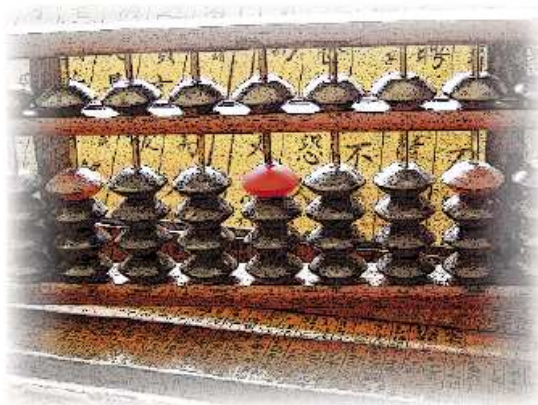


The  
Pragmatic  
Programmers

# Enterprise Integration with Ruby



*Maik Schmidt*

The Facets



of Ruby Series



The book you're reading is still under development. As an experiment, we're releasing this copy well before we normally would. That way you'll be able to get this content a couple of months before it's available in finished form, and we'll get feedback to make the book even better. The idea is that everyone wins!

Be warned. The book has not had a full technical edit, so it will contain errors. It has not been copyedited, so it will be full of typos. And there's been no effort spent doing layout, so you'll find bad page breaks, over-long lines, incorrect hyphenations, and all the other ugly things that you wouldn't expect to see in a finished book. We can't be held liable if you use this book to try to create a spiffy application and you somehow end up with a strangely shaped farm implement instead. Despite all this, we think you'll enjoy it!

Throughout this process you'll be able to download updated PDFs from [http://books.pragprog.com/titles/fr\\_eir/reorder](http://books.pragprog.com/titles/fr_eir/reorder). When the book is finally ready, you'll get the final version (and subsequent updates) from the same address. In the meantime, we'd appreciate you sending us your feedback on this book at [http://books.pragprog.com/titles/fr\\_eir/errata](http://books.pragprog.com/titles/fr_eir/errata).

Thank you for taking part in this experiment.

► **Dave Thomas**

# Enterprise Integration with Ruby

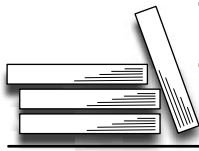
---

A Pragmatic Guide

Maik Schmidt

**The Pragmatic Bookshelf**

Raleigh, North Carolina   Dallas, Texas



# Pragmatic Bookshelf

---

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at

<http://www.pragmaticprogrammer.com>

Copyright © 2006 The Pragmatic Programmers LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.

ISBN 0-9766940-6-9

Printed on acid-free paper with 85% recycled, 30% post-consumer content.

B1.2 printing, January 2006

Version: 2006-1-24

# Contents

---

|          |                                                        |            |
|----------|--------------------------------------------------------|------------|
| <b>1</b> | <b>Introduction</b>                                    | <b>1</b>   |
| 1.1      | What Is Enterprise Software? . . . . .                 | 2          |
| 1.2      | What Is Enterprise Integration? . . . . .              | 3          |
| 1.3      | Why Ruby? . . . . .                                    | 3          |
| 1.4      | Who Should Read This Book? . . . . .                   | 5          |
| 1.5      | PragBouquet . . . . .                                  | 5          |
| 1.6      | Acknowledgments . . . . .                              | 6          |
| <b>2</b> | <b>Databases</b>                                       | <b>8</b>   |
| 2.1      | The Coupon Application . . . . .                       | 9          |
| 2.2      | Database Interface (DBI) . . . . .                     | 25         |
| 2.3      | Object-Relational Mappers . . . . .                    | 28         |
| 2.4      | Lightweight Directory Access Protocol (LDAP) . . . . . | 51         |
| <b>3</b> | <b>Processing XML</b>                                  | <b>75</b>  |
| 3.1      | A Short XML reminder . . . . .                         | 77         |
| 3.2      | Generating XML documents . . . . .                     | 79         |
| 3.3      | Processing XML Documents . . . . .                     | 91         |
| 3.4      | Validating XML Documents . . . . .                     | 123        |
| 3.5      | Are There Alternatives to XML? . . . . .               | 128        |
| <b>4</b> | <b>Low Ceremony Distributed Applications</b>           | <b>141</b> |
| 4.1      | “I’d Rather Use a Socket” . . . . .                    | 142        |
| 4.2      | Remote Procedure Calls Using HTTP . . . . .            | 155        |
| <b>5</b> | <b>Distributed Applications with RPC</b>               | <b>175</b> |
| 5.1      | Another Day, Another Protocol . . . . .                | 175        |
| 5.2      | We Will Take No REST, Will We? . . . . .               | 185        |
| 5.3      | SOAP . . . . .                                         | 196        |
| 5.4      | CORBA, RMI, and Friends . . . . .                      | 210        |

