an individual that will be a member of the table's

class. Each column will be mapped to a property. Foreign keys will be mapped with a property that links one individual to another. The range of other properties will be literals.

The direct mapping of the relational data is usually not what is needed. It rather provides a basis for defining and comparing more intricate transformations (see [8], Section 1). It has several downsides. For example, many-to-many relations in a relational database are usually modelled by having a table that links entities of one table to entities of another table. This causes the direct mapping to create an additional class for this table, even though it is easily expressed in RDF with a simple property. Another disadvantage is that the direct mapping will not reuse the vocabulary of any existing ontologies. All URIs are generated based on the schema and data of the database.

Listing 4.1 on the facing page gives an example of a direct mapping. As can be seen, the structure of the generated RDF directly reflects the structure of the database. A better mapping would, for example, use the FOAF [14] vocabulary to describe persons. It might also remove the `<Person#OwnsCar>` property, and only keep `<Person#ref-OwnsCar>`. The ID property of the persons can also be removed, as it was meant to be used only within the relational database. The IDs were used to construct the URIs of the persons.

A general purpose mapping language is much more useful than a direct mapping. A language like this works by specifying a mapping that describes the relation between the relational database and the RDF. A read-only general purpose mapping language is usually quite expressive. However, this may make the language harder to understand and learn. The high expressiveness also increases the complexity of the mappings, which in turn makes it hard to add write support. Examples of read-only general purpose mapping languages include R2RML and D2RQ.

Read-write general purpose mapping languages are usually less expressive than the read-only mappings. This is mainly due to the *view update problem*, which makes it hard to update the database based on an update to a view [34]. The reason is that several different updates to the database may cause the same change to the view. In SQL, this is solved by adding restrictions to updateable views. In order to have general support for write access, it is therefore necessary to add restrictions to the mapping language. An example of a read-write general purpose mapping language is R3M.

Section 4.1, gives an overview of how D2RQ works, while Chapter 5 describes the mapping language R2RML in great detail.

Car		Person		
RegNum	CarModel	ID	Name	OwnsCar
CE54321	Honda Civic	1	John	DK12345
DK12345	Tesla Model S	2	Ann	NULL

Figure 4.1: An example relational database.

```

1 @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
2 @base: <http://www.example.org/> .
3
4 <Person/ID=1> rdf:type <Person> ;
5     <Person#ID> 1 ;
6     <Person#name> "John" ;
7     <Person#OwnsCar> "DK12345" ;
8     <Person#ref-OwnsCar> <Car/reg=DK12345> .
9 <Person/ID=2> rdf:type <Person> ;
10    <Person#ID> 2 ;
11    <Person#name> "Ann" .
12
13 <Car/reg=DK12345> rdf:type <Car> ;
14    <Car#reg> "DK12345" ;
15    <Car#CarModel> "Tesla Model S" .
16 <Car/reg=CE54321> rdf:type <Car> ;
17    <Car#reg> "CE54321" ;
18    <Car#CarModel> "Honda Civic" .

```

Listing 4.1: The direct mapping of the relational database in Figure 4.1.

4.1 D2RQ

D2RQ is a language for mapping relational databases to RDF. It is a part of the D2RQ system, which makes it possible to view relational databases as virtual, read-only RDF graphs. The D2RQ project was founded by Christian Bizer, and is currently maintained by Richard Cyganiak [33]. D2RQ aimed to fill the gap between linked open data (RDF) and relational databases. People wanted to make the data they already had in relational databases available as linked open data.

The D2RQ language is used to make mappings that describe the relation between the database and the RDF. The D2RQ mapping itself is written as RDF. A mapping consists of class maps and property bridges. A class map represents a class from an OWL ontology or RDFS schema, and specifies how the instances of that class can be identified in the database (see [32], Section 5). A property bridge is used to add properties to objects from a class map (see [32], Section 6).

In a class map, the property `d2rq:class` specifies the URI of the class. The instances of the class can be specified in a number of ways. The most normal way is to use a URI pattern that is instantiated by inserting values from the database into the pattern. A URI pattern consists of a skeleton of a URI, and column names from a table in the database. If the values in the database already are valid URIs, they can be used directly by using the `d2rq:uriColumn` property. It is also possible to generate blank nodes.

Property bridges are used to add properties to objects in a class. It has a property called `d2rq:belongsToClassMap` that specifies which class the property is for. The instances of the specified class will be the subjects of the generated triples. The property `d2rq:property` specifies which property that will be used. It is also possible to use `d2rq:dynamicProperty` to generate the property URI at runtime (using a pattern). If multiple properties are specified, then one triple for each property will be generated.

There are several ways to specify what the objects of the triples will be. The easiest way is to use `d2rq:column`, which will use the values of the given database column as literal values in the generated triples. If the values of the column are URIs, it is possible to use the `d2rq:uriColumn` property. It is also possible to generate literals and URIs using patterns and SQL expressions. Adding datatypes or language tags to literal values can be done by using, respectively, `d2rq:datatype` or `d2rq:language`.

The property `d2rq:refersToClassMap` makes it possible to use the subjects of another class map, as objects. This is useful to express the foreign keys in the database. If the class maps get their values from different tables, then a join has to be specified (`d2rq:join`). A join is specified by two columns in different tables that should be equal (for an example, see line 20 in Listing 4.2 on the next page).

The example mapping in Listing 4.2, maps a portion of the database shown

```

1 @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
2 @prefix foaf: <http://xmlns.com/foaf/0.1/> .
3 @prefix ex: <http://www.example.org/> .
4
5 ex:CarMap a d2rq:ClassMap ;
6   d2rq:uriPattern "http://www.example.org/reg=@@Car.reg@@" ;
7   d2rq:class ex:Car ;
8   # Specifies a database connection (not covered here)
9   d2rq:dataStorage ex:Database1 .
10
11 ex:PersonMap a d2rq:ClassMap ;
12   d2rq:uriPattern "http://www.example.org/ID=@@Person.ID@@" ;
13   d2rq:class foaf:Person ;
14   d2rq:dataStorage ex:Database1 .
15
16 ex:CarOwner a d2rq:PropertyBridge ;
17   d2rq:belongsToClassMap ex:PersonMap ;
18   d2rq:property ex:ownsCar ;
19   d2rq:refersToClassMap ex:CarMap ;
20   d2rq:join "Person.OwnsCar => Car.RegNum" .

```

Listing 4.2: An example D2RQ mapping that can be used on the database in Figure 4.1 on page 13.

in Figure 4.1 on page 13, to RDF. It contains the class maps `ex:CarMap` and `ex:PersonMap`. The instances of these classes come from, respectively, the `Car` and `Person` tables. Each instance will either have the type `ex:Car`, or `ex:Person`. The URIs of the instances are generated by a URI pattern. For example, a person with an ID set to 1, will get the URI: `http://www.example.org/ID=1`. The property bridge `ex:CarOwner` relates persons to their cars. In the database, the table `Person` has a foreign key to the table `Car`. It is therefore possible to use the `d2rq:refersToClassMap` property to use instances from the `ex:CarMap` class map as objects in the generated triples. To do this, we also need a join condition. The join condition says that the `OwesCar` column in the `Person` table, must be equal to the `RegNum` column in the `Car` table.

Chapter 5

R2RML

R2RML (RDB to RDF Mapping Language) is a mapping language that maps relational databases to RDF triples [12]. An R2RML mapping specifies the relationship between the database and the RDF triples. A mapping consists of a triples map, that with the help of its subparts, can specify the RDF triples. R2RML offers many ways to specify RDF triples. This makes it useful in many different situations. This is in contrast to direct mappings, which does not allow any control of the resulting RDF graph.

R2RML allows the user to specify mappings that are used to create read-only RDF views of relational data. The mappings themselves are also serialized as RDF. R2RML is independent of any application that may use the mappings. This means that an application that queries an RDF view, does not need to care that the RDF view was created by an R2RML mapping. For the application, querying an RDF view created by an R2RML mapping will be exactly the same as if the data was stored directly as RDF.

Listing 5.1 on page 27 gives an example of an R2RML mapping in Turtle syntax. Using this mapping on the database in Figure 5.3, will generate the graph in Listing 5.2. As can be seen, a lot more can be done with an R2RML mapping than what is possible with a simple direct mapping. For example, the URIs created by the direct mapping are based only on the database schema and its values. R2RML makes it easy to reuse existing vocabularies, such as the FOAF [14] vocabulary about persons. The structure of the RDF graph can also get much better when using an R2RML mapping.

The upcoming sections will give some background for the R2RML mapping language, before going into detail about the structure of a mapping and how it works. The last section of this chapter will describe some of the advantages and disadvantages of using R2RML.

5.1 Background

R2RML was created by W3C’s RDB2RDF Working Group. In October 2007, the W3C formed an incubator group to explore the area of RDF access to relational databases. This group made a survey [31] of the state of the art techniques, tools and applications for mapping relational databases to RDF. This further led to a group report [22], that recommended the initiation of a working group that was going to standardize a mapping language. This was the start of the RDB2RDF Working Group.

The group report defined some use-cases for the mapping language, as well as giving the working group a starting point for the development. They were going to start with the mapping languages developed by the D2RQ and the Virtuoso¹ efforts. This is what links D2RQ to R2RML. R2RML builds on the lessons learned from D2RQ. Richard Cyganiak, who worked on the D2RQ mapping language, was also central in the development of R2RML.

Virtuoso is another mapping system that provides a SPARQL-based language that is used to create *quad map patterns*. A quad map pattern defines a mapping between RDB columns and RDF triples. The tables of the database are mapped to RDFS classes, and the columns to predicates. It also takes into consideration whether the columns are part of a primary key or a foreign key.

The people behind D2RQ pointed out some of the problems that needed to be solved [10]. For example, the performance of D2RQ varies depending on the access method of the generated RDF view. They also experienced that people often needed very weird mappings for databases that did not conform to accepted database design principles. This requires a more expressive mapping language. R2RML also supports named graphs, while D2RQ does not.

Based on the survey and the report from the incubator group, the working group published a document that defined the requirements of R2RML, from some use cases [9]. The following list contains a shortened form of some of the requirements:

1. A minimal configuration must provide a (virtual) RDF graph representing the attributes and relationships between tuples in the relational database.
2. The R2RML language must express transformations of the relational graph to produce the (virtual) RDF graph.
3. The R2RML language must allow for a mechanism to generate globally unique identifiers for database entities.
4. The R2RML language must allow the mapping specification to have sufficient information to enable transformation of SPARQL queries

¹See <http://virtuoso.openlinksw.com/>.

over the RDF view into efficient SQL queries over the relational database.

5. The R2RML language must allow the mapping to have sufficient information to provide a dump of the entire RDF graph (materialized RDF graph).
6. Relational databases typically use three tables to represent Many-to-Many relationships. This requirement is to allow such relationships to be mapped using direct links between the entities.

The first requirement concerns direct mappings (see Section 4 on page 11). The minimal configuration will be a direct mapping from the database to an RDF graph. This can be used to let the database schema determine the ontology of the RDF view.

It is good practice to reuse existing ontologies. The second requirement says that it must be possible to do transformations on the relational graph to produce the RDF graph. This makes it possible for the data to fit into an existing ontology. Consider a database with a table of teachers. One of the columns in this table specifies the subjects that they are teaching. If an existing ontology defines classes for the different kinds of teachers (MathTeacher, PhysicsTeacher etc.), it should be possible to map the teachers into each of these classes, based on the value of the subject-column.

The third requirement is very important. A relational database is quite isolated by nature. This means that, for example, a primary key of a person in one database may collide with the primary key of another person in another database. It must therefore be possible to create globally unique identifiers for the entities in the database.

The fourth and fifth requirements concern how the RDF view can be accessed. In order to access the RDF views with SPARQL, it needs to translate the SPARQL query to a SQL query over the relational database. The fourth requirement states that the mapping must contain enough information for this to be possible. The fifth requirement states that it must be possible to materialize the RDF view.

The sixth requirement concerns many-to-many relationships. Usually in relational databases, such relationships are expressed by having a table that links one entity to another. However, RDF does not require such constructs to express many-to-many relationships. Such tables should therefore be mapped to properties, instead of classes.

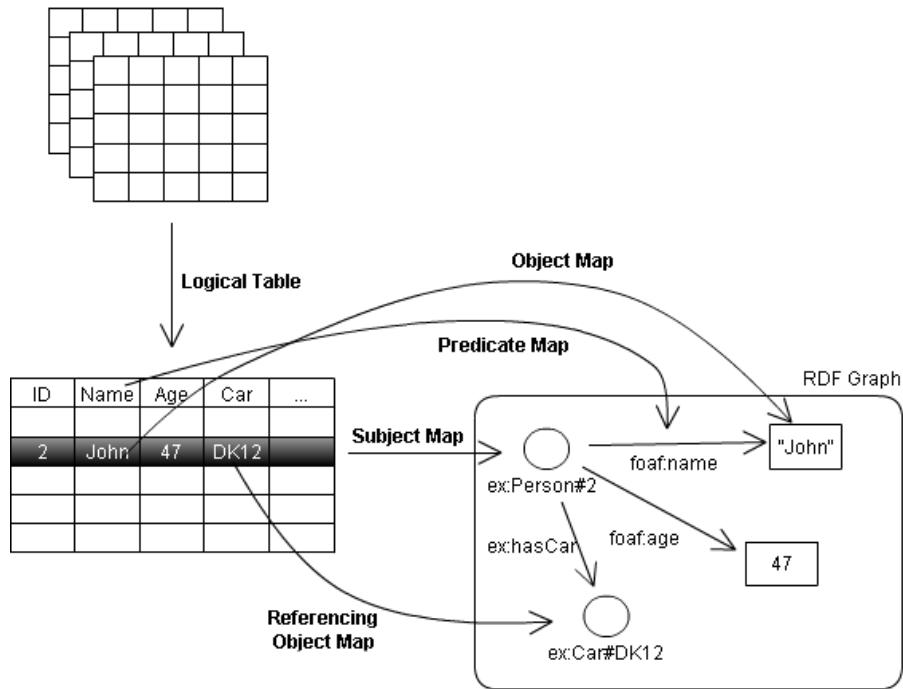


Figure 5.1: The process of mapping an RDB to RDF.

5.2 Mapping a Database to RDF with R2RML

Figure 5.1 illustrates the process of mapping a database to RDF using R2RML. Starting with a set of database tables, the *logical table* selects one of the tables (or creates a view) which will be used by the mapping. The *subject map* specifies the mapping between rows in the table and resource nodes in the RDF graph. In this case, it maps the second row to the resource node `ex:Person#2`.

A *predicate map* maps a column of the table to a predicate in the RDF graph. There may be several predicate maps in order to specify more predicates. In this case, a predicate map maps the columns `Name`, `Age` and `Car` to, respectively, the predicates `foaf:name`, `foaf:age` and `ex:hasCar`.

