University of Twente
Department of Electrical Engineering, Mathematics and Computer Science
Database group

Object XML Mappings

Joost Diepenmaat
M.Sc. Thesis
July 9, 2009

Graduating committee:
Dr. ir. Maurice van Keulen
Ing. J. Flokstra
Dr. ir. Ander de Keijzer

Simplicity is the ultimate sophistication.
— Leonardo da Vinci

Acknowledgments

This thesis would not have been possible without a lot of help and support from the people around me. I would like to thank the following people for their contributions and encouragement:

Maurice van Keulen, Jan Flokstra, Ander de Keijzer and all the great people from the database group at the University of Twente who gave me great feedback and support.

A big "thank you" to the guys at work, Berend van Bruijnsvoort and Edwin Vlieg, who stimulated me, gave great feedback and gave me enough time to get this thesis done.

My parents who always gave me all the essentials to find my passion. After health and love, passion is the most important in life, and the best ingredient to success.

And of course my biggest supporter in all of this was my girlfriend, Anneke, who was always motivating me and a great source of inspiration.

It has been more than a pleasure to work on this thesis.

Thank you all,

Joost Diepenmaat

Abstract

Lots of things changed during the last decade in the behavior and needs of enterprise computer systems. The complexity of systems increased, and therefore the task of building systems gets harder. Many enterprises store their complex data in the popular XML format. This data usually grows both in structure and amount, and gets more and more difficult to handle within software. This thesis explores a new method that makes complex XML data more tractable for developers by using a well understood domain language.

We propose an Object XML Mapping mechanism that allows us to map complex structured XML data onto Ruby objects using minimal programming macros. These mapped objects can also contain additional business logic. The Object XML Mapping allows developers to work with complex XML data, in a better understandable way.

The Object XML Mapping we propose is based on Pathfinder's[23] relational tree encoding algorithm. This range encoding has proven its applicability for large-scale XML processing as it servers the backbone of the open-source XQuery implementation MonetDB/XQuery[9].

First, we introduce a mapping mechanism that maps macros to a class level *object map*. This *object map* defines how XML data is mapped onto the object. Second, we introduce an object instance storage mechanism that merges the *object map* with XML data to an instance level *mapped nodes table*. This *mapped nodes table* holds the XML nodes in Pathfinder's relational tree encoding to allow persistent storage straight from object to a relational database. Finally, we introduce a dynamic programming method that can access XML data from the *mapped nodes table*.

A proof of concept demonstrates that we can map a relative complex XML structure in minimal steps. Performance measurements show that the Object XML Mapping mechanism needs minimal system resources.

Contents

Chapter 1

Introduction

The past has proven that building computer systems is hard. As the complexity of the system increases, the task of building the system may get exponentially harder. Lots of things changed during the last decade in the behavior and needs of our users. Nowadays, we are producing huge amounts of data, in thousands of languages, in dozens of formats, etc. All this complex data has to be accessed and interpreted with software written by humans. As we all can imagine this is an increasing difficult job. This thesis explores a new method that makes complex data more tractable by using objects in a well understood domain language. We design a system that enables us to work with complex XML data, in a better understandable way.

We define scope of the thesis and provide some background information in this chapter.

1.1 Enterprise applications

There are several distinct kinds of software in the industry, that come with their own challenges and complexity. This thesis targets enterprise applications that have complex data and comprehensive business rules. Enterprise applications include CRM data, patient records, supply chain information, e-commerce systems, office document storage, etc. It does not include systems such as alarm systems, router software, operating systems, games, machine robot software, etc.

Enterprise applications usually have persistent data that needs to resist for a long time. It may need to be restructured or extended, as new details need to be added. It almost often involves a large data set, that is continuously growing. Enterprise applications have often many concurrent users, that access the data simultaneously, using a variety of user interfaces. Other applications may also want to integrate with the enterprise application, to access the data. A large amount of business logic is accompanied by this data, where the business logic is changing frequently due to political or strategic decisions.

1.2 Layers and domain logic

One of the most common techniques that software engineers use to break complicated systems into understandable parts is layering. The strongest benefit of layering is that it gives us the ability to work on a specific part of a system without knowing much about other layers; focus on something, forget about the rest for a while. A common example of this is that a software engineer can understand how to build a POP3 service without having to understand TCP/IP. Each layer hides its lower layer and is unaware of its higher layer. Layers make previous work reusable for higher layers.

The three primary layers[25] as in table 1.1 are widely used in Enterprise systems. Despite the fact that layers deliver impressive clarity to the system, they can also cause performance issues. The art of defining layers requires a good feeling for proper separation and organization of elements in the system. The method proposed in this thesis lives in the domain layer of the three primary layers, but also has some major boundaries with the data layer.

Layer	Responsibilities
Presentation	Handles the interaction between the user and the software
Domain	Logic that is the core element of the system
Data	Communicates with other systems that carry out tasks, such as databases, messaging systems, other packages

Table 1.1: Primary layers of an enterprise system

1.2.1 Ruby on Rails and Active Record

> *Ruby on Rails is an open-source web framework that's optimized for programmer happiness and sustainable productivity. It lets you write beautiful code by favoring convention over configuration.*
> — *www.rubyonrails.org*

Ruby on Rails (RoR) [13] gives developers a pure Ruby framework for developing web-applications implementing the Model-View-Control pattern [29]. RoR adopts domain mappings in its Active Record library using a wide flexible variety of SQL databases as its main data source. These databases include DB2, MySQL, Oracle, Postgresql, SQLite, SQL Server and many others. Active Record is a good example of an Object Relational Mapping that brings clarity to the system. With minimal amount of code needed, developers can build a real-world domain model. These domain models make developers often understand and control complex situations better.

1.3 Enterprise applications and XML documents

Many years ago, researchers addressed the importance of standardization of documents for interoperability and consistent storage in enterprise applications. Nowadays, an increasing number of enterprise applications is using a standardized document format. This standardized data is often stored in the XML format. XML data is human readable, but due to the complexity also fine-granular. This makes XML documents difficult to understand, and complex to handle within software. Although considerable research effort has been conducted to store and query XML documents efficiently, software development to handle XML documents within the application domain is still a time consuming job.