|          |                                                 |            |
|----------|-------------------------------------------------|------------|
| <b>6</b> | <b>Tools and Techniques</b>                     | <b>230</b> |
| 6.1      | Internationalization and Localization . . . . . | 230        |
| 6.2      | Logging . . . . .                               | 250        |
| 6.3      | Creating Daemons and Services . . . . .         | 269        |
| 6.4      | Build and Deployment Process . . . . .          | 276        |
| 6.5      | Project Automation with Rake . . . . .          | 293        |
| 6.6      | Testing Legacy Applications . . . . .           | 304        |

*There are two types of complex systems: those that have grown out of simpler systems and those that do not work.*

► Unknown

## Chapter 1

# Introduction

---

Have you ever worked for a big enterprise? Do you remember your expectations as you walked into work on that first day? Whistling as the sun shone brightly, you might have been thinking “It will be great to work for <company name here>. They will have a professional environment, where coffee is free, where every system has been specified accurately, implemented carefully, and tested thoroughly. Hmmm... I wonder which database and programming language they use.”

After your fifth cup of free coffee (around 9:07) you came to realize that the real world looks completely different from your expectations. Typical enterprises use dozens, hundreds, and sometimes even thousands of applications, components, services, and databases. Many of them were custom-built in-house or by third parties, some were bought, others are based on Open Source projects, and the origin of a few—usually the most critical ones—is completely unknown. A lot of applications are very old, some are fairly new and seemingly no two of them were written using the same tools. They run on heterogeneous operating systems and hardware, they use databases and messaging systems from various vendors, they were written in different programming languages.

The reasons for this are manifold. You can find countless books explaining why the situation is so bad. You can even find books claiming that they help you to prevent such a chaos. This book uses another approach. We will not help you to clean up this mess, but we will help you to deal with the problems pragmatically. Instead of complaining that valuable data is spread across different database schemas or across databases from several vendors, we will write code that integrates it. We will take it even a step further and write new applications which aggregate *all* your existing resources. It doesn't matter if we

have to use relational databases, LDAP repositories, XML files, or web services based on different protocol standards. We will blend data from multiple, disparate databases to create new business knowledge.

Along the way we'll show you how to solve all the small day-to-day problems. These are the things that occur over and over again, especially when developing enterprise software. We will access relational databases such as Oracle and MySQL and we will work with LDAP repositories. We'll show you how to do application logging, how to deploy your software, how to automate tedious and error-prone tasks, and how to survive in an international environment. Oh, and as you might have guessed already from the book's title, we will use Ruby to accomplish all these things.

## 1.1 What Is Enterprise Software?

In *Patterns of Enterprise Application Architecture* [?], Martin Fowler writes: "Enterprise applications are about the display, manipulation, and storage of large amounts of often complex data and the support or automation of business processes with that data."

That's a concise but nevertheless abstract definition, because every non-trivial piece of software has to store, manipulate, and display data. Video games do nothing else (and modern video games also need huge amounts of data that often can get complex). The key point in the definition above is the second part: that the data in enterprise applications is used for business processes and not for rendering alien space ships.

Unsurprisingly, there are more differences between enterprise applications and other types of software. For example, enterprise applications are often created only for a small user group that is in close contact with the development team, implying the developers know their customers very well. In extreme cases programs are written for only a single person (special report generators for the CEO, for example).