An *object map* maps the values of the database to objects in the RDF graph. A *referencing object map* is useful to map foreign key relations. Two normal object maps and one referencing object map are used in Figure 5.1. The two normal object maps, maps the values `John` and `47` to literal values in the RDF graph. The referencing object map maps the value `DK12` to the resource node `ex:Car#DK12`. This resource node must have been specified by a subject map of another mapping. Which objects that are connected to which predicates, are decided by *predicate object maps*.

Listing 5.1 on page 27 contains a complete mapping that maps the database in Figure 5.3 to RDF. This mapping will be used as an example for the rest of this chapter.

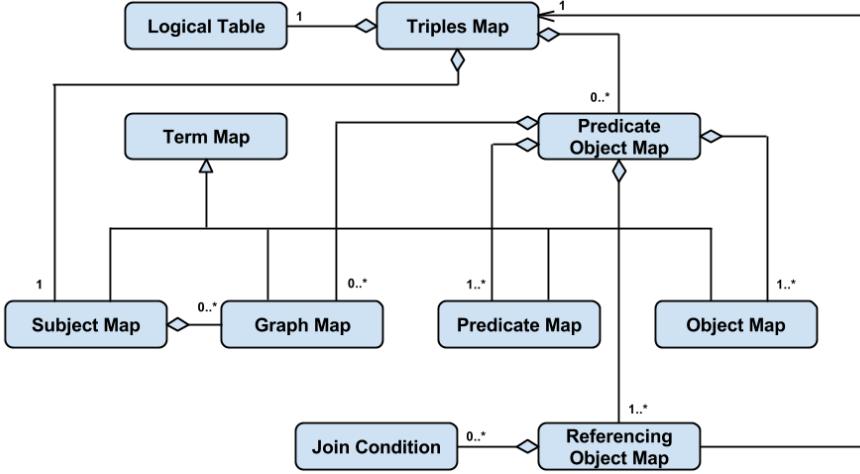


Figure 5.2: An overview of the R2RML data model.

5.3 The R2RML Data Model

An R2RML mapping consists of several parts. As can be seen in Figure 5.2, there is a main part called the triples map, which contains a logical table, a subject map and a number of predicate object maps. This section will describe all the different parts of the R2RML data model. Listing 5.1 on page 27, will be used as an example throughout this section.

5.3.1 Triples Map

The *triples map* is the main part of an R2RML mapping (see [12], Section 6). A triples map specifies RDF triples corresponding to a *logical table*. A logical table can be, for example, a table in a database. The triples map also contains a *subject map* and a number of *predicate object maps*, which specifies how the triples should be. There are two triples maps in the example (Listing 5.1 on page 27), `ex:CarMap` and `ex:PersonMap`. The `ex:CarMap` maps the cars in the database to RDF, while `ex:PersonMap` maps the persons.

5.3.2 Logical Table

The logical table is the SQL table that the RDF triples will be generated from (see [12], Section 5). The logical table can either be an SQL table, an SQL view, or an R2RML view. An R2RML view is an SQL query similar to a normal SQL view. However, an R2RML view allows the mapping to specify a view without having to add it to the actual database. A logical table has an effective SQL query that will produce the contents of the logical table when executed on the input database. The effective SQL query of an

R2RML view, is basically its SQL query. For an SQL table or view, it is the query:

```
SELECT * FROM {table};
```

In the example, the logical table of `ex:CarMap` has the table name `Car`. This means that the logical table of `ex:CarMap` will be identical to the table `Car` in the database. The logical table of `ex:CarMap` is given by the following SQL query:

```
SELECT * FROM Car;
```

5.3.3 Predicate Object Map

A predicate object map specifies predicates and objects of the RDF triples that will be generated (see [12], Section 6.3). A predicate object map has at least one *predicate map* and one *object map*. A predicate map specifies the predicates while an object map specifies objects. An object map may either be a normal object map, which specifies objects directly based on the values in the database, or a *referencing object map* which is used to map foreign keys (see Section 5.3.4 and 5.3.5).

In the example, `ex:CarMap` and `ex:PersonMap` contains two predicate object maps each. The first predicate object map of `ex:CarMap` (line 13 to 16) is used to add registration numbers to the cars.

5.3.4 Term Maps

A term map specifies RDF terms from the logical table (see [12], Section 7). There are four kinds of term maps, one for each part of an RDF triple: Subject maps, predicate maps, object maps and graph maps. Each of these are used in different parts of the mapping.

A term map can either be constant-valued, column-valued or template-valued. If it is constant-valued, all the terms will have the same specified value. If it is column-valued, the values in the specified column in the logical table will be used to create the terms. If it is template-valued, the terms will be specified based on the given template. A template is a string that can contain several column names. A template is useful to create unique URIs based on the values of one or more columns. For example, at line 10 in the example, the following template is specified:

```
"http://www.example.org/car/{RegNum}"
```

Creating URIs from this template are done by replacing `{RegNum}` with values from the `RegNum` column in the logical table.

The term map can also have a specified term type. The term type determines what kind of RDF term it is. The possible term types are URIs, blank nodes and literals. The term type can only be set for column-valued and template-valued term maps. The type of the RDF term specified by a constant-valued term map, is determined directly by the given constant value. For example, if the constant-value is set to `ex:hasReg` (line 14 in the example), the term type will be set to URI.

Subject Map

A subject map specifies subject terms from the logical table (see [12], Section 6.1). The subject map may have a class that each specified subject will be a member of. It may also have a *graph map*. The term type of a subject map can either be URI or blank node, as literals are illegal in the subject position of an RDF triple. It is very common for a subject map to be template-valued.

In the example, `ex:CarMap` contains the following subject map:

```
ex:CarMap
  rr:subjectMap [
    rr:template "http://www.example.org/car/{RegNum}";
    rr:class ex:Car ;
  ]
```

This subject map is template-valued. The subjects are specified by the template, which will use the `RegNum` column of the logical table to create the subject URIs. In addition, the subject map has the class `ex:Car` specified. All subjects created using this subject map will be a member of this class.

Predicate Map

A predicate map specifies predicate terms for the RDF triples (see [12], Section 6.3). A predicate map can be constant-valued, column-valued or template-valued as usual term maps. In most cases, the predicate map is constant-valued. The predicates that is generated using the predicate map, will be paired with objects by the predicate object map. The term type of a predicate map must be URI.

Line 14 in the example gives an example of a constant-valued predicate map. It is specified as follows (`_:pom` is the predicate object map that it is contained in):

```
_:pom rr:predicate ex:hasReg .
```

This predicate map has the constant value `ex:hasReg`. Note that the predicate `rr:predicate` is used. This is a shortcut predicate that can be

used only for constant-valued predicate maps.

Object Map

An object map specifies the object terms for the RDF triples (see [12], Section 6.3). The term type of an object map may be URI, blank node or literal. The object map may also have a data type or a language tag if the term type is literal. As with the predicate map, the objects that the object map generate will be paired with predicates by the predicate object map. An example of an object map can be seen at line 15 in the example.

Graph Map

A graph map is a term map that allows the RDF triples to be stored in a specified named graph (see [12], Section 9). Both a subject map and a predicate object map may have a specified graph map. If the graph map is specified in the subject map, then all the triples will be stored within that graph. If it is specified in a predicate object map, then all triples generated using that predicate object map, will be stored in the graph.

The term type of a graph map must be URI. As usual, a graph map can be column-valued, template-valued or constant-valued. If the graph map is constant-valued, then all triples will be in the same graph. Column-valued or template valued graph maps can be used to store the triples in graphs named by the values in the database.

One thing that can be done with a graph map in the example, is to store all cars of the same model in the same graph. This can be done by adding a template-valued graph map to the subject map of `ex:CarMap`. The graph map can look like this:

```
[  
    rr:template "http://www.example.org/graph/{CarModel}"  
]
```

5.3.5 Referencing Object Map

A referencing object map is not a term map. However, it may be used in place of an object map. A referencing object map allows subjects from another triples map to be used as objects (see [12], Section 8). This is useful to map, for example, foreign keys in the relational database. The referencing object map has a reference to a triples map called the parent triples map. The subject map of the parent triples map is used to retrieve the objects. The triples map that the referencing object map is contained in, is called the child triples map.

If the logical table of the parent triples map is different from the logical table of the child triples map, then the logical tables have to be joined. This is done using join conditions. A join condition is specified by two columns. One from the child triples map and one from the parent triples map. Several join conditions can be specified in order to join on more than one column.

The referencing object map has a joint SQL query that is used when generating the objects. In this query, any existing join conditions are checked. The subject map from the parent triples map is used on the result of this query in order to generate the objects of the referencing object map. In this query, the child and parent queries are the effective SQL queries of the logical tables of, respectively, the child and parent triples maps.

If the referencing object map has no join conditions, then the joint SQL query is:

```
SELECT * FROM ({child-query}) AS tmp
```

If it has at least one join condition, the joint SQL query is:

```
SELECT * FROM ({child-query}) AS child,
        ({parent-query}) AS parent
WHERE child.{child-column1}=parent.{parent-column1}
    AND child.{child-column2}=parent.{parent-column2}
    AND ...
```

In the example, the second predicate object map of `ex:PersonMap` contains a referencing object map. This referencing object map is specified as follows (`_:pom` is the predicate object map that it is contained in):

```
_:pom rr:objectMap [
    a rr:RefObjectMap ;
    rr:parentTriplesMap ex:CarMap ;
    rr:joinCondition [
        rr:child "OwnsCar";
        rr:parent "RegNum";
    ] ;
]
```

As can be seen, the parent triples map is `ex:CarMap`. Because the logical tables of `ex:PersonMap` and `ex:CarMap` are different, we need to specify a join condition. The join condition says that the column `OwnsCar` from the table `Person`, must be equal to the column `RegNum` from the table `Car`. The joint SQL query of this referencing object map, will therefore be the following:

```
SELECT * FROM (SELECT * FROM Person) AS child,
        (SELECT * FROM Car) AS parent
WHERE child.OwnsCar = parent.RegNum;
```

As can be seen, the child and parent tables are defined as the effective SQL queries of the logical tables of, respectively, the `ex:PersonMap` and `ex:CarMap` triples maps.

5.4 Advantages and Disadvantages of R2RML

The article by Hert, Reif and Gall [18] compares several mapping languages, including R2RML. The comparison is based on the features of each mapping language. As can be seen in Table 1 on page 29 in the article, R2RML is the only language that supports all features except write support. This makes R2RML well suited for a variety of use cases. Keep in mind that the article was written in 2011, so some of the mapping languages may have changed since then.

R2RML provides a very expressive mapping language. Together with SQL, it is possible to map the database in any way one wants. R2RML relies quite heavily on SQL. This can be both an advantage and a disadvantage. It is an advantage, because it allows R2RML to reuse the power of SQL to produce mappings. This will push some of the mapping work from the R2RML processor to the DBMS. However, using SQL can hide much of the mapping semantics inside the SQL query, where it is not as easily accessible [18, p. 30].

R2RML does not support write access to the generated RDF views. However, this was explicitly listed as out of scope by the RDB2RDF Working Group (see [17], Section 1.2). Adding support for write access to the mapping language creates several problems. The main problem is the *view update problem* (See Section 4 on page 11), which makes it hard to update the database based on a view. It would therefore be necessary to add restrictions to the mapping language to get general write support. This is impractical for R2RML.

5.5 An example mapping

Car		Person		
RegNum	CarModel	ID	Name	OwnsCar
CE54321	Honda Civic	1	John	DK12345
DK12345	Tesla Model S	2	Ann	NULL

Figure 5.3: An example relational database.

```

1 @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
2 @prefix rr: <http://www.w3.org/ns/r2rml#> .
3 @prefix foaf: <http://xmlns.com/foaf/0.1/> .
4 @prefix ex: <http://www.example.org/> .

5
6 ex:CarMap
7     a rr:TriplesMap ;
8     rr:logicalTable [ rr:tableName "Car" ] ;
9     rr:subjectMap [
10         rr:template "http://www.example.org/car/{RegNum}" ;
11         rr:class ex:Car ;
12     ] ;
13     rr:predicateObjectMap [
14         rr:predicate ex:hasReg ;
15         rr:objectMap [ rr:tableName "RegNum" ] ;
16     ] ;
17     rr:predicateObjectMap [
18         rr:predicate ex:carModel ;
19         rr:objectMap [ rr:column "CarModel" ] ;
20     ] .
21
22 ex:PersonMap
23     a rr:TriplesMap ;
24     rr:logicalTable [ rr:tableName "Person" ] ;
25     rr:subjectMap [
26         rr:template "http://www.example.org/person/{ID}" ;
27         rr:class foaf:Person ;
28     ] ;
29     rr:predicateObjectMap [
30         rr:predicate foaf:name ;
31         rr:objectMap [ rr:column "name" ] ;
32     ] ;
33     rr:predicateObjectMap [
34         rr:predicate ex:ownsCar ;
35         rr:objectMap [
36             a rr:RefObjectMap ;
37             rr:parentTriplesMap ex:CarMap ;
38             rr:joinCondition [
39                 rr:child "OwnsCar" ;
40                 rr:parent "RegNum" ;
41             ] ;
42         ] ;
43     ] .

```

Listing 5.1: A R2RML mapping that generates the graph in Listing 5.2 on the following page.

```

1 @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
2 @prefix foaf: <http://xmlns.com/foaf/0.1/> .
3 @prefix ex: <http://www.example.org/> .
4
5 ex:car/CE54321
6     a ex:Car ;
7     ex:hasReg "CE54321" ;
8     ex:carModel "Honda Civic" .
9
10 ex:car/DK12345
11    a ex:Car ;
12    ex:hasReg "DK12345" ;
13    ex:carModel "Tesla Model S" .
14
15 ex:person/1
16    a foaf:Person ;
17    foaf:name "John" ;
18    ex:ownsCar ex:car/DK12345 .
19
20 ex:person/2
21    a foaf:Person ;
22    foaf:name "Ann" .

```

Listing 5.2: The resulting RDF graph given by applying the mapping in Listing 5.1 on the preceding page to the database in Figure 5.3.

Chapter 6

API Design

When doing any sort of programming task, using an appropriate API can make the task a lot easier. An API should make abstractions that simplify and speed up the process of making complex programs. High level programming languages, like Java, provide large libraries that contain most of the common functionality that a programmer needs. This makes it easier for the programmer to focus on the actual problem that should be solved, instead of focusing on all the intricate details of how everything works on a low level.

APIs can be divided into several categories. Two wide categories are behavioural, and structural APIs. Behavioural APIs are used to make complex tasks easier to solve. These contain algorithms that can be used by the programmer. A structural API is used to model data structures. These two types are not mutually exclusive, as they often overlap. For example, a structural API may contain complex algorithms to manipulate its components.