We use XML documents as the base language for document storage in this thesis.

1.4 XML in the domain layer

XML comes with XPath[19] as a language for addressing parts in XML documents. XPath queries enable us to get certain specific parts out of a large XML document, using a compact syntax. Despite XPath's widespread use in software, there are still plenty of difficulties in using it within your application. Many application developers do not understand XPath well enough and, as a result, have problems defining effective XPath expressions.

Layer	Responsibilities
Presentation	HTML interfaces that handle the interaction between user and system
Domain	Object XML Mapping that contains the core elements of the system
Data	XML database that stores XML data

Table 1.2: Abstract overview of components

It would be easier to access XML data using mechanisms that fit in the application development language better. Object Relational Mappings/Domain Models[26] currently support this for the relational SQL world. As they manage interconnected objects, where each object represents something meaningful

within the application domain. A Domain Model in an application contains a whole layer of objects that model the business. These objects mimic the data in the business and/or capture the business rules.

Table 1.2 gives an overview on how we can apply the object relational mapping concepts from the "SQL world" on XML data.

1.5 Problem statement

The problem we address in this thesis is that current ORM systems do not support XML documents. Therefore we need to create a mechanism that separates XPath expressions from the domain logic and places them in separate classes. These classes form a gateway to the data source. The thesis studies the possibilities of an Object XML Mapping with XML/XPath as a base language for persistent storage. We explore and describe the fundamentals of Object XML Mappings.

1.6 Research questions

The following research questions can be derived from the problem statement:

How can we implement XML based Object Mapping?

1. *How can we implement Pathfinder's[23] relational tree encoding algorithm in a Ruby on Rails plugin, such that it supports Postgresql?*
2. *How can we implement an Object XML Mapping on Pathfinder's[23] relational tree encoding?* The current ORM patterns are SQL based. In this step we define an Object Mapping for XML data. This includes intuitive design of basic features such as mapping, associations and direct manipulation.

1.7 Scope

This project will focus on support for Object XML Mappings based on Pathfinder[23] within the Ruby on Rails framework. Some parts of research can result in unforeseen effort, therefore we have the following boundaries within this project:

- The research will initially focus on Postgresql support driven by the Sequel database toolkit [15], other databases are out of scope.
- A working prototype (Ruby on Rails plug-in) will be built as a proof of concept.
- In depth performance and efficiency tuning are currently not the most important issues. The focus is on a proof of concept and basic performance tuning that enables us to work on small up to medium size data sets.

1.8 Research method

This project will answer the research questions by making several iterative steps. At each step, the support for Object XML Mappings will be extended or improved. Each step will contain the following sub steps:

- Literature research on Pathfinder backgrounds, existing mapping methods and ORM techniques.
- Define requirements and design an Object XML Mapping mechanism.
- Examine whether the Object XML Mapping approach is feasible in terms of performance and delivers the developer productivity wins.

Chapter 2

Background and Related Work

2.1 Using Pathfinder for XML storage using a RDBMS

As XML standards within enterprises are being adapted more frequently, the need for efficient XML processing has increased tremendously. Both standards XPath[19] and XQuery[20] can select bits of data from XML documents and are widely used within enterprise software.

Research[24] discovered the high potential of suitable tree encodings that turn relational database back-ends into high efficient XPath processors such as Pathfinder[23]. With a rock solid RDBMS implementation, the effort to implement XPath processing mechanisms has become minimal, while the scalability excels with multi-gigabyte XML instances. This enables us to handle XML documents faster with a higher volume data-set.

Recent work[27] showed us that an off-the-shelf RDBMS can perform very well, without modifying the database kernel for tree-awareness in the relational system such as previously required with techniques such as the `Staircase Join`[28]. This opens the gates to integrate the Pathfinder tree-encoding algorithm within any unmodified traditional RDBMS in any enterprise system. We based our research on the Pathfinder relational tree encoding, as it has proven to be high scalable and can be used on an existing RDMBS.

2.1.1 Zooming in on relational tree encodings

It is fundamental to understand the principals behind Pathfinder's relational tree encoding algorithm that we have based our design on. The encoding shreds each XML document node into a relational-database tuple. For each node v a range that hosts all nodes v` is recorded in the subtree below v. The internal architecture of the relational tree encoding heavily depends on the SAX parser model. We enumerate each with the following properties:

- Node v's pre-order rank *pre (v)* according to the XML document order.
- Node v's *size (v)* as v's number of descendant nodes.
- Node v's distance from the tree's root *level (v)*.

During enumeration we also add two properties to the relational-database tuple:

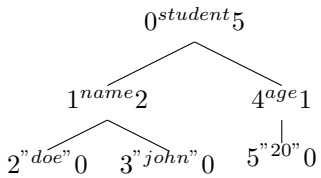
- Node *kind (v)* $\in$ { `elem`, `attr`, `text`, . . . }.
- Node *name (v)* holding v's tag name or textual context.

The tree and table in 2.1 illustrate this encoding for the XML fragment 2.1

Listing 2.1: XML fragment for student John Doe

```
1 <student>
2     <name last="Doe" first="John" />
3     <age>20</age>
4 </student>
```

This range encoding has proven its applicability for large-scale XML processing as it servers the backbone of the open-source XQuery implementation MonetDB/XQuery.



pre	size	level	kind	value	name
0	5	0	elem		student
1	2	1	elem		name
2	0	2	attr	Doe	last
3	0	2	attr	John	first
4	1	1	elem		age
5	0	2	text	20	

Table 2.1: Encoded nodes in tree and table

2.1.2 XPath versus XQuery

XPath has become a popular language that provides simplicity for extracting specific bits of data from an XML document. XQuery has additional power and flexibility by providing support for queries that express more complex record-selection, transforming results or even recursion.

XPath

Support for XPath is becoming an important part in many programming languages. That is because XPath is, for example, the easiest way to create a list of e-mail addresses from a subset of student records in an XML document. XPath can save developers a considerable amount of time and effort. Even individuals that never dealt with XPath before can quickly harness its power with minimal programming skills.