Enterprise software demands a certain set of tools. Large amounts of data—complex or not—have to be stored somehow and somewhere. Often it is stored in relational databases, but it can also be in plain text files or LDAP repositories. In addition, modern enterprise software is often based on distributed architectures consisting of many small to mid-size components that perform specialized tasks and that are connected by some kind of middleware such as CORBA, RMI, SOAP, and XML-RPC.



Obviously, as an enterprise software developer you're better off if you know how to deal with such technologies. You shouldn't be troubled by the details of reading from a relational database or accessing a LDAP repository. Mastering skills such as these help you to concentrate on the fun stuff—the application itself.

## 1.2 What Is Enterprise Integration?

Enterprise integration is a rather vague term and cannot be defined in a strict mathematical sense. Simply put, it happens whenever you use an existing enterprise resource to achieve some results. If you use an existing database or web service in your application, you're performing enterprise integration. If you build a new component that is used by other pieces of your existing architecture, you're doing enterprise integration, too.

Integration needn't just happen inside a single enterprise. It's possible—and not too unusual—that the software or data of two different enterprises has to be integrated. If you're using a payment gateway to bill your customers, for example, you're effectively integrating enterprise software.

You might ask yourself if every development activity in an enterprise environment is some kind of enterprise integration. There are a few exceptions. Enterprise integration does not happen when you build a completely new piece of software from scratch, for example. In reality this case is rare, but from a theoretical point of view this is the only clear exception.

Enterprise integration often means integration with standard software such as databases, LDAP repositories, message queues, ERM systems, and so on. If you're using one of these technologies, chances are good that you're doing some enterprise integration.

## 1.3 Why Ruby?

Most enterprise software running today was written in languages such as COBOL, C/C++, and Java. Because of its distributed nature, enterprise software often makes it easy to use new tools and programming languages. When you have to create a small standalone application—one that only relies upon an existing database, SOAP service, or LDAP repository—it almost doesn't seem to matter if you were to write it in

C++, Java, or Ruby. But if you look into it more deeply, dynamic languages such as Perl, Python, and Ruby have many of advantages, especially in enterprise environments:

- They are interpreted and do not need a compile phase, which increases development speed tremendously. After editing your program you can see the results of your changes immediately.
- Enterprise software is about munging data. Dynamic languages are designed to handle data, and include high-level data types such as hashes.
- Memory management is dealt with by the language. This is a great advantage over languages such as C++ where you have to specify the length of each string you read from a database. Dynamic languages prevent waste and result in more concise, more robust, and more secure software.
- Software written in dynamic languages is installed as source code, so you always know exactly which version is currently running on your production system. Gone are the days when you had to guess if a certain binary executable is the right one.

We will show you Ruby's strengths and how it helps you to accomplish many tasks much faster, more elegant, and with more fun than with any other programming language available today. But, even more important, we will also tell you about Ruby's weaknesses. Ruby is comparatively young and although the core of the language is mature and lots of excellent libraries are available, many things are still missing or incomplete.

Although there is no industry standard for enterprise programming with Ruby (as there is with J2EE or .NET), everything you need is readily available. The most important libraries come with every Ruby distribution and the standard distribution has grown rapidly over the last years. All the other stuff can be found in public places such as RubyForge<sup>1</sup> or the Ruby Application Archive<sup>2</sup>.

---

<sup>1</sup><http://www.rubyforge.org>

<sup>2</sup><http://raa.ruby-lang.org>

## 1.4 Who Should Read This Book?

This book was written for experienced enterprise developers who know Java, C#, or C++, but don't know much Ruby (although you should probably have read *Programming Ruby* [?]). We assume you are familiar with relational databases and have at least an idea what LDAP is. Maybe you do not know RELAX NG, but you understand the concepts of XML and you know what *well-formed*, *SAX2*, and *DOM* mean.