Behavioural APIs provide algorithms that can help the programmer solve difficult tasks. An example is the Java Socket API, which lets developers do network communication without having to deal with all the intricate details of the OSI stack. Without such an abstraction, network communication would require much more work to perform.

Structural APIs focus on modelling some kind of data structure. The functionality of such an API is usually centered around manipulating elements in the data structure. An example of this is the Java Collections API [20], which contains interfaces and classes that represent common data structures like lists, maps, sets and so on. Using an API like this will let the developers work on common data structures, without having to implement them on their own. This reduces the programming effort, as well as increasing performance.

A widely used API, like Java Collections, will also increase the interoperability among other APIs. Several implementations of the same functionality will often lead to difficulties for the developer. For example, if two APIs

uses different list abstractions, the developer would have to convert between the lists in order to use both APIs.

6.1 API Requirements

When designing an API, one must keep in mind that other people are going to use it. We therefore need some design goals that make it good to use. Here is a list of some important design goals that an API should meet (in no specific order):

1. It must be easy to learn.
2. It must be easy to use.
3. It must be hard to misuse.
4. It must be easy to maintain code that uses the API.
5. It must be correct.
6. It must fulfill the API's specific requirements.
7. It should be easy to extend.
8. It should be easy to maintain the API itself.

Most of these design goals may seem trivial. At the same time, some of them might seem hard to fulfill. Some advice may therefore be helpful.

General Advice

This section will give some advice that will help meeting the design goals. The scope of this advice is somewhat smaller than what is coming in the next two sections, but that does not make them any less important.

Identify the requirements of the API. Without knowing the requirements, it is impossible to know if the API is working correctly. The requirements should be clear and concise, so it will be easy to verify whether a requirement is fulfilled or not. The requirements should give the reader a full understanding of what functionality the API will have, as well as how the API should be built. The requirements may also give some constraints that specify what the API should not do.

Document everything in the API. The documentation will help new users to navigate through the API. A thorough documentation will also make the API harder to misuse. Classes should be documented with what they represent. Methods should be documented with what they do, and what side effects they may have. Interfaces should be described in detail about what its implementations should do. If the interface can be instantiated with, for example, a factory in the API, it may be a good idea to say that in the interface documentation.

An API should have a central access point that can be used to get the most important functionality of the API. This will be of great help to new users, as a large API often can be hard to navigate through. The documentation of the access point may contain information about how to access other parts of the API as well.

The naming of methods, classes, variables and so on, are always important. The user should not be confused about what the components of the API do. A good name should be self-explanatory, while at the same time concise. The naming should also be consistent, so that the same word means the same thing everywhere. This will make the API more readable and easier to use. For example, consider these two method declarations:

```
public Car[] getAllCarsWithBlueColor();
public Car[] getCars(Color c);
```

The first method declaration has a very long and specific name. The method will return all blue cars, and can be used only for that purpose. The second method declaration has both a short and concise name, and is more general than the first method. What if the user wants to get all red cars? By using the first naming scheme, the API would also need a method to return all red cars. By using the second naming scheme, only one method is enough.

The API should fail fast if something goes wrong. For example, if an invalid object is made, the API should throw an exception when the object is made, not when it is used. It is always easier to debug code that fails at the moment the mistake happens. Throwing an exception too late will make it very hard to track down the actual source of the mistake.

Make the API clean

Making the API clean is perhaps the most important advice. Not only the code itself, but also the structure of the API, should be clean. Having a well-known structure (by using design patterns) together with well-formed code, will greatly help the readability of the code. This will in turn make it much easier for users to both learn and use the API. It will also make it easier for the developer to maintain the API. The two next paragraphs will give some advice for making the structure of the API clean. The first paragraph concerns what the API should do. The second paragraph concerns how the API should be exposed to the user.

The API should focus on doing one thing, and it should do that thing well. It might be tempting to add methods and classes for everything that a user could possibly want. However, this may cause more harm than good. More functionality in the API means more things can go wrong. In addition, the user will have a hard time finding out how to use the API properly. It is always easier to add functionality than to remove it. Removing existing functionality will destroy backwards compatibility. It may therefore be

a good idea to release the first version of the API with a low amount of functionality, and then add more later based on user feedback.

A clean API should not expose more of the code than what the user needs. A common tip is to code towards an interface instead of an implementation. Here, *interface* is not meant as a Java interface, but rather an abstract definition. The only thing the user of the API should need to care about is how the API is used, not how it works internally. Some design patterns can help greatly with this. An example is the Abstract Factory Pattern (discussed in Section 7.2 on page 36), where the factory classes have methods that return objects that implement a specific interface. The objects' concrete classes are unknown for the user.

In programming languages like Java, there are many conventions for making clean code. These conventions regard things like how to define getter and setter methods, naming conventions, and so on. As previously mentioned, there are also standard APIs that should be used when needed. There is usually no reason to reimplement existing, well-known functionality.

Make the API modular

Having a modular API design is important for several reasons. First of all, it makes the code easier to maintain. With clearly separated and independent modules, the programmer does not have to focus on the complete architecture when doing work on a single module. With increasingly large and complex APIs, splitting it into modules will greatly benefit the design. Learning the API will also be much easier, because the user can keep focus on the modules that are needed, instead of the whole API.

When using a modular design, the modules will need to communicate through interfaces. These interfaces must be made early in the design, because changing them later may affect a lot of modules. The goal is to be able to develop the modules individually. The modules must therefore assume that the interfaces they depend on, do not change.

As mentioned earlier, the API should be programmed towards an interface, not the implementation. There is a straight forward approach to make a modular design that takes advantage of this principle. It consists of a specification module and an implementation module. The specification module contains interfaces and some factory classes, together with a precise documentation on how it should work. The implementation module contains implementations of the interfaces specified in the specification module. The implementation module is dependant on the specification module. The user can only access the specification module, while the implementation module is hidden. There can be multiple implementation modules, which can be acquired from a factory class in the specification module. For this to be successful, there must be at least one implementation module available for the specification module.

Chapter 7

Design Patterns

Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice [3, p. x].

This quote by Christopher Alexander, was originally about towns and buildings, but it applies well to object-oriented design patterns as well. A design pattern gives us a general way of solving a family of similar problems. It is not like having a hash map or a linked list, which can be reused as is, but rather a common pattern that can be adapted to a group of similar problems. A design pattern has four essential elements [15, p. 3]:

- The **pattern name** is a handle we can use to describe a design problem, its solutions, and consequences in a word or two. It lets us design at a higher level of abstraction.
- The **problem** describes when to apply the pattern. It explains the problem and its context.
- The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations.
- The **consequences** are the results and trade-offs of applying the pattern. The consequences may concern space and time trade-offs, or address language and implementation issues. The consequences may affect a system's flexibility, extensibility, or portability.

Design patterns can be very useful to solve common problems when programming. A pattern provides a new level of abstraction on top of the actual problem instance. When solving a complex programming task, a design pattern can help finding the appropriate classes to implement. It also gives programmers a common vocabulary, which makes it easier to talk about the design of a program, without having to know everything about the problem specifics.

Many design patterns are widely used in systems all over the world. This means that the most known design patterns are tested in a lot of real world systems. We can therefore have high confidence in their robustness. The hard part may be to find out which design pattern to use. There are a large amount of different design patterns, each with their own advantages and disadvantages.

Another advantage of using design patterns are that it makes it easier for others to understand the code. When the code follows a well known design pattern, the design pattern itself provides good documentation on how the code works. For example, if an API uses the Abstract Factory Pattern (which will be discussed in Section 7.2 on page 36), the user can learn how to create the objects just by knowing how the pattern works.

Design patterns can be quite language dependent. Some languages may have built-in constructs that can accomplish the same thing as a pattern. An example of this is the programming language *Scala*, where a class easily can be specified as a singleton (see Section 7.1) in its declaration. Many design patterns are meant for an object oriented programming language, and can therefore be useful for a lot of different languages.

There are different kinds of design patterns. It is common to group them into three categories. These are *creational*, *structural* and *behavioral* patterns. Creational patterns abstract the instantiation process of objects. Structural patterns concern the composition of the classes, and how they can be used to form larger structures. Behavioral patterns focus more on algorithms and the communication between objects.

7.1 Singleton

The singleton pattern is a quite simple design pattern that is used to make sure that a class can be instantiated only once. Even though the concept of a singleton is simple, the implementation of it can be harder. It may be very hard to ensure that no more than one object is made of a class. In addition, we must make sure that the singleton has a global access point, so that the singleton can be retrieved by the user. This section will describe how to make a singleton class in Java.

There are several disadvantages of using the singleton pattern. Some people criticise the pattern for introducing a global state to the program. However, in contrast to global variables, a singleton does not use any memory resources before it is instantiated the first time. Neither does it use any global namespace, as all members of the singleton must be accessed through the object.

Using a singleton also makes the program difficult to test. One of the reasons is that the instantiation process of the singleton can not be controlled by the programmer. This is a problem, because it makes it impossible to create a test instance of the singleton. Using the singleton as a test instance can

lead to problems if the singleton preserves its state between the different tests (the tests should be isolated from each other).

Another disadvantage is that it is quite hard to make a subclass of a singleton. To make the user of the class unable to instantiate new objects of the singleton class, the constructor must be set to private. This makes the constructor invisible for the subclass. Also, the subclass should not be instantiated if the superclass has been instantiated earlier.

When deciding to use the singleton pattern, it is important to be very sure that only one object of the singleton class will ever be needed. Overuse of the pattern can lead to problems if it later turns out that more objects are needed. Some alternatives should therefore be considered before making a singleton class. For example, it may be enough to pass a single object of the class to the classes that need it. This may, however, give the user of the object some responsibility to use the object in the correct way.

Singleton Implementation

A simple approach to implement a singleton class in Java, is to have a static `getInstance` method that returns the object of the class (see Listing 7.1 on the following page). This is the global access point for the singleton. The method creates a new object when it is called the first time, and then returns this object on all subsequent calls. In addition, to avoid that the user creates an object without the use of the `getInstance` method, the constructor of the class is made private. On first sight, this seems quite simple. However, it can be quite hard to ensure that no more than one object of the class will be created.

The Java Reflection API allows the programmer to modify the runtime behaviour of the program. With this API, it is possible to access classes, methods and so on, and use them as normal objects. For example, an object of the class `Class`, can create a new object of the class without using the `new` keyword (by using the `newInstance` method). This opens up for another possibility to create more than one object of a singleton. This is known as a *reflection attack*. However, when someone wants to use reflections to create another object of a class, it is usually for a very special purpose. Making a singleton resistant to a reflection attack is therefore not that important.

There are several problems with the implementation in Listing 7.1 on the next page. Firstly, it is not thread-safe. If two threads calls the `getInstance` method at the same time, both threads might find that the `instance` variable is null and create a new object. These threads will then get different singleton objects. However, subsequent calls to `getInstance` will all return the same object (the object of the last thread that wrote to the `instance` variable). The obvious solution is to make the `getInstance` method synchronized. This assures that only one thread may execute a synchronized method in the object at the same time. This will work, but it may slow down the program significantly.

```

1  class Singleton{
2      private static Singleton instance = null;
3
4      private Singleton(){
5      }
6
7      public static Singleton getInstance(){
8          if(instance == null){
9              instance = new Singleton();
10         }
11         return instance;
12     }
13 }
```

Listing 7.1: A simple (but unsafe) singleton implementation.

```

1 enum Singleton{
2     INSTANCE;
3
4     private Singleton(){
5     }
6 }
```

Listing 7.2: A modern singleton implementation.

Another problem with the implementation in Listing 7.1, is serialization using the `Serializable` interface. Whenever a class is deserialized, Java will create a new object of the class. A good way to get around this, is to use an enum class. This utilizes the fact that Java guarantees that an enum class is instantiated only once in a program. This also makes the implementation thread-safe. In addition, it makes the singleton safe of a reflection attack, as it is impossible to create a new object of an enum using reflections.

As can be seen in Listing 7.2, the implementation of a singleton using an enum class is very concise. The global access point to the singleton is given by the `INSTANCE` member. The instantiation of the singleton is done when it is accessed for the first time, without the possibility for any other threads to interfere. This implementation is now considered the modern way of implementing the singleton pattern in Java.

7.2 Abstract Factory Pattern

The abstract factory pattern is a creational pattern that is used to construct families of related objects without having to know their exact classes. It provides an interface (the abstract factory) that can be used to create objects, without specifying the exact class of the object. A class that implements this interface is called a factory. The user will only know about the abstract factory interface, while the concrete class of the factory is

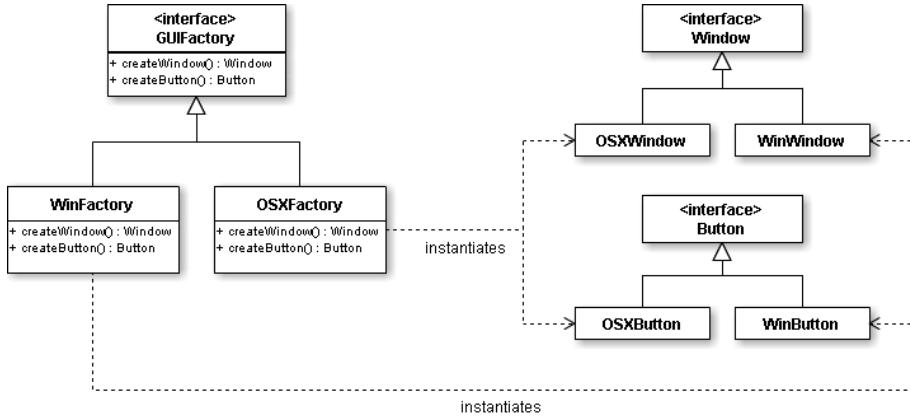


Figure 7.1: An example using the Abstract Factory Pattern.

hidden. This means that there can be several factories that implement the same interface. Each of these factories can be used to construct similar objects of the same theme.

The abstract factory pattern is useful to encapsulate the creation of objects. Creating objects might be quite complex, as it often involves processes that do not relate to the object directly. For example, the factory might need to add the objects to a registry of some sort. Using the objects that are created by the factory will be similar regardless of which factory is used. The interface that the concrete classes implement, forces the objects to be used in the same way.

It is very easy to change the family of objects that are created. It all depends on the factory that made them. It is therefore sufficient to change just the class of the factory. This class appears only once in a program, which is where it was instantiated. Because all objects created with the factory is in the same family, the whole family changes at once.

A disadvantage with the abstract factory pattern, is that the abstract factory interface is hard to extend to create new types of objects. By doing this, the new methods would need to be added to all the classes that implement the interface as well. The abstract factory interface specifies a fixed set of objects that it can create. It is possible to create a subinterface of the abstract factory, that can define new methods. However, this will remove the advantage of having an easy way of changing the factory that is used. A concrete factory that implements the superinterface will be incompatible with a factory that implements the new subinterface. This means that if the program uses any of the new methods in the subinterface, then the factory can not be changed to a factory that only implements the superinterface.