In most cases, XPath expressions are not only concise, but also very readable. Even developers that do not know XPath well, are able to understand most expressions. The language understands the (complex) node structure in an XML document and, more importantly, the relationships among those nodes. Besides elements and element values, XPath also understands attributes, namespaces, processing instructions and so on.

In many cases it seems hard to express XML queries cleaner and conciser than with XPath. Serious limitations for XPath arise when it comes to more complicated queries, that do not fit in XPath's nature. The XQuery language overcomes many of these limitations by introducting a little more compexity.

XQuery

XQuery supports the complete XPath language natively, as part of its own syntax. But it can also be considered as a general programming language with a far more advanced expression syntax. It ties together XML, XPath, functions and special expressions in its functional language.

XQuery and XPath are both ideal to express a query to obtain data from an XML document. There is strong a similarity with SQL from the relational-database world. But XQuery provides additional functionality for expressing arbitrary transformations of the result set. This enables developers to transform results directly from XML into HTML, CSV, SQL or any text-based format. Data transformations with XQuery are often far easier and faster than in Java or Ruby or any other programming language.

XPath does enough

Object XML Mappings, as discussed in the next chapter, essentially map XML result sets to objects that can be used within the programming language. Both XPath and XQuery produce XML results that can be used for the mapping. Therefore we decided to focus on basic XPath support.

2.1.3 Managing semantics and business logic with XML schema

The XML schema[21] specification describes the type of an XML document in terms of constraints on the structure and content. The XML document itself can verify whether it validates on the semantics described by the designer of an XML standard. XML schemas are almost always specified when an XML standard is defined. Validations on schemas take place when a document is stored, modified or when it is received from another interconnecting enterprise system or application.

As XML schemas are able to validate semantics, they are often used to manage business logic decisions partly. For example, an XML schema can be used to validate that a `student` in an XML document should always have `firstname` and a `lastname` element. But it fails to implement a simple business rule such as: `"The supervisor receives an e-mail with the student details, when a new student joins the university."`. This is where an Object Relational Mapping joins the game.

2.2 Object Relational Mappings

Enterprise applications, with their large continuous growing data-sets, have major code components that serialize and persist the object data in a relational database. An ORM (object-relational mapping) framework is often the easiest solution to reduce the amount of database-access code that needs to be written and increases developer productivity. It allows the developer to declaratively define the mapping between the object model and database schema. In addition to the basic mapping of columns from the table schema, objects in the ORM can maintain complex associations, inheritance, validation rules and much more. A sophisticated ORM can also express database-access operations in terms of objects and manages business logic.

Several ORM frameworks are in use today. For example, the Hibernate[7], the OpenJPA[3] and TopLink[11] frameworks are populair with Java developers, and NHibernate[10] is used by many .NET developers.

An ORM for Ruby[14], that recently received a lot of attention from enterprise web-developers is Active Record[1], included in the Ruby on Rails[13] web-development framework. Active Record in Ruby on Rails is named after the Active Record design pattern:

> *Active Record: An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data.*
> *— Martin Fowler, in [26]*

2.2.1 Mapping to relational databases with Active Record

Imagine a database table `students` with two columns `name` (string type) and `age` (integer type). There is a `Student` class implementing the Active Record pattern. The following code:

```
1 student = Student.new
2 student.name = "John"
3 student.age = 25
4 student.save
```

This will create a new row in the `students` table with the given values and is equivalent to the SQL command:

```
1 INSERT INTO students (name, age) VALUES ('John', 25);
```

A class can be used to access a database as well:

```
1 student = Student.find_by_name("John")
```

This will query the `students` table for all students named `John`:

```
SELECT * FROM students WHERE name = "John";
```

Another typical example of Active Record objects are associations such as:

```
courses = Student.find(:first, :order => "age desc").courses
```

This will return all the `courses` taken by the oldest `student` assuming you specified "age" in the `student` model.

2.2.2 Managing business logic with Active Record

At first glance Active Record looks like an efficient way for transferring your data from object to database. But it can do more: Embedding business logic into the business objects you have created. Imagine the following three very simple business rules:

1. The `name` and `age` should always be present for a student.
2. The `age` of a student should always be a number greater than 14 and less than 60 years old.
3. The supervisor receives an e-mail with the student details, when a new `student` joins the university.

These three rules can be embedded in your Active Record object with simple validation rules.

Listing 2.2: Validation rules in ActiveRecord

```
class Student < ActiveRecord::Base
    validates_presence_of :name, :age
    validates_numericality_of :age,
        :greater_than => 14, :less_than => 60
    after_create { |student| Mailer.deliver_subscription(student)}
end
```

The few lines in listing 2.2 guarantee the business logic over all the `Student` instances within the system. Active Record comes with a nice set of build-in functions that enables developers to manage business logic easier than before.

2.3 Dynamic programming in Ruby

Dynamic programming languages allow runtime creation of new programming elements. Dynamic features within a language such as Ruby enable developers to do things that are impossible in a static language such as Java. Runtime properties and methods can be created for class instances. This technique[30] has proven to be very useful in traditional ORM frameworks such as Active Record and GORM [5]. Ruby supports dynamic programming in its language nature very well.

An important technique in dynamic programming, for both Ruby and GORM, is `method missing` invocation. This method is called at runtime when the application attempts to access the undefined property or method from an object instance. By catching these access calls, `method missing` enables an object to behave like if the property or method exists.

The prototype in this thesis strongly depends on dynamic programming techniques. Wide experience with Ruby and natural support for dynamic programming makes Ruby the language of our choice for this thesis.

2.4 Attempts

ORM frameworks are very popular and used over wide spectrum in the domain. Of course, integrating XML facilities within ORM frameworks is nothing new. Many of the ORM frameworks[1, 3, 7, 10, 11] have facilities to import XML files to their domain objects. Nevertheless, they do not support storage in native XML databases at all.

The major difference between previous work and this thesis is clear. This research will shine its light on how to adapt the power of the Pathfinder[23] XML encoding algorithm, with persistent storage support, into an Object XML Mapping. Hereby we leave the relation-database world and fully depend on high scalable native XML storage.