You've probably used tools such as object-relational mappers. Maybe you're familiar with Enterprise Java Beans (EJB), Java Data Objects (JDO) and so on. Maybe you're fed up with editing configuration files instead of coding. You are looking for better ways to integrate the existing resources in your company and you are looking for better ways to quickly create new and fancy applications based on all the wonderful stuff you already have.

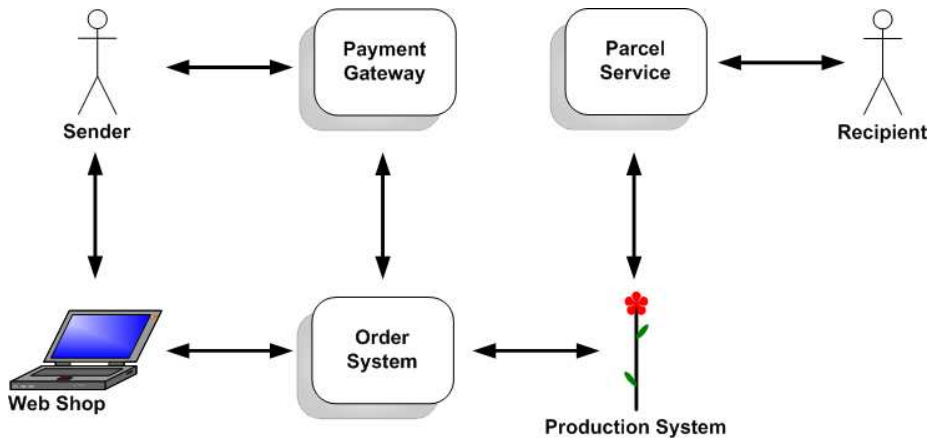
Depending on the tools you've used to build your architecture, different choices are available for the integration process. If you're using message queues you have a lot of freedom and flexibility for integrating your services and software with others. The same holds true for all kinds of web service protocols. It's slightly different with databases, because they usually do not offer interfaces as clean as message based systems do. Sometimes you have to access tables directly, sometimes you have to use a set of stored procedures written in a proprietary database programming language.

In this book we do not talk about sophisticated messaging patterns. Instead, we cover the basics. We show you how to use databases, web services, XML files, and all the other legacy stuff you want to combine for building new applications.

## 1.5 PragBouquet

To make things more interesting and tangible we've founded an imaginary company called *PragBouquet*. It sells flowers from a web shop. Customers from all over the world can order flowers and send them to people living in the United States.

PragBouquet's business demands a lot of components and services. It depends on several partners, too. Their current infrastructure is shown in Figure 1.1, on the following page. Customers place orders in the web shop. The shop communicates with the central order system. Because




---

Figure 1.1: PragBouquet Infrastructure

---

PragBouquet has no billing system, the order system uses an external payment gateway to charge orders. In parallel the production system is informed of new orders and busy florists create wonderful bunches of flowers. Eventually, the floral goods are picked up by a parcel service and are delivered to the happy recipient.

This is only a rough overview. We'll show single components in more detail when necessary.

## 1.6 Acknowledgments

First, I'd like to thank Dave Thomas and Andy Hunt for giving me the opportunity to write this book for *The Pragmatic Bookshelf*. Working with them has been both an honor and a pleasure. I couldn't imagine better or more professional working conditions.

It would be impossible to write a book about software for enterprise integration without the software itself. The following gentlemen kindly made their ingenious work public for free, and have always responded quickly and accurately to all my questions: Yukihiro “matz” Matsumoto, Will Drewry, arton (the author of Rjb), Sean Russel, Ian Macdonald, Takaaki Tateishi, Thomas Uehlinger, Jim Weirich, Nikolai Lugovoi, Daniel Berger, *why the lucky stiff*, Minero Aoki, Michael Neumann, Kubo Takehiro, Tomita Masahiro, Matt Mower, David Heinemeier Hansson, Hiroshi

Nakamura, John W. Small, Takahashi Masayoshi, Gotou Yuuzou, Yoshida Masato, and Grant McLean.

Please, stand up with me and give my reviewers a round of applause: Frank Tewissen, Matthias “Matze” Klame, Uwe Simon, and Kaan Karaca did an awesome job! Without their corrections and suggestions this book wouldn’t be half as good.