The implementation of the abstract factory pattern is pretty straight forward. Figure 7.1 gives an example of what the class structure may look like. There are the two factories, `WinFactory` and `OSXFactory`. These factories implement the interface `GUIFactory`. This interface defines methods that can be used to create objects for a graphical user interface.

```
1 GUIFactory gf = new WinFactory();  
2  
3 Window w = gf.createWindow();  
4 Button b = gf.createButton();
```

Listing 7.3: Usage of the Abstract Factory Pattern.

The `createWindow` method in these classes, returns a new object of type `Window`. The concrete class of the object is hidden for the user (`WinWindow` or `OSXWindow`). The same applies for the `createButton` method, which returns an object of type `Button` (with a concrete class that is either `WinButton` or `OSXButton`). The concrete class of the object that is returned, depends on which factory that was used. As expected, a `WinFactory` creates objects of type `WinWindow` and `WinButton`, while an `OSXFactory` creates objects of type `OSXWindow` and `OSXButton`. As can be seen in Listing 7.3, changing only the factory that was used (line 1) is sufficient to create windows and buttons for another operating system.

The factories are often implemented as singletons, so that all objects of the same concrete type is created by the same factory object. For example, all windows of type `OSXWindow` should be created by the same `OSXFactory` object.

Chapter 8

Library Independency

Many Java APIs are dependent on other libraries. An API may need functionality that has been implemented before. There is usually little reason to reimplement this functionality, as it is much better to use an already existing implementation. Sometimes, there may be several libraries that can be used to achieve the same thing. For example, both Jena and OpenRDF Sesame are libraries that handle RDF data. Instead of having the API depend on one specific library, we want to make the API compatible with all libraries that implement the needed functionality. This will make the API usable for more developers, because no underlying library is forced upon the user.

If the API only uses the library internally, the choice of library is usually not a problem. In this case, the user would not have to interact with the library anyway. However, problems arise if the user of the API has to deal directly with the classes from the external library. In this case, the library the user wants to use, may not be the library that the API supports.

We want the API to be able to support several libraries that implement similar functionality. The user of the API must be able to choose which underlying library to use, or even be able to extend support for another library. Ideally, we want the user to call a method with a library-argument, which then returns the access point to the API customized for the given library. Using the API should be no harder than if it only supported one specific library.

In order to achieve this goal, the API should fulfill some requirements:

1. The user must be able to choose which library that the API will use.
2. When the user has chosen a library, the API must not import classes from another supported library.
3. The API must work when only one of the supported libraries have been installed.
4. It must be possible to extend the API to support other libraries with

similar functionality.

5. The core of the API must be made in a generic way, so that it can work with any library that supports the needed functionality.

The first requirement is essential, as the point of all this is that the user can choose which library to use. Changing which library to use should be as simple as only changing the initial call to get the API.

The second requirement ensures that the API does not load more classes than it should. If the user chooses a library, the API should not load any classes from any of the other supported libraries. This is related to the third requirement, because Java will throw an error if it tries to load a library that does not exist.

There may be more libraries with similar functionality than the API has support for. If the user wants to use another library, it should be possible to extend the API to support that library as well. This is the fourth requirement.

The fifth requirement refers to the core of the API. The core is the part of the API that needs to be independent from the external libraries. This core must be made in a generic way, because it does not know which library it will use.

The API will need to interact with a library that it does not know anything about. The only thing the API knows, is what the library is *supposed* to do. There are two main problems that arise when trying to meet these requirements:

1. How can the API use functionality from a library without knowing how it works?
2. How can the API deal with classes from the library without knowing their types?

An approach that can solve the first problem, will be discussed in Section 8.1 on the next page. The main point of this approach, is that the API must depend on an abstraction of the functionality it needs, rather than the library itself.

The second problem has to do with how the API should deal with classes from the library. The API may need to, for example, store objects of classes from the library. The user may need to access these objects through methods in the API. Approaches that can solve this problem will be discussed in Section 8.2, 8.3 and 8.4. As with the first problem, the classes that are needed from the library, will represent abstractions of what the API needs. All the three approaches have this in common. Note that these abstractions are not enforced by, for example, having the classes implement interfaces. This would make the libraries dependent on the abstractions. The abstractions will therefore have to be thoroughly documented.

The abstractions have to be constructed in a way that will make them fit

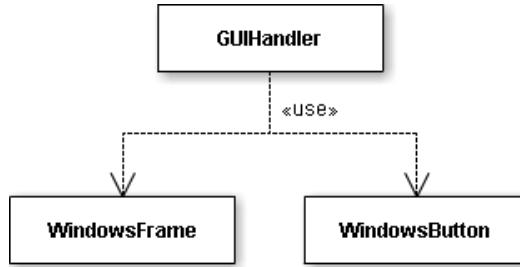


Figure 8.1: `GUIHandler` is dependent on `WindowsFrame` and `WindowsButton`.

with several libraries. Less constraints on the abstractions will make them compatible with more classes, which in turn may make it possible to add support for additional libraries.

8.1 Dependency Injection

Dependency injection is a design pattern where a dependent object gets its dependencies injected at run-time. The term was introduced by Martin Fowler in the article *Inversion of Control Containers and the Dependency Injection pattern* [19]. With dependency injection, the dependencies are passed to the object by an injector. The injector will typically be some kind of factory class. Dependency injection changes the dependency relationships between the modules. It will make the object dependent on an abstraction of its dependencies, instead of the dependencies themselves.

Figure 8.1, and Figure 8.2 on the next page, give an example of this. The class `GUIHandler` are going to make a GUI consisting of frames and buttons. It is dependent on the `WindowsFrame` and `WindowsButton` classes, which can only create frames and buttons for the Windows operating system. `WindowsFrame` and `WindowsButton` are reusable for any program that needs to create a Windows GUI. `GUIHandler` however, are reusable only in contexts that need to create Windows frames and Windows buttons. It would be better if the `GUIHandler` could be used to create other kinds of GUIs as well. Figure 8.2 shows how the dependencies will move when using dependency injection. `GUIHandler` will have dependencies to abstractions of frames and buttons. `WindowsFrame` and `WindowsButton` will have dependencies to, respectively, `Frame` and `Button`. Other classes can implement these interfaces as well, which will make `GUIHandler` able to handle more than just a Windows GUI.

There are three ways to inject the dependencies into an object [19]:

- Constructor injection
- Setter injection
- Interface injection

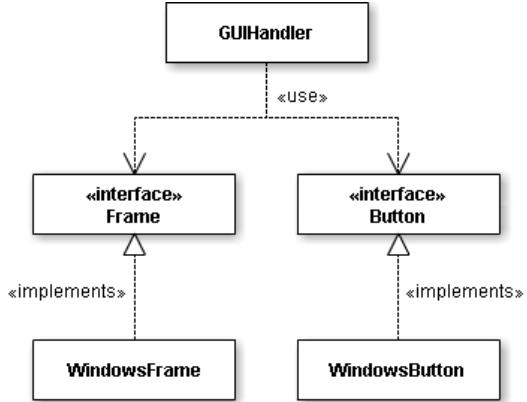


Figure 8.2: **GUIHandler** is dependent on the abstractions **Frame** and **Button**.

```

1  class GUIHandler{
2      Frame f;
3      Button b;
4
5      GUIHandler(Frame f, Button b){
6          if(f == null || b == null){
7              throw new IllegalArgumentException();
8          }
9          this.f = f;
10         this.b = b;
11     }
12 }
```

Listing 8.1: A simple implementation of constructor injection.

With *constructor injection*, the dependencies are injected through the class constructor. This will ensure that the dependencies are present when the object is created. However, it limits the possibility to change the dependency later, but this is often a good thing. Changing the dependency may in some cases make the object invalid. A simple implementation of constructor injection on the **GUIHandler** class from Figure 8.2, can be seen in Listing 8.1. As can be seen, the constructor will throw an exception if one of the dependencies are **null**.

Setter injection uses a setter method to inject the dependencies. This offers more flexibility to change the dependencies after the object has been created. However, if there are more than one dependency, it may be harder to ensure that all the dependencies have been injected. The object should therefore be validated to see if all dependencies have been set, before it is used. To implement the **GUIHandler** class with setter injection, it needs to implement two setter methods. One to set a **Frame** and one to set a **Button**.

With *interface injection*, the injection is done through an interface specifying setter methods. This can be used to enable the dependencies to do the work of the injector themselves. A dependency can have an *inject* method

```

1 interface GUIAbstraction{
2     public void setFrame(Frame f);
3     public void setButton(Button b);
4 }
5
6 interface Injector{
7     public void inject(GUIAbstraction g);
8 }
9
10 class GUIHandler implements GUIAbstraction{
11     Frame f;
12     Button b;
13
14     public void setFrame(Frame f){
15         this.f = f;
16     }
17
18     public void setButton(Button b){
19         this.b = b;
20     }
21 }
22
23 class WindowsButton implements Injector{
24     public void inject(GUIAbstraction g){
25         g.setButton(this);
26     }
27 }
```

Listing 8.2: An implementation of interface injection.

that takes an instance of the interface, which it can use to inject itself. However, this approach does not remove the original injector class, because the object and the dependencies will still have to be introduced somewhere. Implementing interface injection on the `GUIHandler` class can be done like in Listing 8.2.

So, how is dependency injection useful for our purpose? As can be seen in Figure 8.2, the dependencies have to implement the abstraction interfaces that the object uses. In our case, the dependencies are pre-made libraries that do not know about any such interface. The libraries can not be dependent on anything from the API. This can be solved by using a *library configuration*.

8.1.1 Library Configuration

The API must be able to use functionality from a library which it does not know the specifics of. In order to do this, we must define an interface which defines methods for all the functionality that the API needs. The interface can then be implemented by classes using the libraries that are going to be supported. This interface will be called a *library configuration*.

The library configuration interface will act as an abstraction of the libraries

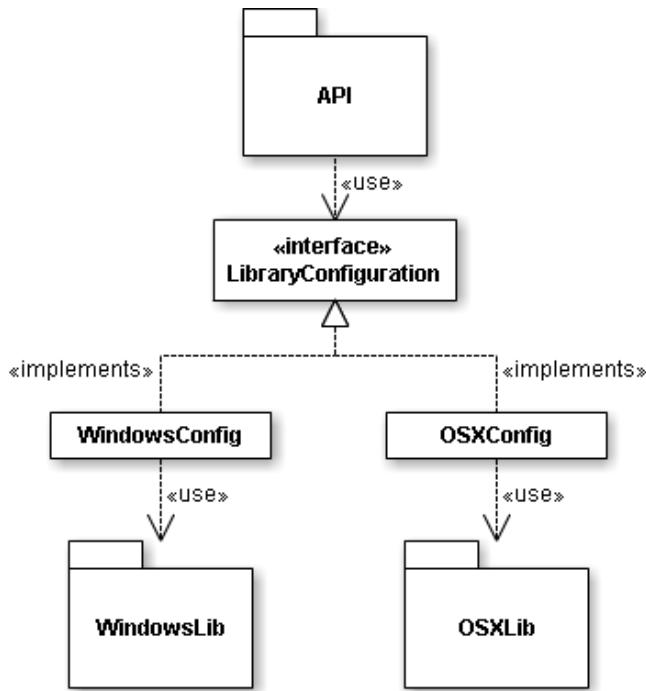


Figure 8.3: Overview of dependencies using a library configuration.

that the API is going to support. However, the libraries will not directly depend on the library configuration. Instead, the library configuration will be implemented by a class that depends on the library. Figure 8.3 illustrates an example where an API needs to use a library configuration to handle operating system specific functionality. The library configuration has two implementations, one uses the Windows library and the other uses the OSX library.

Every class in the API that needs to use a library, will have a reference to a library configuration object. This will be done with dependency injection using the *constructor injection* approach. This will make sure that the dependency exists when the class is instantiated. Also, the reference will be hard to change after initialization. Changing an object's library configuration after it has been initialized, will very often make the object invalid.

Listing 8.3 on the next page gives an example of how a library configuration can be implemented in Java. As with the example in Figure 8.2 on page 42, the `GUIHandler` class is going to create a GUI. It takes a `LibConfig` object as a parameter to its constructor. The `LibConfig` interface defines a method to create a frame. The method declaration does not imply anything about how the frame will be made. The classes `WindowsConfig` and `OSXConfig`, implement the interface. These two classes use different libraries in order to implement the `createFrame` method. Which library that is used will not be visible for the `GUIHandler`. If the classes are defined in different files, both libraries does not even have to be imported, because `GUIHandler` will only

```

1 interface LibConfig{
2     public void createFrame();
3 }
4
5 class WindowsConfig implements LibConfig {
6     public void createFrame(){
7         // Create a new WindowsFrame.
8     }
9 }
10
11 class OSXConfig implements LibConfig {
12     public void createFrame(){
13         // Create a new OSXFrame.
14     }
15 }
16
17 class GUIHandler {
18
19     final LibConfig lc;
20
21     GUIHandler(LibConfig c){
22         if(c != null){
23             lc = c;
24         } else{
25             throw new NullPointerException();
26         }
27     }
28
29     public void createFrame(){
30         return lc.createFrame();
31     }
32 }
```

Listing 8.3: A simplified example of how the library configuration (`LibConfig`) can be used.

need one of them.

The library configuration will also act as the library argument for the API. A factory class can have methods that return the API using the already supported libraries, as well as a method that takes a library configuration as a parameter. This allows the user to extend the support for additional libraries. By using one of the supported libraries, the user does not have to be exposed for the library configuration at all.

This approach alone will not make the API completely independent of its dependencies. A problem that remains is how the classes from the external library can be treated. What if the `GUIHandler` class has to store a variable of a type from the external library? Or if the `createFrame` method returns a frame specific for the library? There are several ways of solving these problems, something which will be described in the next sections.

8.2 Generics

An obvious idea that makes it possible to use arbitrary classes from the libraries, is to use Java Generics. Generics is a feature that was added in Java 5.0. It allows “a type or method to operate on objects of various types while providing compile-time type safety” [21].

Generics works by adding type parameters to a class, an interface or a method. A type parameter specifies the class that are going to be used. In order to instantiate a generic class, the type parameter has to be given. A generic class or interface must have the type parameter specified when it is instantiated, or by a sub- or implementing class. A generic method’s type parameter must be specified when it is called, or it can be inferred if the method takes a generic parameter.

A common use case for generics is data collections. For example, the `List` interface in Java is declared generic. A small part of it is declared like this:

```
1 public interface List<E> extends Collection<E> {
2     boolean add(E e);
3     E get(int index);
4     ...
5 }
```

Listing 8.4: The Java `List` interface.