2.4.1 Object mappers in Ruby, DOM versus SAX

Various XML to object mappers exist for Ruby such as ROXML[12] and HappyMapper[6]. Both libraries do a great job when it comes to simple mappings from XML to Ruby objects. Nevertheless, storing XML data and XML specific key features are not available.

A lot of work during the mapping process is for [12, 6] is done using the LibXML DOM parser. DOM parsers consume a lot of memory, due to their own object model, and are relative slow comparing to SAX parsers that implement an own object model. As Pathfinder's[23] encoding algorithm heavily depends on SAX parsing strategies, we will move to SAX and show its potential for XML to object mappings.

Chapter 3

Object XML Mappings

3.1 The nature of complex XML data

Over the past few years, XML has become the new standard for data transmission. As more enterprises begin to enforce XML standards for all kind of purposes, increasingly complex formats are emerging. More in-depth reasoning makes us understand this problem better.

3.1.1 Evolution

Complex XML standards such as [4, 17] take years to accomplish. After dozens of validations, testing and adoption rounds, the standards are put in the wild. At this point, it is not guaranteed that a certain XML standard is the right choice for an enterprise. The adaption of a new XML standard can take years. It also can fail for many, often political, reasons.

The world around us changes quickly. Over the years, we have seen many evolutions in software applications and communication methods. Often, this change has to be reflected within enterprise systems by changing business logic or adapting more intelligent storage structures. Standards, and historic data needs to become more intelligent and extended with new facilities over time. Hence, being able to adapt difficult, and fast changing, standards easier is a main objective for Object XML Mappings.

3.1.2 Technical aspects

The objective of XML is to support data exchange. Nowadays most large database vendors have released their own native XML database. But relational databases are still mainstream as they excel in fast data storage. The main reason we store XML in native XML databases is the complexity of decomposing XML.

Complex XML formats can include multiple namespaces, recursive definitions and contain a huge set of possible elements. It has become increasingly challenging to manage the documents that grow in size and complexity. The world has adopted XML. But, as developers, we face difficult and complex performance problems to handle XML in high volume every day.

3.2 The big box that makes LEGO complex

LEGO bricks and XML documents share similar properties. We found a metaphor that explains us why.

The number of LEGO bricks that someone collects increases over the years, as people buy more packages LEGO. Most people start decomposing LEGO bricks, and put them in a large box, after a while. Those bricks in a large box become a more and more complex data-set over time.

When you buy a small box of LEGO, the package contains a bag of bricks and an instruction manual. Anyone with a bit of knowledge can complete the building within a short amount of time. Nevertheless, mixing the few bricks from the small box into a full large box with a huge amount of bricks, before starting to build the package, will make it a lot more difficult to accomplish the same goal. Table 3.1 describes the similarities between LEGO and XML.

LEGO	XML
Box LEGO	XML document
LEGO brick	XML element
Increasing number of bricks	Increasing number of XML documents
More variety in bricks	New variables in XML standards

Table 3.1: Similarities between LEGO and XML

Sorting helps

LEGO builders often use containers to sort their bricks in a desired sorting model. There are several conditions we can sort on such as size, color, category or by part. All sorting mechanisms have their downsides, none of them is perfect. But at least it will make the LEGO builder more agile in his work.

The function of LEGO sorting containers is a metaphor of Object XML Mappings, such that both suppose to make development of a systems easier.

3.3 Key characteristics of Object XML Mappings

We introduce a variation on the implementation of the Object Relational Mapping pattern. An Object XML Mapping connects XML data and business logic in one wrapping. In the first phase we are concerned about basic functionalities as described in the following paragraphs.

3.3.1 Automated mapping between classes and elements or attributes

The mapping process enables us to convert fragments of XML data into objects. Imagine the following piece of XML code in 3.1.

Listing 3.1: Basic XML structure

```
<student>
    <firstname>John</firstname>
    <lastname>Doe</lastname>
    <age>20</age>
</student>
```

We should be able to convert this into an object with minimal macros showed in 3.2.

Listing 3.2: Basic mapping macros

```
class Student
  include Oxm
  string "firstname"
  string "lastname"
  integer "age"
end
```

More advanced mapping features for advanced XML structures in 3.3 that enable us to convert complexity into understandable objects in 3.4 should also be possible.

Listing 3.3: Advanced XML structure

```
<person>
    <name frstnm="John" lstnm="Doe" />
    <details>
        <yearsold>20</yearsold>
    </details>
</person>
```

Listing 3.4: Advanced mapping macros

```
class Student
  include Oxm
  tag "person"
  string "firstname", :element => "name", :attribute => "frstnm"
  string "lastname", :element => "name", :attribute => "lstnm"
  integer "age", :element => "details/yearsold"
end
```

Defining this mapping once, gives us the ability to access object attributes easier, without constantly being concerned about the complex XML structure as showed in 3.5.

Listing 3.5: Mapped object

```
student.firstname    # returns "John"
student.lastname     # returns "Doe"
student.age          # returns 20
```

In listing 3.5 the object instance variable `firstname` is now the shorthand for accessing the value of the `frstnm` attribute in element `name`.

3.3.2 Associations between objects by simple meta-programming macros

Associations are macro-like class methods for tying objects together, based upon child-and and parent nodes. They express relationships like "Student has many courses" or "Student belongs to University". In many cases XML fragments consist of multiple objects that relate to each other, within one result-set.

Listing 3.6: XML with associations

```
<student>
    <firstname>John</firstname>
    <lastname>Doe</lastname>
    <courses>
        <course>
            <subject>XML Databases</subject>
            <score>8</score>
        </course>
        <course>
            <subject>Relational Algebra</subject>
            <score>7</score>
        </course>
    </courses>
</student>
```

We like to map specific object relations with meta-programming macros such as `belongs_to`, `has_many` or `has_one`.

Listing 3.7: Association mapping

```
class Course
  include Oxm
  string "subject"
  integer "score"
  belongs_to :student
end

class Student
  include Oxm
  string "firstname"
  string "lastname"
  has_many :courses
end
```

In 3.7 example the `has_many :courses` macro (on line 12) knows that all child elements, from the `courses` element are associated as `courses` for an object instance. Such that we can retrieve the `courses` from a given `student` with a simple statement.

```
1 student.courses   # returns a hash with courses from the student
```

3.3.3 Validation rules

Validation rules in 3.8 enable us to define business logic for objects easily.