As can be seen, the `add` method takes an object of the type `E` as a parameter, while the `get` method returns an `E`. The `E` is the type parameter of the list. The list can be instantiated with the class `ArrayList`. In order to do that, the type parameter has to be specified. For example, creating a list of strings can be done with this code:

```
List<String> l = new ArrayList<String>();
```

The type parameters makes it possible to check the generic types at compile-time. This reduces the amount of type casting, which can cause a run-time exception if the types are not compatible. Casting can not be checked at compile-time. This is the main advantage of using generics. Runtime exceptions can often be hard to debug, because the origin of the error might not be where the exception is thrown.

When the generic type parameters are compiled, they are first checked for correctness, before the generic type information is removed through a process called *type-erasure*. This means that there will be no generic type information present at run-time. For instance, `List<String>` will be converted to the raw type `List`.

In order to use generics for our purpose, the classes in the API would need to have type parameters. The number of type parameters that is needed,

depends on the number of classes the API needs from the external library. In general, this can be very many.

A high number of type parameters can make the code using the API very verbose. Every time the user declares a variable of a class from the API, all the type parameters have to be included in the declaration. Even though only one class in the API uses classes from the library, the type parameters will often propagate to most of the other classes in the API. The reason for this is that if one class is generic, then all classes with a variable of that class will also have to be generic (unless it specifies the type parameter). This might be clearer with a small example:

```
1 class ListWrapper<T> {
2     List<T> list;
3
4     ListWrapper(List<T> l) {
5         list = l;
6     }
7 }
```

Listing 8.5: An example showing how the type parameters propagate to other classes.

As can be seen, the class must be declared as `ListWrapper<T>`. Omitting the type parameter will make the type parameter for the input list unknown. This would result in a compile error. If some other class would need a generic pointer to a `ListWrapper`, then that class has to be generic as well. It is therefore very rare that only parts of the API will have to be generic. When an API declares a class, it will be very likely that some other class has a variable of that type.

Another disadvantage with having all the type parameters propagate back to the user, is that the user would need to know about all the external classes that the API uses. Ideally, only the classes that are actually used by the user, should be exposed.

The decision of whether to use generics for our purpose, mainly depends on one thing. The number of type parameters that are needed. A large amount of type parameters will make the code verbose and hard to read, and will therefore decrease the usability. If the API exposes only one class from the library to the user, it might be worth it in order to get the compile-time type safety.

Many of these concerns become evident in the example in Listing 8.6 on the next page. In this example, the `GUIHandler` can create a `Window` that contains frames and buttons. `Window` is a class in the API. The `GUIHandler` is created on line 7. As can be seen, with only two type parameters, this declaration is very verbose. On line 8, the `GUIHandler` creates a `Window` which initially contains only a frame. This declaration also needs the type parameters. A `WindowsButton` is added to the `Window` on line 9, and retrieved on line 10.

```

1 WindowsFrame wf = new WindowsFrame();
2 WindowsButton wb = new WindowsButton();
3
4 LibConfig<WindowsFrame, WindowsButton> lc =
5         new WindowsConfiguration();
6 GUIHandler<WindowsFrame, WindowsButton> gh =
7         new GUIHandler<WindowsFrame, WindowsButton>(lc);
8 Window<WindowsFrame, WindowsButton> w = gh.createWindow(wf);
9 w.setButton(wb);
10 wb = w.getButton();

```

Listing 8.6: Usage of an API with Java Generics.

A couple of things should be noticed from this example. Firstly, the declarations of objects using classes from the API gets very verbose. Even though this example only uses two type parameters, the declarations of the `LibConfig` and `GUIHandler` have to span over two lines. All declarations using classes from the API will have these type parameters. Secondly, even though we may not have used `WindowsButton` at all, it has to be specified as a type parameter.

On the positive side, the usage of the declared objects is similar to how it would have been without using generics. As can be seen on line 9 and 10, the type parameters are not needed here. The type-safety is also ensured by the compiler.

8.3 Class Objects

Another approach is to use class objects. In Java, all classes have a class object associated to them. These can be used to do several things: Checking whether the type of an object is that class, creating a new instance of the class and so on. A class object can be retrieved by calling an object's `getClass` method, or by adding `.class` to the name of the class. For example, retrieving the class object of `String` can be done like this:

```
Class<String> cls = String.class;
```

Note that `Class` is generic. This makes the class contain both run-time and compile-time type information. Using a class object as a parameter to a method makes it possible to use the type parameter as a return type of the method. This can be used to create getter methods that return a type specified by the caller. This means that instead of having to return an object of class `Object`, which then has to be cast to the correct class by the caller, the method can return the correct class itself. The declaration of such a method can look like this:

```
public <C> C getInstance(Class<C> cls)
```

The question is how this can be used for our purpose. The classes in the API will contain variables of unknown types from an external library. With generics, the type of these variables would be given by the type parameters of the class. When using class objects, the types of these variables will be set to `Object`. Setter methods that assign these variables, will also take an `Object` as a parameter. This makes it possible to store objects from the external library, without having to know the exact class.

This introduces some problems. First of all, we lose the compile-time type safety that we have when using generics. Because it is possible to call the setter methods with whatever class the user wants, we can end up with setting a variable to an object of the wrong type. The mistake would not be detected before the getter method are called, because that is when the object will be cast to the correct type.

However, this can be solved with the *library configuration*. It can have methods that return the class object of each class from the library that the API needs to use. By doing this, the type of the object can be checked when the setter method is called. We will still get a run-time error instead of a compile-time error if the type is wrong, but at least the run-time error will be thrown as soon as the mistake actually happens. A setter method would therefore have to look something like this (`lc` is the library configuration):

```
1 public void setInstance(Object i){  
2     if(i != null && !lc.getObjectClass().isInstance(i)){  
3         obj = i;  
4     } else{  
5         throw new IllegalArgumentException("Error!");  
6     }  
7 }
```

Listing 8.7: Example of a set-method using class objects.

There are several advantages of using class objects. Firstly, the code that uses the API will be much cleaner than it would be with generics. Using a lot of classes from the external library will not have any effects on how the code will look. In addition, the user will only be exposed to the library's classes when actually using them. This means that, in contrast to using generics, the user does not have to know about all the classes that the API uses from the library. However, exactly which class that is used by a specific setter or getter must be specified in the documentation.

If the API needs a lot of classes from the external library, using class objects will greatly increase the readability of the code using the API. The downside is that we lose the compile-time type safety. Using class objects therefore requires some caution. Checking the types should be done as early as possible to avoid difficulties with debugging the code.

A disadvantage with class objects, is that the API may become more difficult to maintain. Because objects will be of the type `Object`, it may become hard to know which types the objects actually have. Variables with the type

```

1 WindowsFrame wf = new WindowsFrame();
2 WindowsButton wb = new WindowsButton();
3
4 LibConfig lc = new WindowsConfiguration();
5 GUIHandler gh = new GUIHandler(lc);
6 Window w = gh.createWindow(wf);
7 w.setButton(wb);
8 wb = w.getButton(WindowsButton.class);

```

Listing 8.8: Usage of an API with class objects.

Object should therefore have a comment that tells the developer which class it represents.

Listing 8.8 gives an example of how to use class objects to create a **Window**. The result of running this code is exactly the same as in the example using generics (see Listing 8.6 on page 48). To recap, the **GUIHandler** takes a **LibConfig** as a parameter, and creates a new **Window** containing a **WindowsFrame**. A **WindowsButton** is then set, and retrieved from the created **Window**.

As can be seen, the code using class objects is much cleaner than the code using generics. There are no type parameters needed in the declarations. The only extra thing that has to be done by the user, is to send the class object of **WindowsButton** as a parameter to the **getButton** method (see line 8). If the parameter to the **setButton** method has the wrong type, it will throw an exception. The **getButton** method may also throw an exception if the class object parameter is wrong. On line 8, the compiler will check that the type of **wb** is equal to the class object parameter of **getButton**.

8.4 C++ Templates

C++ provides a template mechanism that allows classes and functions to operate on generic types. It tries to accomplish much of the same things that Java Generics do, but it works very differently. C++ templates could solve some of the difficulties that comes with the other approaches. However, since we are dealing with an API written in Java, this will not be applicable in our case. Anyway, this section will discuss how it could have been done using templates.

There are two kinds of templates: Function templates and class templates. Function templates make it possible to create generic functions, while class templates make it possible to create generic classes.

Listing 8.9 on the facing page gives an example of a simple class template. In this example, **T** is the type parameter. With Java Generics, the compiler would first look for type errors, before the type-erasure process would convert the generic parameters to the **Object** class. However, this is not how it is done with C++ templates. The compiler will instead create a

```

1 template<class T>
2 class Container {
3     T Data;
4
5     public:
6         Container() {}
7
8     void SetData(T nValue) {
9         Data = nValue;
10    }
11
12    T GetData() const {
13        return Data;
14    }
15 };

```

Listing 8.9: An example of a class template.

new class (or function) for each type that the class has been instantiated with. For example, the compiler will create two classes when instantiating the template class like this:

```

Container<int> c1;
Container<double> c2;

```

There are a lot more to templates than this, but much of it is not useful for the purpose of creating a library independent API. As was seen in Section 8.2 on page 46, one of the problems of using Java Generics, was that every declaration had to contain all the type parameters. This can be solved in C++ by having a structure that contains type definitions [11]. These type definitions specify which classes that will be used from the external library. The structure can be used as the type parameter to the template classes in the API. By doing this, we will only need one type parameter to the classes. A simple structure of this type can be defined like this:

```

struct Configuration {
    typedef double t1;
    typedef int t2;
    typedef List<int> t3;
};

```

This structure defines the type **t1** as **double**, **t2** as **int**, and **t3** as a list of integers. The types could also be defined as classes from a library. We can now extend the **Container** class so it can store three items, utilizing the **Configuration** structure. This can be seen in Listing 8.10 on the following page.

The types of the elements that are stored depend on the configuration that is given as the type parameter. Note that the class are accessing elements of **Config**, even though it does not know that the elements exist. However, the

```

1 template<class Config>
2 class Container {
3   Config :: t1 data1;
4   Config :: t2 data2;
5   Config :: t3 data3;
6
7   void SetData1(Config :: t1 nValue) {
8     Data = nValue;
9   }
10
11  Config :: t1 GetData1() const {
12    return Data;
13  }
14
15  ...
16 };

```

Listing 8.10: Extended class template.

C++ compiler will check this based on the instances of the class that are created. This means that instantiating the class with the type parameter `int` will cause a compile time error, because `int` does not contain either `t1`, `t2` or `t3`.

As can be seen, C++ templates make it possible to use many classes from the external libraries, without having to add a lot of type parameters. The configuration structure groups all the needed types together. When using the API, the user will only need to know about the configuration of the library that is used. Using templates would therefore be very convenient when making a generic API in C++.

Chapter 9

R2RML API Design

The R2RML API was originally designed as a part of the Optique project. The Optique system needed an API for creating and managing R2RML mappings. The goal was to have an API that provided an abstraction above the level of the mappings' RDF serializations.

The first version of the API, that was made specifically for Optique, used the OpenRDF Sesame API to handle RDF. During the development of this version, we decided to publish the API as an open source project and try to establish it as a de-facto standard. This meant that the API had to be made applicable in a wide range of scenarios. The API should therefore be made independent of any external library. The user should be able to choose which library the API will use. The only requirement to the chosen library was that it needed functionality to handle RDF.

To make the API independent of a specific external library, an idea would have been to create an implementation of RDF specifically for the R2RML API. However, there already exist several RDF implementations, so creating a new one was never really considered. In the new version of the API, we decided to add support for three widely used libraries. These are OpenRDF Sesame, Jena and OWL API (see Chapter 10). Users of the API could use one of these, or extend the support for another library.

9.1 Requirements

One of the most important things to have in place when making an API, is the requirements. If there are no requirements, it will be impossible to determine if the API does what it is supposed to do. The requirements should specify what the API is for, how it should be built, and how the interface to the users should be. The requirements should be clear and concise, so that they can be easily verified.

The requirements are divided into two groups. The first group specifies some functional requirements. These describe what the user can expect

of the API's functionality. The second group specifies implementation requirements. These describe the structure of the API in greater detail, as well as describing some of the constraints that must be implemented. Some of the implementation requirements will also help toward meeting the general requirements specified in Chapter 6.1 on page 30.

Functional Requirements

These requirements mainly concern how the user will interact with the API and what functionality the API will have. The users of the R2RML API will be programmers that want to create and manipulate mappings. These programmers do not want to manipulate the mappings directly as RDF. Instead, they want a structure that abstracts away from the RDF, while at the same time being familiar to use.

1. The API must provide an abstraction of the R2RML mapping language, above the level of RDF. The class hierarchy must be similar to the R2RML data model.
2. The users of the API must be able to create and manipulate R2RML mappings easily.
3. It must be possible to manipulate all components of the R2RML mapping programmatically.
4. It must be possible to use the API together with Jena, OpenRDF Sesame and OWL API.
5. It must be possible to add support for other external libraries that can handle RDF.
6. It must be possible to take an RDF graph containing R2RML mappings as input, and construct a collection of triples maps from it.
7. It must be possible to serialize the triples maps to an RDF graph using the R2RML vocabulary.

The first requirement gives the basis of the structure of the API. The R2RML data model fits very well for an object oriented API, so there is little reason to make the API different from that.

The second requirement is central, as we want to be able to create and manipulate the mappings without having to deal with RDF directly. As the third requirement states, it must be possible to manipulate all the components of the R2RML mapping.

The fourth and fifth requirement are about support for underlying libraries. The API must be made generic, so that it can support several libraries. The only requirement for the underlying library is that it must be able to manipulate RDF graphs. In addition to supporting the three specified libraries, it must be possible to extend the API to use other libraries as well.

The sixth and seventh requirement deal with input and output of the mappings. The API must be able to parse an RDF graph, and produce a collection of triples maps from it. The RDF graph should be a class that represents a collection of triples in the underlying library that is used. It must also be possible to take the collection of triples maps and produce an RDF graph from it. A mapping that has been parsed, then written back again, does not necessarily have to be equal. The output may for example use shortcut predicates for constant-valued term maps, or add types for some of the components.

Implementation Requirements

As mentioned, the implementation requirements deal with the structure of the API, and some of the constraints that must be implemented. These will hopefully make the API robust, good to use and easy to maintain.

1. The API should give the user access to interfaces in order to hide the implementation details.
2. It should be very hard to create a structurally invalid R2RML mapping.
3. The API must work when only one of the supported libraries are present (or zero if the user is using another external library). The API must only load the library that has been chosen by the user.
4. Two mappings are equal if and only if their contents, including their resource URI, are equal.
5. All components of the mapping must have a resource. If the resource is not given, it must be set to a fresh blank node.
6. Shortcut properties for constant-valued term maps should be used in the serialization when appropriate.
7. The serialization should add `rdf:type` triples for all the types of components in the mappings.

The user should not be concerned with the implementation details of the API. Therefore, the first requirement states that the API should give the user access to interfaces, rather than the concrete classes. This can be achieved by using, for example, the abstract factory pattern (see Section 7.2 on page 36).

The second requirement states that it should be difficult to create a invalid mapping. Especially with regards to the structure of the mapping. For example, a triples map must have a logical table, so creating a triples map without that should not be possible. A mapping may, however, be incomplete while it is being edited. There may be, for example, a finalize method that can be called after the editing is done. However, it may be hard

to verify that a mapping is completely valid. Therefore, the requirement only states that it should be very hard.

The third requirement concern how the API is going to use the external libraries. The user will choose which underlying library to use. The API must work when only the chosen library is present. If there are other supported libraries that are present, the API should only load the chosen one into memory.

The fourth requirement defines what is needed for two mappings to be equal. Two mappings must have the exact same contents to be considered equal. The resource URIs of the mappings must also be the same. The reason for this requirement has to do with how the mappings will be handled by, for example, a `Set`. Consider two equal mappings with different resource URIs, which will be added to a `Set`. Both of these mappings should be added to the set successfully. However, if they are considered equal, then the set will only add the first one and throw away the second. This means that we will lose the resource URIs of the second mapping. This might have severe consequences for the application using the API.

The fifth requirement states that all components of a mapping must have a resource URI associated with it. A component without a given resource, will get a freshly generated blank node. Due to the fourth requirement, this means that two mappings where the resource URI have not been set explicitly, will never be equal. The resource is needed when the mapping is serialized.

The sixth and seventh requirements are about the serialization. The serialization should use the shortcut predicates `rr:subject`, `rr:predicate` and `rr:object` when serializing constant-valued subject, predicate, object and graph maps. The alternative to this would be to serialize all term maps with their full specification. However, this will just be a more verbose way of stating the same thing. As the seventh requirement states, the RDF serialization should also contain the types of all the components. For example, a subject map should have the types `rr:TermMap` and `rr:SubjectMap` added via the `rdf:type` predicate. The types of the components should be added so that they will not have to be inferred at a later stage in the mapping process. These requirements were added in order to make the serialization process more predictable.

9.2 API Architecture

Mapping Structure

The first functional requirement gives the basis for the architecture of the API. The API must model the data model of the R2RML mapping language as accurately as possible. A diagram of the data model can be seen in Figure 5.2 on page 21. Each of the components of the mapping will be

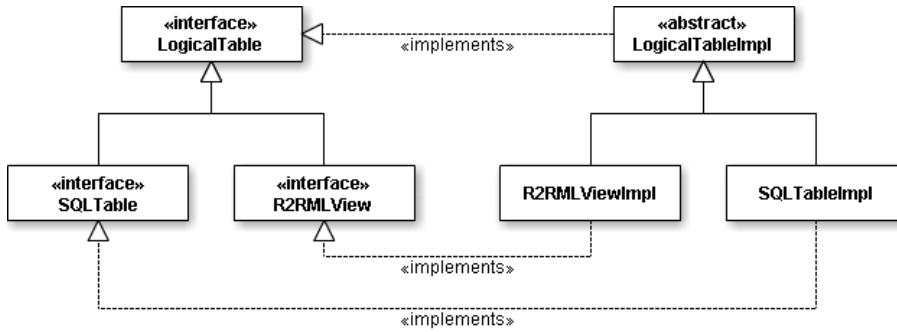


Figure 9.1: Class diagram of the LogicalTable.

represented by an interface. The manipulation of mappings will be done by accessing objects of these interfaces.

In addition to the components from the R2RML data model, the logical table will have the subinterfaces `SQLTable` and `R2RMLView`. In R2RML, a logical table has either an `rr:sqlQuery` or an `rr:tableName` predicate. The predicate that is used, decides whether the logical table is an R2RML view, or an SQL base table or view. From an object oriented point of view, a better approach is to make the two subinterfaces. The class implementing the logical table interface will be abstract. This makes it impossible to instantiate a logical table that is neither an `R2RMLView` or an `SQLTable`. Figure 9.1 gives an overview of how this will be done.

Templates and inverse expressions are structured constructs that should have their own classes. This will be more convenient than simply storing them as strings. For example, a template consists of a series of string segments and column names. The template class will have functionality to access the individual string segments and columns. This makes it possible to change parts of the template without having to construct a completely new one.

Some of the components of an R2RML mapping contain SQL queries. SQL queries are, like templates and inverse expressions, also structured constructs. However, we decided to store these as strings. There are two reasons for this. The first is that the user may want to use another query language than SQL. This can not be done if the API processes the SQL strings. The second reason is that the SQL queries would have to be handled by another external library. Because the API is going to be independent of any external library, the user would also need to choose which SQL library to use. This would make the API unnecessarily complex.

Library Support

In order to have support for the external libraries, the API must be made generic. We decided to use the variant with class objects, after testing the different approaches discussed in Chapter 8. This variant makes the code

concise and readable for the user, but some of the type checking will be moved from compile-time to run-time.

There will be a central access point to the API. This access point will be used to retrieve a customized version of the API, depending on which external library that will be used. As can be seen in Figure 9.2 on the next page, the class `R2RMLMappingManagerFactory` instantiates an object of the type `R2RMLMappingManager`. This object can be used to import and export mappings, as well as retrieving a `MappingFactory`. The `MappingFactory` is used to create the actual mappings.

The `LibConfiguration` interface is an essential part to make the API work with an external library. It specifies what the libraries have to do in order to be compatible with the core API. It works as an SPI¹ that the libraries have to implement in order to work with the API. Jena, Sesame and OWLAPI will each have their own implementations of the `LibConfiguration` interface. The `R2RMLMappingManagerFactory` class has methods that instantiate an `R2RMLMappingManager` for each of these three implementations. The factory will also have a method that instantiates an `R2RMLMappingManager` based on a custom `LibConfiguration`. This allows the user to extend the support for additional libraries by providing a new implementation of `LibConfiguration`. The `LibConfiguration` object is only meant to be used internally by the API.

The methods in the `LibConfiguration` are specified so that they will be easy to implement with most RDF libraries. The basic functionality includes methods for creating resources, blank nodes, triples and graphs. It also has a method that will return the resource of the `rdf:type` predicate for the library. These methods are all needed in order to serialize an R2RML mapping. In addition, the parser needs methods to retrieve resources from the triples in a graph. There are two methods for this, `getSubjects` and `getObjects`. The API also needs methods to retrieve the class objects of the classes that the library uses. These are needed in order to check the types of the objects that the API uses from the library.

The API needs three classes from the external library. These classes are needed in order for the API to handle RDF. An implementation of the `LibConfiguration` may however use more than these three classes internally. In order for the library to work with the API, it must have classes that represent the three following abstractions:

- An RDF resource
- An RDF triple
- An RDF graph

The RDF resource can either be a normal resource, or a blank node. A resource has to have an unique identifier in the form of a URI. In an RDF triple, the subject and predicate are resources, and the object is either a

¹Service Provider Interface

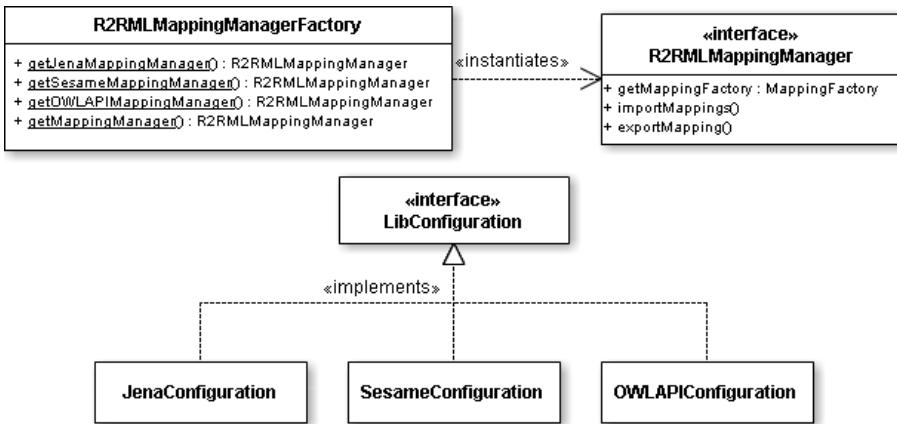


Figure 9.2: Class diagram of the central access point to the API.

resource or a literal. Literals are represented by strings, but the external library might wrap it into a literal class when making an RDF triple. The predicate resource can not be a blank node. The RDF graph represents a set of triples. Which classes are used by the supported libraries, are described in Chapter 10.

The get-methods in the API that return an object of a class from the library, will take a class as a parameter (see Section 8.3 on page 48). This class decides the type of the returned value. To not get a `ClassCastException`, the input class has to be equal to (or a superclass of) the class of the returned object. For example, the signature of the `getResource` method will look like this:

```
public <R> R getResource(Class<R> c);
```

If the resource class in a library is called `Resource`, then this method has to be called with `Resource.class` (or a superclass of this class) as a parameter. In order for the API to call these methods with the correct class object, it will call the method that retrieves the class in the `LibConfiguration`.

The set-methods that take a class from the library as a parameter, are a little easier. The type of the parameter will be `Object`. The method will check the type of the parameter when it is called, and throw an `IllegalArgumentException` if the type is wrong. To check the type, the API will use the `isInstance` method from a class object retrieved from the `LibConfiguration`. Which type that is allowed for the parameter needs to be well documented.

Creating Mappings

Creating new mappings will be done with a mapping factory. This will be implemented using the abstract factory pattern (see Section 7.2 on page 36). The factory instantiates objects that implement the interfaces

for the different mapping components. The factory will make sure that the components will get what they need as parameters. For instance, for a triples map to be valid, it must at least have a logical table and a subject map. The factory will also have some convenience methods that can, for example, create a triples map with a collection of predicate object maps.

An alternative to using the abstract factory pattern would be to use the builder pattern². The builder pattern focuses on creating complex objects step by step. However, we chose to use the abstract factory pattern because it focuses on creating a family of related objects. This fits well with the R2RML data model.

Parsing and Serializing Mappings

The parsing of the mappings will be done with the `importMappings` method in `R2RMLMappingManager`. This method takes an RDF graph as input, and returns a collection of triples maps. As previously mentioned, the `LibConfiguration` has to contain methods that can retrieve triples from graph. The R2RML mapping that will be parsed may originally come from, for example, a file or a triple store, but it has to be read into an RDF graph before the API can parse it. The reason for this decision was to give the user more control on how to retrieve the triples. The R2RML API should focus on the actual mappings, not reading RDF files or querying triple stores.

The serialization of mappings will be done by calling the `exportMappings` method in the `R2RMLMappingManager`. This method will take a collection of triples maps as input, and return a new RDF graph containing the RDF triples that represent the mappings. In order for the mapping to be serialized, its components need to implement the `SerializeR2RML` interface. This interface will have a method called `serialize`, that returns a set of triples. To serialize a mapping to RDF triples, all mapping components must have a resource associated with it. The components must therefore also implement the `ManageResource` interface, which contains methods to set and get the resource. The constructor of the mapping components will initially set the resource to a new blank node. This means that `LibConfiguration` must have a method that creates resources.

Installing the API

The API will be split into two parts. The main part contains the core API, which consists of all the library independent classes. The second part consists of all the library dependent classes. The first part will be archived in the main jar file. The second part will be stored in three jar files, one for each supported library. These will serve as a bridge between the core API and the library itself.

²http://en.wikipedia.org/wiki/Builder_pattern

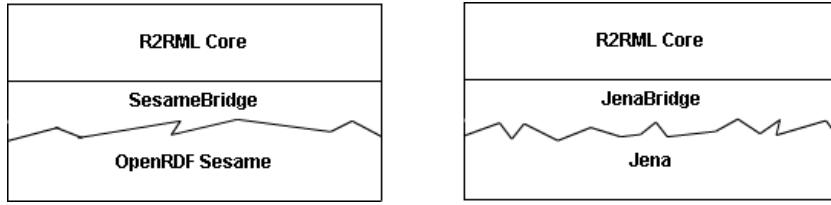


Figure 9.3: Illustration of how the bridge module makes the R2RML API work with an external library.

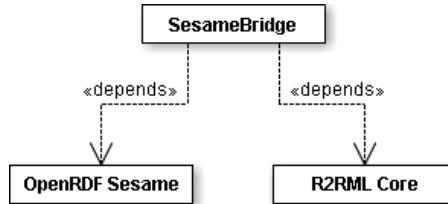


Figure 9.4: The bridge module is dependent on both the core module and the library.

The bridge module must contain everything that the API needs in order to use the library. More specifically, it must contain the library's implementation of the `LibConfiguration` interface. In the case of OWL API, the bridge will also contain a utility class that makes it easier to read and write to a stream.

To use the R2RML API with one of the supported libraries, the core module, the bridge and its corresponding library must be included. This design will help to meet the third implementation requirement, as only one bridge and library is needed for the API to work. If the user provides a custom implementation of the `LibConfiguration` interface, the bridge module will not be needed.

Figure 9.3 illustrates how the bridge module will make a library work with the core module. The bridge module is dependent on both the core module and the library (see Figure 9.4). The methods in the `R2RMLMappingManagerFactory` which retrieve a customized version of the API, will throw an exception if the library's bridge module is not present. To use the API with another library, the `LibConfiguration` interface must be implemented by a class and given as a parameter to the `getMappingFactory` method in `R2RMLMappingManagerFactory`.

Chapter 10

Supported Libraries

The R2RML API has to use an external library for some of the functionality it needs. For example, in order to serialize the mapping, the API needs RDF triples and resources. Several libraries already have implementations of these concepts. The API should therefore use one of these libraries instead of creating a new implementation. We decided that the API should support three widely used libraries. These are Jena, OpenRDF Sesame, and OWL API. These libraries will be described in this chapter.

As mentioned in Section 9.2 on page 56, the R2RML API needs classes from the external library that represent three abstractions. Which classes each library will use, will also be discussed in this chapter. The abstractions were:

- An RDF resource
- An RDF triple
- An RDF graph

10.1 Jena

Jena is a large open source Java API for programming Semantic Web applications. The work on Jena was started in 2000, by HP Labs [5]. It contains several libraries for developing applications using RDF, RDFA, RDFS, OWL, SPARQL and so on. It also has a triple store for persistent data storage, a SPARQL engine for querying RDF data and inference engines for both OWL and RDFS.

The RDF API can be seen as the core of Jena [4]. The main access point to the RDF API is the `ModelFactory`. This factory is used to create a `Model`. Models represent of a set of RDF statements, and contain many methods that can manipulate these statements. The model object also works as a factory for creating resources, literals and statements.