Listing 3.8: Validation rules

```
1 class Student
2   include Oxm
3   validates_presence_of :firstname, :lastname
4   validates_uniqueness_of :email
5 end
```

The free available Ruby gem `validatable` [18] handles all these validation functions instantly.

3.3.4 Callbacks

Callbacks in 3.9 are hooks in the life-cycle of an object that can trigger logic before or after an alteration of the object state.

Listing 3.9: Callbacks

```
1  class Student
2    include Oxm
3
4    after_save :send_verification_email
5
6    def send_verification_email
7      Mailer.send_verification(self.email)
8    end
9
10 end
```

The module `ActiveSupport` [2] within Ruby on Rails supports a complete series of callback functions. Mixing this module within the OXM allows us to use callbacks.

3.3.5 Finders

Finder operations return object with specific result-sets. These operations make it easy for developers to retrieve data from the data source, without the hassle of defining complex XPath expressions. Example 3.10 below show how finder operations could work:

Listing 3.10: Finder operations

```
1 Student.find(:all)
2 Student.find(:all, :conditions => "[age>35]")
3 Student.find_all_by_lastname("Doe")
```

Each finder operation is converted into an XPath expression that queries the XML database using the well known Pathfinder [24] mechanism.

3.3.6 Direct manipulation

We like to support direct data manipulation to `create`, `update` or `destroy` objects instances. Developers can work with objects without knowing the internal XML structure for an object.

Listing 3.11: Direct data manipulation

```
s = Student.find_all_by_lastname("Doe").first
s.firstname = "Alice"
s.save
```

In 3.11 search for the first user with `lastname` "Doe" and change its name to "Alice". After that we `save` the user. We do not use any XML specific syntax here, but use only object syntax supported by the Object XML Mapping.

3.3.7 Database abstraction

We like the ability to adapt to many different databases in order to integrate with an existing setup. This is done in one single configuration file that allows us to configure the database settings in 3.12.

Listing 3.12: Database abstraction

```
:adapter  => "postgresql",
:host     => "localhost",
:username => "john",
:password => "secret",
:database => "oxm"
```

Sequel[15] implements this database file, and allows us to use any of its independent database adapters such as ADO, DataObjects, DB2, DBI, Firebird, Informix, JDBC, MySQL, ODBC, OpenBase, Oracle, PostgreSQL and SQLite3.

Chapter 4

Design

In this chapter we propose a design for object XML mappings. The design contains two key components; an *object map* and *mapped nodes table*. We explain the design in three steps:

1. A mapping mechanism that maps macros to a class level *object map*.
2. An object instance storage mechanism that merges the *object map* with XML data to an instance level *mapped nodes table*.
3. A dynamic programming method that can access the *mapped nodes table*.

Both the *object map* and *mapped nodes table* exist in the domain layer of an enterprise system 4.1.

Component	**Layer**	**Description**
Object map	Domain	Stores object mapping macros on class level
Mapped nodes table	Domain	Temporary storage for XML nodes on instance level
Relational database	Data Source	Persistent storage for XML nodes

Table 4.1: *Object map* and *mapped nodes table* in the domain layer.

4.1 Object map

Before we continue, it is crucial to comment on the differences between *class level* and *instance level* variables in Ruby.

- *Class level variables* are shared among all objects of a class, and are also accessible to the class methods.
- *Instance level variables* are only associated with a particular instance of the class.

We introduce a class level *object map* that defines the mappings from XML nodes to object instance variables. This is done using simple meta programming macros as shown in the code example below.

Listing 4.1: Object meta programming macros

```
1 class Student
2   include Oxm
3   string "firstname", :element => "name", :attribute => "first"
4   string "lastname", :element => "name", :attribute => "last"
5   integer "age"
6 end
```

The class level *object map* is shared among all objects of a class and consists of the following fields:

- *name*: The accessible variable name for the object instance.
  For example "firstname" in line 3 of 4.1.

- *type*: The type of the variable ∈ { `String`, `Integer`, `Boolean`, ... }.
  For example "string" in line 3 of 4.1.

- *element*: An expression that points to the element.
  For example "name" in line 3 of 4.1.

- *attribute*: An optional attribute within the *element*.
  For example "first" in line 3 of 4.1.

We have created simple class level functions that are executed prior to object instantiation. These functions add *Mapping* entries to the *object map*. The code fragment in 4.2 explains this.

Listing 4.2: Object meta programming macros source code

```
1  module ClassMethods
2
3    def string(name, options={})
4      add_mapping name, "String", options
5    end
6
7    def integer(name, options={})
8      add_mapping name, "Integer", options
9    end
10
11   ...
12
13   private
14
15       def add_mapping(name, type, options={})
16         item = Mapping.new(name, type, options)
17         @object_map[item.element] = item
18       end
19   end
20 end
```

In code fragment 4.2 the function `def string` (line 3) and `def integer` (line 7) are defined as wrappers around `def add_mapping` (line 15). Both wrapper functions execute the command to add a new row to the *object map* with their own specific type (line 16 and 17).

Table 4.2 presents what the *object map* object map looks like for the Student class in Listing 4.1.

name	type	element	attribute
firstname	string	name	first
lastname	string	name	last
age	integer	age	

Table 4.2: Object map for Student class

4.2 Mapped nodes table

The *mapped nodes table* merges the *object map* with XML data, and is associated with the instance of an object. The *mapped nodes table* holds in-object XML nodes data during the lifetime of an object instance. It temporarily stores object instance data and modifications on that data.

4.2.1 Extending the nodes table

We introduce the *mapped nodes table 4.3*, an in-memory relational-database table, with the Pathfinder[23] tree encoding pattern, that stores XML nodes that are related to the object instance. The *mapped nodes table* also contains mapping information, and therefore we extend the well known tree-encoding[23] table with two columns:

- *mapping:* The accessible variable name for the object instance.
- *type:* The type of the variable ∈ { `String`, `Integer`, `Boolean`, ... }.