Serializing a model can be done directly from the model object. The model can write its RDF triples to either a `Writer` or an `OutputStream`. Reading

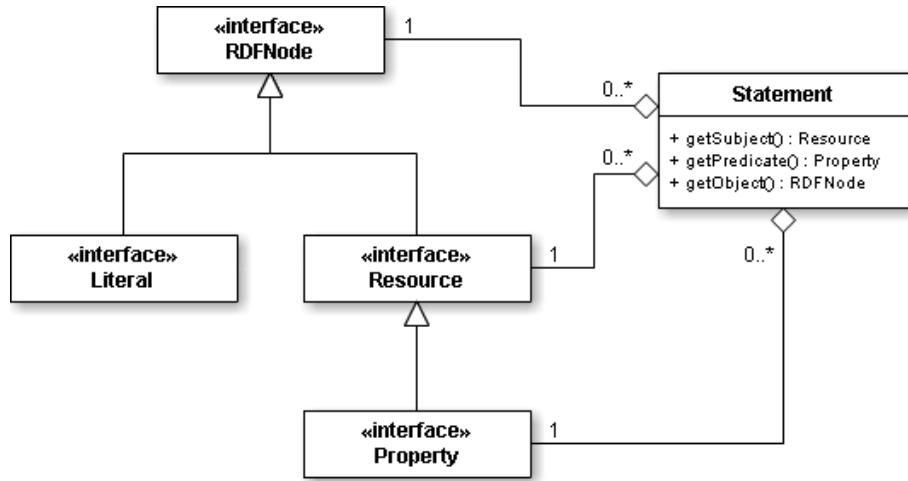


Figure 10.1: The structure of RDF statements in Jena.

can also be done by the model object. Jena is capable of processing several different RDF syntaxes.

The `ModelFactory` can be used to create several kinds of models. The main difference between the models, is how much reasoning that is performed on them. For example, an RDFS model performs RDFS reasoning on the statements, while an ontology model performs reasoning based on a given ontology model specification. An ontology model is useful to make OWL models.

As mentioned, the `Model` class consists of a set of statements. A statement contains a subject, a predicate and an object. In Jena, the `Resource` interface represents both URI resources and blank nodes, while the `Literal` interface represents literals. In addition, the `Property` interface represents RDF properties, which can not be blank nodes. `Property` is a subinterface of `Resource`. Because an RDF triples can contain both resources and literals in the object position, the `RDFNode` interface is a superinterface of both `Resource` and `Literal`. The class `Statement` represents an RDF triple. The subject of a statement is a `Resource`, the predicate is a `Property`, while the object is a `RDFNode`. See Figure 10.1 for an overview of this structure.

The Jena RDF API fits very well with the three abstractions that the R2RML API needs. The RDF resource will be represented by the `Resource` class, the RDF triple by the `Statement` class and the RDF graph by the `Model` class. The `Literal` class is not needed, because literals will be represented by strings.

10.2 Sesame

OpenRDF Sesame is another open source Java framework for programming with RDF data. It does not offer quite as much functionality as Jena,

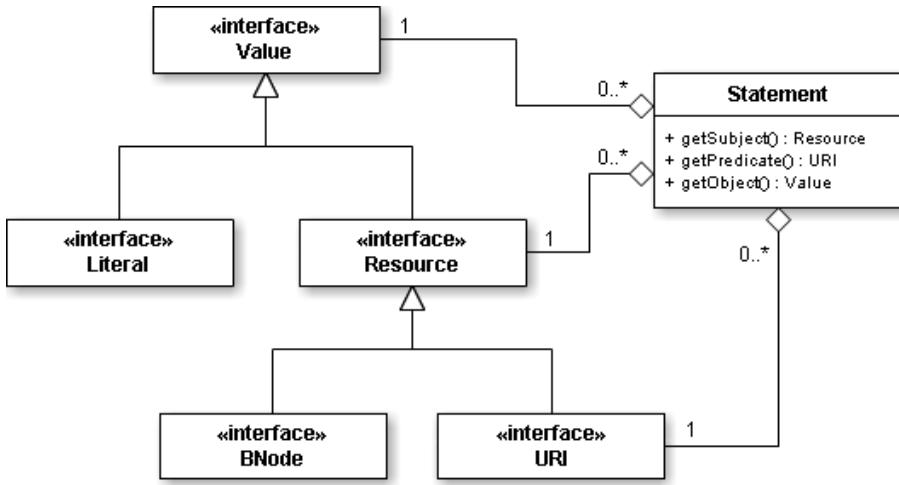


Figure 10.2: The structure of RDF statements in OpenRDF Sesame.

but it can be extended with several plugins that make it usable in a lot of different scenarios. Without any plugins, Sesame supports RDF and RDFS reasoning, as well as querying RDF with SPARQL and SeRQL (Sesame RDF Query Language). Sesame was originally developed by Aduna in connection to a research project called On-To-Knowledge [25]. It is now a community project with Aduna as a project leader.

As with Jena, the RDF library is an important part of Sesame. In Sesame, the resources and statements does not have to be associated with a model, but they can be. The `Model` class is simply a set of statements, which provides convenience methods handling namespaces, getting all resources in, for example, the object position of the statements, and so on.

The main access point to Sesame's RDF API, is the `ValueFactory` class. The `ValueFactory` is used to create resources and statements. The resource class structure in Sesame is very similar to Jena. The superinterface for all the RDF objects is called `Value`. This interface has the subinterfaces `Resource` and `Literal`. The resource interface has the subinterfaces `URI` and `BNode`. When creating a statement, the subject is a `Resource`, the predicate is an `URI`, and the object is a `Value`. See Figure 10.2 for an overview of this structure.

Like Jena, Sesame's RDF API fits well with the three abstractions that the R2RML API needs. Sesame will use `Resource` class for RDF resources, `Statement` for RDF triples and `Model` for RDF graphs.

10.3 OWL API

The OWL API is an open source Java API for creating and manipulating OWL ontologies. It is maintained at the University of Manchester, although it has received contributions from other groups and companies

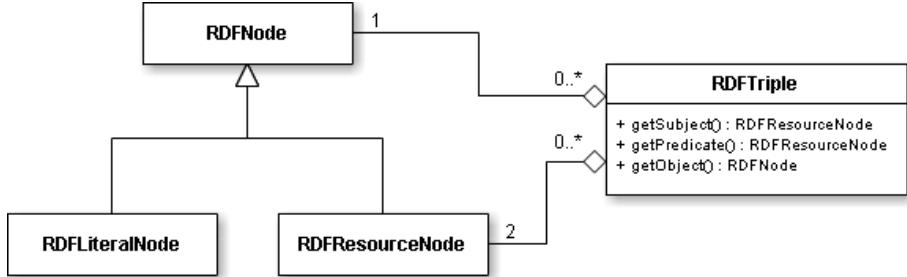


Figure 10.3: The structure of RDF statements in OWL API.

as well [27]. OWL API follows the OWL 2 structural specification closely. This specification therefore works well as documentation for the API, as well.

The API provides an axiom-centric view of the ontologies [23]. This means that the axioms are not at the level of RDF triples. Instead, the axioms are defined through interfaces that can be instantiated with the factory called `OWLDataFactory`. This approach makes it a lot easier to create axioms, because the RDF representation of an axiom may contain several triples that the user would need to handle individually. The main access point to OWL API, is the `OWLOntologyManager` class. This class can create an `OWLOntology`, which contains axioms.

The visitor pattern is used throughout the API, which separates the data structures from the functionality. The API has several visitor classes, that provides a lot of functionality for ontologies, axioms and other OWL objects. For example, the visitor `DLExpressivityChecker`, can find out which description logic the ontology is in. The visitor pattern also makes it easy for the user to extend the functionality, by creating new visitor classes.

OWL API focuses on creating and manipulating OWL ontologies, but this is not what is needed by the R2RML API. The R2RML API needs functionality to handle RDF. However, OWL API also contains some classes for this. The RDF part is used internally by OWL API in an intermediate step between the RDF serializations and the OWL ontology abstractions. The functionality around the RDF triples are therefore not as rich as it is in Jena and Sesame.

As can be seen in Figure 10.3, the superclass of the RDF objects is `RDFNode`. The `RDFResourceNode` class represents both named resources and blank nodes. In an `RDFTriple`, the `RDFResourceNode` class is used as the type of both subjects and predicates. Note that this also makes it possible to use blank nodes as a predicate.

The three classes that will be used by the R2RML API from OWL API will be `RDFResourceNode` for RDF resources, `RDFTriple` for RDF statements, and `Set<RDFTriple>` for an RDF model. OWL API also contains an `RDFGraph` class which could be used as the RDF model, but its functionality is not sufficient for what is needed by the R2RML API.

The R2RML API will also contain a utility class for OWL API. This is because the RDF part of OWL API is only meant for internal use. The utility class will make it easier for the user to perform some of the tasks that the RDF part of OWL API lacks functionality for. For example, it will contain methods to read and write RDF triples to a stream.

Chapter 11

R2RML API Implementation

This chapter will describe some of the details about the implementation of the R2RML API. The first section will describe some of the tools that have been used to create the API. The second section gives some code statistics, as well as describing how the API was tested. Finally, the third section will give some examples of how the API can be used.

11.1 Tools

The tools that have been used to create the R2RML API are mainly development tools for programming in Java. The *Eclipse* IDE was used for coding, while *Subversion* was used for version control.

11.1.1 Eclipse

Eclipse is an open-source integrated development environment (IDE), originally developed by IBM in 2001 [2]. It is now developed by the Eclipse Foundation, which is a non-profit organization. Eclipse is mainly made for development in Java. However, using various plug-ins makes it possible to develop software in other languages as well [13].

Eclipse itself consists of only a small run-time kernel. All the functionality of Eclipse comes from plug-ins. This plug-in architecture makes Eclipse very versatile and therefore useful in a lot of different development situations. A fresh install of Eclipse contains several plug-ins for Java development. Among these are plugins for debugging, advanced refactoring and more. Other plug-ins can easily be downloaded for free at the Eclipse Marketplace.

11.1.2 Subversion

Subversion (SVN) is an open-source software versioning system that is used to maintain current and older versions of source code, documentation and so

on. It was initially created by CollabNet Inc. in 2000, but is now developed by a global community of contributors [7].

Subversion’s vision is to “be universally recognized and adopted as an open-source, centralized version control system characterized by its reliability as a safe haven for valuable data; the simplicity of its model and usage; and its ability to support the needs of a wide variety of users and projects, from individuals to large-scale enterprise operations.” [6].

11.2 Implementation of the R2RML API

The first version of the API was made for the Optique project. As previously mentioned, it used the OpenRDF Sesame API to handle RDF. This was the starting point of the second version, which was going to be independent of any specific external library.

The development of the second version can be divided into two phases. The first was a testing phase, where the different approaches for making the API independent was tested (see Chapter 8 on page 39). A smaller subset of the API was implemented using the different approaches. In the second phase, the API was fully implemented using the approach that was chosen after reviewing the testing phase.

The chosen approach was to use class objects together with the library configuration (see Section 9.2 on page 56). This made the code using the API clear and concise, without increasing the complexity of the API too much. The final version of the API contains just over 3100 lines of code, not counting tests (counted using Cloc¹). It features in-memory mapping management, as well as parsing and serialization of mappings to RDF. The code that parses an RDF graph in the first version, was made by Timea Bagosi from the *Free University of Bolzano*. This university is a collaborator in the Optique project. In the second version of the API, this code was modified in order to make it independent of OpenRDF Sesame.

The API was tested with a set of test cases developed by Marco Ruzzi and Riccardo Mancini from *La Sapienza University of Rome*, who is also working in the Optique project. There are 36 test cases which were made to test the first version of the API. The tests are based on the R2RML test cases². The tests check that the API handles the different parts of an R2RML mapping correctly. In the second version, the tests were extended in order to test the new features. They were made in three versions, one for each of the supported libraries. The number of tests for the second version therefore tripled, to 108.

¹<http://cloc.sourceforge.net/>

²<http://www.w3.org/2001/sw/rdb2rdf/test-cases/>

```

1 @prefix rr: <http://www.w3.org/ns/r2rml#> .
2 @prefix foaf: <http://xmlns.com/foaf/0.1/> .
3 @base <http://example.com/base/> .
4
5 <TriplesMap1>
6   rr:logicalTable [ rr:tableName "Student" ; ] ;
7   rr:subjectMap [ rr:template "http://example.com/{Name}" ] ;
8   rr:predicateObjectMap
9     [
10       rr:predicate foaf:name ;
11       rr:objectMap [ rr:column "Name" ]
12     ] .

```

Listing 11.1: The mapping used in the code examples.

11.3 Code examples

This section will give some small code examples that will show how to use the API. All examples will use the example mapping given in Listing 11.1.

Mapping Creation

The example in Listing 11.2 on the next page, retrieves a mapping factory configured for Jena, and creates the example mapping. Note that in order to create the `PredicateMap` and `ObjectMap`, the term map type has to be specified. This is only needed for constant- and column valued term maps, because the value is a string. The `ResourceFactory` used on line 22 is a Jena class that creates Jena resources. All components except the `TriplesMap` will have a blank node as their resource.

Importing Mappings

The example in Listing 11.3, reads the test mapping from a file using Jena, and then imports the Jena model in order to get triples maps. The mapping is in the `testMap.ttl` file. The only thing that is needed from the R2RML API is to call the mapping manager's `importMappings` method (line 12).

Serializing Mappings

The example in Listing 11.4 on page 73, writes a collection of triples maps to a file using Jena. The variable `tm` used on line 7, is the `TriplesMap` that contains the mapping that was created in the first example. Note that the `exportMappings` method takes the class object of `Model` as a parameter. Writing the model to the file is done using Jena. It is also possible to serialize to a set of triples by calling the `serialize` method of the `TriplesMap`.

```

1 // Retrieve the MappingFactory configured for Jena.
2 R2RMLMappingManager mm =
3         R2RMLMappingManagerFactory.getJenaMappingManager();
4 MappingFactory mf = mm.getMappingFactory();
5
6 LogicalTable lt = mf.createSQLBaseTableOrView("Student");
7
8 Template t = mf.createTemplate("http://example.com/{Name}");
9 SubjectMap sm = mf.createSubjectMap(t);
10
11 PredicateMap pm = mf.createPredicateMap(
12             TermMapType.CONSTANT_VALUED,
13             "http://xmlns.com/foaf/0.1/name");
14 ObjectMap om = mf.createObjectMap(
15             TermMapType.COLUMN_VALUED,
16             "Name");
17 PredicateObjectMap pom = mf.createPredicateObjectMap(pm,om);
18
19 // Create the TriplesMap.
20 TriplesMap tm = mf.createTriplesMap(lt,sm,pom);
21 tm.setResource(
22             ResourceFactory.createResource(
23             "http://example.com/base/TriplesMap1"));

```

Listing 11.2: How to create a Jena mapping with the R2RML API.

```

1 // Get the R2RMLMappingManager configured for Jena.
2 R2RMLMappingManager mm =
3         R2RMLMappingManagerFactory.getJenaMappingManager();
4
5 FileInputStream fs =new FileInputStream(new File("testMap.ttl"));
6
7 // Read the file into a model.
8 Model m = ModelFactory.createDefaultModel();
9 m = m.read(fs, "http://example.com/base/", "TURTLE");
10
11 // Parse the model to get TriplesMaps.
12 Collection<TriplesMap> coll = mm.importMappings(m);

```

Listing 11.3: How to import a Jena mapping with the R2RML API.

```

1 // Get the R2RMLMappingManager configured for Jena.
2 R2RMLMappingManager mm =
3     R2RMLMappingManagerFactory.getJenaMappingManager();
4
5 // Create a collection and add the TriplesMaps.
6 Collection<TriplesMap> coll = new HashSet<TriplesMap>();
7 coll.add(tm);
8
9 // Export the mapping to a Jena model.
10 Model out = mm.exportMappings(coll, Model.class);
11
12 // Write the model to a file.
13 FileOutputStream fos =
14     new FileOutputStream(new File("testMap.ttl"));
15 out.write(fos, "TURTLE");

```

Listing 11.4: How to serialize a Jena mapping with the R2RML API.