Student::nodes							
pre	size	level	kind	value	name	**mapping**	**type**
0	5	0	elem	student			
1	2	1	elem	name			
2	0	2	attr	Doe	last	**lastname**	**string**
3	0	2	attr	John	first	**firstname**	**string**
4	1	1	elem	age			
5	0	2	text	20		**age**	**integer**

Table 4.3: Mapped nodes table

We extended the Pathfinder node shredder with minimal statements that lookup the *object map* for mappings during the shredding procedure. The `mapping` and `type` properties are added to the desired nodes in the *mapped nodes table (in 4.3)* whenever a match is made.

4.2.2 Accessing variables using method_missing invocation

Accessing the *mapped nodes table* is done by `method_missing` invocation. This is a typical technique used in dynamic programming. Whenever a Ruby object receives a message for a method that is not implemented in the receiver, it invokes the `method_missing` method. This is used to catch both `get` and `set` operations.

Each `method` that comes in on the `method_missing` listed in 4.3 is matched against a regular expression to determine whether it is a `get` or a `set` operation:

- In case of a `get` operation (E.g. `s.firstname`) the *mapped nodes table* is queried for the matching pattern *(line 3)*. The queried `value` from the *mapped nodes table (line 4)* is typecasted *(line 7 to 10)* and returned.
- In case of a `set` *(line 13)* operation (E.g. `s.firstname = "Andrew"`) the *mapped nodes table* is updated with the given argument *(line 15)*.

Listing 4.3: Method missing implementation

```
1  def method_missing(method, *args)
2    case method.to_s
3    when /^\w+$/       # Filter GET request using regular expression
4      dataset = @mem_db["SELECT value, type FROM nodes WHERE
5          mapping = ?", method.to_s]
6      case dataset.map(:type).to_s
7      when "String"
8        typecast(dataset.map(:value).to_s, String)
9      when "Float"
10       typecast(dataset.map(:value).to_s, Float)
11
12     ...
13   when /^\w+=$/      # Filter SET request using regular expression
14     dataset = @mem_db[:nodes]
```

```
15      dataset.filter('mapping = ?', method.to_s[0..-2]).update(
16          :value => args.to_s, :changed => true)
17    end
18  end
```

Chapter 5

Evaluation

The previous chapter showed us how to map XML data to objects with minimal effort. In this chapter we will evaluate the proposed work on a variety of aspects:

- Developer productivity
- Performance

5.1 Developer productivity

Ruby on Rails, as a web-development framework, became very popular due to its productiveness for developers. The framework enables engineers to develop software quickly. Productivity wins are always relative and situation specific. Neither Java, C++ or Ruby on Rails is the right solution to every problem. Understanding the problem correctly, making assumptions and capabilities of team members, are also important factors. All these factors make it hard to qualify the "developer productivy" or even draw conclusions. We based the concept of the OXM on the principles that made Ruby on Rails successful.

The Ruby on Rails framework is not a typical application-development framework. Ruby on Rails founder David Heinemeier Hansson often calls the framework opinionated software. Long-standing framework conventions have been broken by strong philosophical decisions, that are strictly followed throughout the framework. We have followed these conventions also:

- **Seamless integration:** Rails is written in the Ruby language. It extends Ruby such that it is often hard to identify where Ruby ends and Rails begins. We have accomplished an integration between the OXM and the model-view-controller (MVC) framework in Ruby on Rails. We can define a mapping, map XML to objects, and use that object in a view, within 5 lines of code. The syntax for the OXM is in Ruby style defined, such that is understandable for programmers.

- **Convention over configuration:** Most frameworks maintain large configuration files. Rails rejects this strategy by making assumptions with a common project directory structure and simple, common naming conventions for methods and classes. Therefore a fraction of the configuration is needed. We have created standard conventions with simple mapping macros for elements, attributes and associations that have their initial settings built in.

- **Do not repeat yourself:** Do not Repeat Yourself, or DRY, is a common buzzword within the Rails community. The trick here is to seek for repetitive tasks that can be abstracted into methods embedded into the framework. We have applied the basic principles from Active Record in to the OXM: It is only necessary to define the object mapping and business rules at one place, in order to get the same object behavior all over the application.

With complying to these basic conventions, we enable developers to adapt XML data in the programming stack, with minimal effort.

5.1.1 Case study

We have done a case study to evaluate the OXM, based on a Twitter [16] application. All users on Twitter have a timeline with status messages that display what they are doing. An example of a status is: *"Creating case study application for his thesis, using a twitter example! by jdiepenmaat on 1:45 AM Jun 4th from web"*.

Twitter comes with the Twitter API that can be used to retrieve status updates (in XML) from a given user. We developed an application that maps these status updates onto Ruby objects in appendix A.1 and prints the status. The example in appendix A.2 shows the minimal developer effort is needed to map Twitter's XML feed onto objects, using a few mapping macros supported by the OXM. This solution enables developers to adapt XML objects within the application with minimal programming effort. The focus is purely on: *"What nodes should be mapped, and what are the associations between objects?"* The success of RoR has proven that, with this elegancy in code design, programmers achieve great results.

Comparing to other solutions

As stated earlier, neither Java or Ruby is the right solution to every problem. Each programming language declares its own functions to handle specific situations. Almost any object oriented languages allows programmers get XML data into objects. This is often done in the following three steps:

1. Initialize a DOM XML parser.
2. Parse the XML feed into the DOM tree.
3. Copy the data from the DOM tree into the object.

In fact there is nothing wrong with this approach. A well experienced programmer can write code for this within a reasonable amount of time. But it is important to realize that developers have to do most of these steps over and over again in the application. Opposite to the OXM that abstracts the essentials, such that developers only have to think what really matters. Nevertheless, the OXM still limits usage for several situation as listed in the next chapter.

5.2 Performance

Basic performance benchmarking was done on several important aspects of the OXM. We used a MacBook 2.4Ghz with 2GB memory, running Mac OS X Leopard and Ruby 1.8.6 to run the benchmark tests.

5.2.1 SAX Parser, tree-encoding and mapping load

The OXM uses three primary components when it maps an XML document onto an object:

1. The SAX parser scans through the document and fires events.
2. The relational tree encoding algorithm adds `pre`, `size` and `level` values to the nodes identified by the SAX parser.
3. The mapping mechanism adds mapped nodes to the *mapped nodes table*.

We benchmarked the individual loads of these components. This was done using a 16000 nodes counting XML document with student data. A `Student` class with 10 object mappings was used.

We can calculate from table 5.1 that the SAX parser uses 42.7% of the resources for reading the nodes. The Tree encoding uses 54% of the resources for numbering the nodes. And the mapping mechanism takes the minimal 3.3% of the total amount of resources available.

used components	time (sec.)	nodes per second
SAX parser	0.0433	369515
SAX parser + Encoding	0.0982	162932
SAX parser + Encoding + Mapping	0.1015	157635

Table 5.1: Benchmark on OXM components

number of nodes	time (sec.)	nodes per second
500	0.011	45454
1000	0.012	83333
2000	0.0181	110497
4000	0.0311	128617
8000	0.0569	140597
16000	0.1096	145985

Table 5.2: Benchmark on mapping objects

5.2.2 Mapping node objects

Other sources [11] indicate that XML Mappers become slower when the document size increases (likely due a DOM parser). Therefore we have benchmarked a set of XML documents with sizes from 500 up to 16000 nodes. Our XML documents contains student information with 10 object mappings for the `Student` class.

The benchmark results in table 5.2 show us that the mapping speed is almost linear for documents with up to 16000 nodes.

5.2.3 Read and write operations

Object instances will be accessed once the XML data is mapped. Therefore we have benchmarked a number of object instances with `500`, `4000` and `16000` mapped nodes. Again, we chose XML documents for a `Student` class with 10 object mappings.

number of nodes	reads per second	writes per second
500	1364	3875
4000	1360	3868
16000	1362	3870

Table 5.3: Benchmark on read/write operations

The benchmark results in table 5.3 show us that the OXM can handle a substantial amount of read and write requests without losing speed.

Chapter 6

Conclusions

6.1 Summary

In this thesis, we introduced an object XML mapping mechanism that maps XML data onto objects with minimal programmer effort. Software developers can use this mechanism to work with XML files more agile. The mechanism enables us to define (complicated) mappings from (difficult) XML documents into common Ruby object instances.

6.2 Research questions

We answer the research question "How can we implement XML based Object Mapping?", as defined in paragraph 1.6.

1. *How can we implement Pathfinder's[23] relational tree encoding algorithm in a Ruby on Rails plugin, such that it supports Postgresql?*

   In chapter 4 we proposed the *mapped nodes table* (paragraph 4.2) based on XPath Accelerator's shredding algorithm. We have implemented the encoding based on the SAX parser model using LibXML's Ruby wrapper.

2. *How can we implement an Object XML Mapping on Pathfinders relational tree encoding?*

   In chapter 4 we introduced an object XML mapping mechanism. Each object based on the object XML mapping has its *object map* (paragraph 4.1) that defines the mappings from XML nodes onto object instance variables. A *mapped nodes table* (paragraph 4.2), based on the Pathfinder relational tree-encoding algorithm, was introduced for in-object node storage. Dynamic programming techniques are used to access and modify data from the *mapped nodes table* in the object. The mechanism is also capable of processing associations and offers facilities for direct data manipulation.

6.3 Future work and research

We have listed possible future work for the OXM. Some of the items listed can be implemented based on knowledge from research in the past. Others may need more in depth fundamental research.

- *Embedding the XPath expression engine*

  The Pathfinder relation tree-encoding [24] comes with a set of steps that convert XPath expressions into relational SQL expressions. It would be appreciated to support those XPath expressions within the OXM such that persistent data can be retrieved from the SQL database underneath. Special attention goes to the question whether we can exclude unmapped nodes, based on the *object map* in paragraph 4.1, during query execution.

- *Loading objects from SQL node result-sets*

  Currently we use the SAX parser to parse XML fragments into the in the *mapped nodes table* (paragraph 4.2). The SAX parser looks up the *object map* during shredding procedure to determine and add mappings to the *mapped nodes table*. The persistent storage in the relational database and the *mapped nodes table* use both the same relational tree encoding. Hence, it would be nice to load objects straight from SQL nodes table into OXM objects without using the SAX parser technique for XML fragments. This could possibly save an arguable amount of time. An alternative, to the SAX parser strategy, to map the data from the *object map* seems to be necessary.

- *Direct data manipulation*

  The OXM can use a relational database, such as Postgresql or MySQL, for persistent XML storage using the Pathfinder relational tree-encoding. Whenever an OXM object is instantiated, nodes from the persistent XML storage are loaded into the object instance's *mapped nodes table* 6.3. The idea is to extend both the persistent nodes table and the *mapped nodes table* with a `node id` column. This allows us to manipulate persistent XML data using SQL expressions, straight from OXM object instance.

- *Connecting to an external XML database*

  Several XML databases exist, such as MonetDB/XQuery[9] and IBM DB2 Pure XML[8]. It is interesting how we can connect the OXM with an XML database and what the consequences are.

- *Concurrency*

  Concurrency is a tricky aspect of software development. We run into concurrency problems whenever multiple processes manipulate the same data. There is currently not a comprehensive overview of all possible concurrency problems. Based on an overview, we can work on specific implementation to solve concurrency problems.

- *Recursive XML*

  A DTD can describe a recursive pattern for an XML document. Difficulties arise when trying to query recursive data [22]. Mapping recursive data into an object model is more than likely hard, and at this stage not supported by the OXM.

- *References and namespaces*

  At this stage references and namespaces in elements are not supported in the mechanism presented. Each of these need more in depth research. Macros and appropriate mapping techniques need to be defined that support using them in the OXM.

- *Automatic serialization*

  In this research, we proposed a mechanism for mapping XML data onto objects. An XML document comes often with an XML schema. Simple serializing procedures to convert objects into XML files exist in Ruby, but remain very basic. It would be challenging to come up with an approach to serialize objects, given the *object map* and XML Schema, into XML automatically (or with minimal programming effort).