Using a Custom Library

This example shows how the API can be extended in order to support additional RDF libraries. The `RDFConfiguration` class in Listing 11.5 on the next page, implements the `LibConfiguration` interface. This gives it the RDF functionality that is needed by the API. Listing 11.6, shows how to add the `RDFConfiguration` to the API. As can be seen, using a different library does not change much of the code when creating mappings. However, some get-methods need a class object as a parameter. This class object has to be changed when the underlying library changes.

```

1  class RDFConfiguration implements LibConfiguration {
2
3      Resource createResource( String uri){
4          return new Resource(uri);
5      }
6
7      Resource createBNode(){
8          return new Resource();
9      }
10
11     // And so on.
12     ...
13
14 }
```

Listing 11.5: An example implementation of the `LibConfiguration` interface.

```

1 // Get the R2RMLMappingManager configured for the custom library.
2 R2RMLMappingManager mm =
3         R2RMLMappingManagerFactory
4             .getMappingManager(new RDFConfiguration());
5
6 MappingFactory mf = mm.getMappingFactory();
7
8 // The R2RML components can now be created as normal.
9 LogicalTable lt = mf.createSQLBaseTableOrView("Student");
10
11 // Some get-methods need a class object.
12 Resource r = lt.getResource(Resource.class);
```

Listing 11.6: Using a custom RDF library with the R2RML API.

Chapter 12

Evaluation and Conclusion

This thesis has discussed how to make an API independent of its dependencies. We want the R2RML API to be a de-facto standard for R2RML mapping management. This means that the API has to be used by many developers. Having the API dependent on a specific library will force the users to use that library. We hope that more developers will use the API by allowing it to be used with an arbitrary RDF library. The user will be able to choose which underlying library the API will use.

Chapter 8 discussed several approaches which could make the API independent. The approaches were tested on a subset of the API, in order to find their advantages and disadvantages. The approach should be robust, while still being as easy to use as the dependent version. The approach should not increase the complexity of the API too much.

There are two problems the approaches have to solve. Firstly, the approach has to make the API able to use functionality from the library, without knowing the details about it. Secondly, the API must be able to store and use objects of classes from the library, without knowing their types.

The first approach that was discussed was *dependency injection* (see Section 8.1 on page 41). The approach decouples an object from its dependencies. The object will instead be dependent on abstractions. However, this approach was not very useful in our case because the dependencies will be dependent on the abstractions as well. The pre-made libraries does not depend on any such shared abstraction. In addition, dependency injection does not concern how the API can deal with unknown classes from the underlying library.

The next approach was using a *library configuration*. The library configuration is an interface that must be implemented using the library that will be supported. This interface will be injected into the API in order to give it the functionality it needs. This approach does not give the external libraries any new dependencies.

The robustness of the library configuration approach relies on how the

library configuration is defined. If the user wants to extend the API to support another library, the methods of the library configuration must be easy to understand. The usability of the API is not affected much when using the library configuration. By using a factory that retrieves a customized version of the API, the user does not need to deal with the library configuration directly.

Using the library configuration solves the first problem. It makes the API able to use functionality from the library without knowing the details about it. However, it does not solve the second problem. We therefore need another approach that can work together with the library configuration.

Two approaches were discussed that could solve this problem: *Java Generics* and *class objects*. Both approaches solve the second problem. Both also have advantages and disadvantages that need to be considered. Using C++ templates were also discussed, but it is not applicable in our case.

Using Java Generics gives the best robustness. It provides full compile-time type safety. All classes that are needed by the API will get a type parameter that can be checked at compile time. However, the usability is severely affected. The user will have to add type parameters to all declarations of objects from the API. If the API is using many type parameters, the code will become very verbose and untidy. This will also make the code harder to read. Even though the user may only need one class from the external library, all classes will be exposed through the type parameters. The maintainability of the code is not affected. All classes will get type parameters, but their names will make it easy to know what they are.

Using class objects have a much better effect on the usability. The code using the API will look very similar to the dependent API version. The only difference is that some get-methods will get a class parameter. However, the robustness will be affected because some of the compile-time type checking is moved to the run-time. Maintainability is also affected, because the objects that have a type from the library will now have the `Object` type in the API code. There is no way to know what they are by just looking at the type. It is therefore important to have a thorough documentation of what the type of each object is.

12.1 Conclusion

Section 1.2 on page 2 listed three contributions that this thesis was going to make. These were:

1. Design an R2RML mapping management API in Java.
2. Discuss possibilities for making an API independent of its dependencies.
3. Apply the approach to the R2RML API.

The first and third goal were discussed in Chapter 9. The first goal was met with the first version of the API, while the third goal was met with the second version. We are hoping that the API will establish itself as a de-facto standard R2RML mapping management API, but this remains to be seen.

The third goal is important, as it will help the API reach the status of a de-facto standard API. The first version of the API was dependent on the OpenRDF Sesame API. The second version of the API was made using the approach that was established for the second goal. Because of the connection to the Optique project, the new version of the API has already seen much use.

The second goal was discussed in Chapter 8. We have seen that a combination of several approaches is needed to make an API independent of its dependencies. For the R2RML API, we decided to use the library configuration together with class objects in the final version. This made the code using the API concise and readable. Some of the type checking is performed at run-time, but it will still be hard to make any serious mistakes.

The library configuration interface makes it possible for the API to use an external library without knowing the details of what it does. Using the library configuration is therefore essential to accomplish what we needed. The API will be dependent on the library configuration, but not the underlying library.

Using Java Generics might be worth it if the API only needs one class from the external library. If this is the case, there will only be one type parameter. This will not be that much of a problem for the user. If there are more type parameters, the code will quickly become very verbose. In that case, using class objects are a good choice.

12.2 Future Work

The current version of the R2RML API is complete with regards to the scope of this work. The current API features in-memory mapping management, as well as parsing and serializing mappings to RDF.

The Optique project may have a need for more functionality in the future. Finding out what this might be, is a work in progress. For example, there might be a need for methods that can do more advanced refactoring of the mappings, or functionality for performing more advanced operations on the mappings. For example, joining two mappings that work on the same logical table.

Another aspect that might be improved, is SQL queries. In the current version of the API, the SQL queries are represented as strings. This allows only simple string manipulation of the queries. SQL queries are structured objects, and can therefore be represented by a class. This would require an

additional external library. Because the API should stay independent of any external library, this might become a big task to implement.

Appendix A

R2RML API Code and Documentation

A zip file containing the full R2RML project can be downloaded from this URL: <http://folk.uio.no/marstran/R2RMLAPI.zip>. The **Jar Files** directory contains the core and bridge jar files that is needed to use the API. The test projects contain all the tests using the three supported libraries. In order to run the Sesame and Jena tests, the library has to be downloaded separately. The jar file for OWL API is included. The main project is in the **R2RML API** directory.

The javadoc documentation of the API can be found by following this URL: <http://folk.uio.no/marstran/doc/>.

Bibliography

- [1] *About SNOMED CT*. 18th Mar. 2014. URL: <http://www.ihtsdo.org/snomed-ct/snomed-ct0/> (visited on 19/03/2014).
- [2] *About the Eclipse Foundation*. URL: <https://www.eclipse.org/org/> (visited on 09/04/2014).
- [3] Christopher Alexander et al. *A Pattern Language*. New York: Oxford University Press, 1977.
- [4] *Apache Jena - Jena architecture overview*. URL: https://jena.apache.org/about_jena/architecture.html (visited on 08/04/2014).
- [5] *Apache Jena - What is Jena?* URL: https://jena.apache.org/about_jena/about.html (visited on 08/04/2014).
- [6] *Apache Subversion*. URL: <http://subversion.apache.org/> (visited on 09/04/2014).
- [7] *Apache Subversion - Wikipedia*. URL: http://en.wikipedia.org/wiki/Apache_Subversion (visited on 09/04/2014).
- [8] Marcelo Arenas et al., eds. *A Direct Mapping of Relational Data to RDF*. 27th Sept. 2012. URL: <http://www.w3.org/TR/2012/REC-rdb-direct-mapping-20120927/>.
- [9] Sören Auer et al. *Use Cases and Requirements for Mapping Relational Databases to RDF*. Ed. by Eric Prud'hommeaux and Michael Hausenblas. 8th June 2010. URL: <http://www.w3.org/TR/2010/WD-rdb2rdf-ucr-20100608/>.
- [10] Christian Bizer and Richard Cyganiak. *D2RQ — Lessons Learned*. 8th Sept. 2007. URL: <http://www.w3.org/2007/03/RdfRDB/papers/d2rq-positionpaper/>.
- [11] Krzysztof Czarnecki and Ulrich W. Eisenecker. ‘Generative Programming: Methods, Tools, and Applications’. In: Addison-Wesley, 2002, pp. 243–245.
- [12] Souripriya Das, Seema Sundara and Richard Cyganiak, eds. *R2RML: RDB to RDF Mapping Language*. 27th Sept. 2012. URL: <http://www.w3.org/TR/2012/REC-r2rml-20120927/>.
- [13] *Eclipse (software) - Wikipedia*. URL: [http://en.wikipedia.org/wiki/Eclipse_\(software\)](http://en.wikipedia.org/wiki/Eclipse_(software)) (visited on 09/04/2014).

- [14] FOAF Vocabulary Specification 0.99. 14th Jan. 2014. URL: <http://xmlns.com/foaf/spec/20140114.html> (visited on 19/03/2014).
- [15] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1994. ISBN: 0-201-63361-2.
- [16] Martin Giese et al. ‘Scalable End-User Access to Big Data’. In: *Big Data Computing*. Ed. by Rajendra Akerkar. CRC Press, 2013.
- [17] Ivan Herman, ed. *RDB2RDF Working Group Charter*. 18th Nov. 2011. URL: <http://www.w3.org/2011/10/rdb2rdf-charter.html>.
- [18] Matthias Hert, Gerald Reif and Harald C. Gall. ‘A Comparison of RDB-to-RDF Mapping Languages’. In: *Proceedings of the 7th International Conference on Semantic Systems*. I-Semantics ’11. Graz, Austria: ACM, 2011, pp. 25–32. ISBN: 978-1-4503-0621-8. DOI: 10.1145/2063518.2063522. URL: <http://doi.acm.org/10.1145/2063518.2063522>.
- [19] *Inversion of Control Containers and the Dependency Injection pattern*. 4th Jan. 2004. URL: <http://www.martinfowler.com/articles/injection.html> (visited on 04/04/2014).
- [20] *Java SE 7 Collections-Related APIs and Developer Guides*. URL: <http://www.w3.org/TR/2012/REC-r2rml-20120927/> (visited on 11/04/2014).
- [21] *JDK 5.0 Java Programming Language*. 2004. URL: <http://docs.oracle.com/javase/1.5.0/docs/guide/language/index.html> (visited on 01/04/2014).
- [22] Ashok Malhotrat, ed. *W3C RDB2RDF Incubator Group Report*. 26th Jan. 2009. URL: <http://www.w3.org/2005/Incubator/rdb2rdf/XGR-rdb2rdf-20090126/>.
- [23] Sean Bechhofer Matthew Horridge. ‘The OWL API: A Java API for OWL Ontologies’. In: *Semantic Web Journal* 2(1) (2011), pp. 11–21.
- [24] Boris Motik et al., eds. *OWL 2 Web Ontology Language Profiles (Second Edition)*. 11th Dec. 2012. URL: <http://www.w3.org/TR/owl2-profiles/> (visited on 19/03/2014).
- [25] *openRDF.org: About*. URL: <http://www.openrdf.org/about.jsp> (visited on 08/04/2014).
- [26] *Optique - Scalable End-user Access to Big Data*. URL: <http://www.optique-project.eu> (visited on 19/03/2014).
- [27] *OWL API*. URL: <http://owlapi.sourceforge.net/> (visited on 08/04/2014).
- [28] Antonella Poggi et al. ‘Linking Data to Ontologies’. In: *Journal on Data Semantics X* (2008), pp. 133–173.
- [29] *Project Description / Optique*. URL: <http://www.optique-project.eu/about-optique/about-optique/> (visited on 28/03/2014).

- [30] *Research Topics / Optique*. URL: <http://www.optique-project.eu/about-optique/research-topics-2/> (visited on 28/03/2014).
- [31] Satya S. Sahoo et al. *A Survey of Current Approaches for Mapping of Relational Databases to RDF*. 8th Jan. 2009. URL: http://www.w3.org/2005/Incubator/rdb2rdf/RDB2RDF_SurveyReport.pdf.
- [32] *The D2RQ Mapping Language*. 12th Mar. 2012. URL: <http://d2rq.org/d2rq-language> (visited on 19/03/2014).
- [33] *The D2RQ Platform - Accessing Relational Databases as Virtual RDF Graphs*. URL: <http://d2rq.org/> (visited on 01/05/2014).
- [34] *View Processing and Update - Presentation*. URL: osm.cs.byu.edu/CS452/supplements/ViewUpdate.ppt (visited on 01/05/2014).
- [35] *Wikipedia: Ontology*. 2nd Mar. 2014. URL: <http://en.wikipedia.org/wiki/Ontology> (visited on 19/03/2014).
- [36] *Wikipedia: Ontology components*. 11th Sept. 2013. URL: http://en.wikipedia.org/wiki/Ontology_components (visited on 19/03/2014).
- [37] *Wikipedia: Ontology (information science)*. 15th Mar. 2014. URL: [http://en.wikipedia.org/wiki/Ontology_\(information_science\)](http://en.wikipedia.org/wiki/Ontology_(information_science)) (visited on 19/03/2014).