Appendix A

Case study

A.1 Mapping a Twitter Feed

Listing A.1 shows the an example XML (truncated) of a Twitteruser's timeline. The timeline contains a `status` and `user` object. The `status` represents what a `user` is currently doing. The `user` part contains specific information about the user.

We mapped the `status` and `user` object in XML feed A.1 with the code in A.2. This mapping enables us to simply parse an object with XML data in A.3 .

Listing A.1: Twitter feed from a user's timeline (truncated)

```
<status>
  <created_at>Thu Jun 04 08:45:58 +0000 2009</created_at>
  <id>2027426795</id>
  <text>Creating case study application for his thesis, using a twitter example
      !</text>
  <source>web</source>
  <truncated>false</truncated>
  <in_reply_to_status_id></in_reply_to_status_id>
  <in_reply_to_user_id></in_reply_to_user_id>
  <favorited>false</favorited>
  <in_reply_to_screen_name></in_reply_to_screen_name>
  <user>
    <id>5457352</id>
    <name>Joost Diepenmaat</name>
    <screen_name>jdiepenmaat</screen_name>
    <location>Enschede</location>
    <description>happy entrepreneur, loves @annekeelferink, plays french horn,
        drinks lots of tea.</description>
    <profile_image_url>http://s3.amazonaws.com/twitter_production/profile_images
        /73611457/DSC_0065_normal.jpg</profile_image_url>
    <url>http://www.moneybird.nl</url>
    <protected>false</protected>
    <followers_count>133</followers_count>
    <profile_background_color>0067CD</profile_background_color>
    <profile_text_color>000000</profile_text_color>
    <profile_link_color>0000ff</profile_link_color>
    <profile_sidebar_fill_color>FFFFFF</profile_sidebar_fill_color>
    <profile_sidebar_border_color>BCBCBC</profile_sidebar_border_color>
    <friends_count>154</friends_count>
    <created_at>Tue Apr 24 07:46:12 +0000 2007</created_at>
    <favourites_count>4</favourites_count>
    <utc_offset>3600</utc_offset>
    <time_zone>Amsterdam</time_zone>
    <profile_background_image_url>http://static.twitter.com/images/themes/theme1/
        bg.gif</profile_background_image_url>
```

```
    <profile_background_tile>false</profile_background_tile>
    <statuses_count>1020</statuses_count>
    <notifications></notifications>
    <following></following>
  </user>
</status>
```

Listing A.2: Mapping macros for the `Status` and `User` object

```
# user.rb
class User
  include Oxm
  string "name"
  string "location"
end

# status.rb
class Status
  include Oxm
  date "created_at"
  string "text"
  has_one "user", User
end
```

Listing A.3: Working with the mapping

```
status = Status.parse(\"status.xml\")
puts "User #{status.user.name} says \"#{status.text}\""
# User Joost Diepenmaat says "Creating case study application for his thesis,
    using a twitter example!"
```

Bibliography

[1] Active record, object-relation mapping put on rails. http://ar.rubyonrails.org/.

[2] Activesupport. http://caboo.se/doc/classes/ActiveSupport/Callbacks.html.

[3] Apache openjpa. http://openjpa.apache.org/.

[4] Electronic business xml. http://www.ebxml.org/.

[5] Grails object relational mapping (gorm). http://www.grails.org/GORM.

[6] Happymapper, xml to ruby object mapper. http://github.com/jnunemaker/happymapper/tree/master.

[7] Hibernate: relational persistence for java. http://www.hibernate.org.

[8] Ibm db2 purexml. http://www-01.ibm.com/software/data/db2/xml/.

[9] Monetdb/xquery. http://www.monetdb.nl/projects/monetdb/XQuery/index.html.

[10] Nhibernate: Relational persistence for .net. http://www.nhibernate.org.

[11] Oracle toplink. http://www.oracle.com/technology/products/ias/toplink/index.html.

[12] Roxml, module for binding ruby classes to xml. http://roxml.rubyforge.org/.

[13] Ruby on rails open-source web framework. http://rubyonrails.org/.

[14] Ruby programming language. http://www.ruby-lang.org/.

[15] Sequel: The database toolkit for ruby. http://sequel.rubyforge.org/.

[16] Twitter. http://www.twitter.com/.

[17] Universal business language v2.0. http://docs.oasis-open.org/ubl/os-UBL-2.0/UBL-2.0.html.

[18] Validatable gem. http://validatable.rubyforge.org/.

[19] Xml path language (xpath). http://www.w3.org/TR/xpath.

[20] Xml query language (xquery). http://www.w3.org/TR/xquery/.

[21] Xml schema. http://www.w3.org/XML/Schema.

[22] Loredana Afanasiev, Torsten Grust, Maarten Marx, Jan Rittinger, and Jens Teubner. Recursion in xquery: put your distributivity safety belt on. In *EDBT '09: Proceedings of the 12th International Conference on Extending Database Technology*, pages 345–356, New York, NY, USA, 2009. ACM.

[23] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. Pathfinder: Xquery—the relational way. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 1322–1325. VLDB Endowment, 2005.

[24] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. Monetdb/xquery: a fast xquery processor powered by a relational engine. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 479–490, New York, NY, USA, 2006. ACM.

[25] Brown et al. *Enterprise Java Programming with IBM Websphere.* Addison-Wesley, ISBN-10: 032118579X, 2001.

[26] Martin Fowler. *Patterns of Enterpise Application Architecture.* Addison-Wesley, ISBN-10: 0321127420, 2003.

[27] Torsten Grust, Jan Rittinger, and Jens Teubner. Why off-the-shelf rdbmss are better at xpath than you might expect. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 949–958, New York, NY, USA, 2007. ACM.

[28] Torsten Grust, Maurice van Keulen, and Jens Teubner. Staircase join: teach a relational dbms to watch its (axis) steps. In *VLDB '2003: Proceedings of the 29th international conference on Very large data bases*, pages 524–535. VLDB Endowment, 2003.

[29] Trygve Reenskaug. Models views controllers. `http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf`.

[30] Chris Richardson. Orm in dynamic languages. *Commun. ACM*, 52(4):48–55, 2009.