A loud “Thank you very much!!!!” goes to all the people who sent errata and suggestions during the beta book process: Lee Grey, Hoang Uong, Ola Bini, Ron Lusk, John Athayde, Blair Zajac, Jim Weirich, Pat Podenski, Gregory Brown, Lachlan Dowding, Sean, Eldon, Stuart Halloway, Raymond Brigleb, Ken Barker, Peter Morelli, Eric-Olivier Lamey, and Jim Kimball.

Perhaps there are authors who write books in isolation under a rock or on a lonesome island. Fortunately, I didn’t and got invaluable support from a lot of wonderful people. I am deeply grateful to my parents (this one’s for you), my sister Yvonne Janka (yet another book you won’t read?), my brother André Schmidt (for relaxing shopping/running tours and even more relaxing evenings with “the boys”), Christian & Agnieszka Rattat (for being true friends when I needed them most), Frank Tewissen (for listening patiently and for advising carefully), Manu (for being “die Manu”! Heja BVB!), AleX Reinartz (I’m looking forward to the next decades), Bettina Hamidian & Corinna Lorscheid (for insightful talks and lots of fun), Katja Wevelsiep (let’s have a coffee tomorrow, OK?), Frank Möcke (for giving me the opportunity to publish texts in my mother tongue), Dr. Andreas Kötz (for your appreciation), and to the “Gleis drei” staff (for providing a perfect proof-reading environment).

## Chapter 2

# Databases

---

Database management systems are one of the oldest and most widely used applications in information technology—they are indispensable to enterprises. It's nearly impossible to do some serious enterprise integration without touching some kind of database directly or indirectly. Various types exist (relational databases, object-oriented databases, directory services, XML databases, and hash databases such as Berkeley DB). They differ mainly in the way data is organized and accessed internally. Under the hood, though, they are all similar: data is stored in some kind of file system and is accessed through a special layer, often over a network. You can find one or more of the different types in every company, but relational databases are by far the most popular ones in use today.

Although it's often tedious, repetitive, and error-prone work, accessing databases is, in principle, easy. You open a connection, create and execute some statements, read and process some data, and finally free all resources occupied. At least, that's how the Gods Of Persistence wanted it to be. But real life in our sinful world looks different. Information and business logic is often spread across different schemas and databases. To make things even worse many companies use products from multiple vendors. This happens for various reasons: they want to prevent vendor lock-in, our company is the product of a corporate merger, different departments prefer different tools, and so on.

Unfortunately, PragBouquet is no exception. Its data is stored in both Oracle and MySQL databases. In this chapter we will show you not only how to directly manipulate different types of databases, but also how to access them using more advanced tools such as object-relational mappers and database abstraction layers.

## 2.1 The Coupon Application

PragBouquet's business has been doing well, but business can always be better, can't it? To boost sales, the marketing department wants to send a coupon to every customer who's used the online store, but hasn't used it in the last 6 months. People who have been asked not to be e-mailed should not get an e-mail.

That does not sound too difficult. PragBouquet already has a mass mailing program that expects a CSV (Comma Separated Values) file containing e-mail addresses, customer names, and a text to be sent. The problem becomes selecting names and e-mail addresses of all customers who did not place an order in the last 6 months, filtering out those who do not want to be e-mailed, and writing the rest to the CSV file.

Instantly you've fired up your favorite text editor thinking that this is a great opportunity to strengthen your Ruby skills. Creating CSV files is a breeze and selecting some data sets from a database should not be a problem, either. So you ask your database administrator where you can find the information you need and he takes you down a peg or two. He tells you that for historical reasons (an euphemism for "Nobody knows why") the information you need is spread across two databases. Customer data and order data are stored in an Oracle database, but the white list containing the e-mail addresses of all customers who want to receive e-mail from PragBouquet is stored in the web shop's MySQL database. You scribble a bit on your notepad and realize that the system architecture has to look like Figure 2.1, on the next page.

### Exploring The Environment

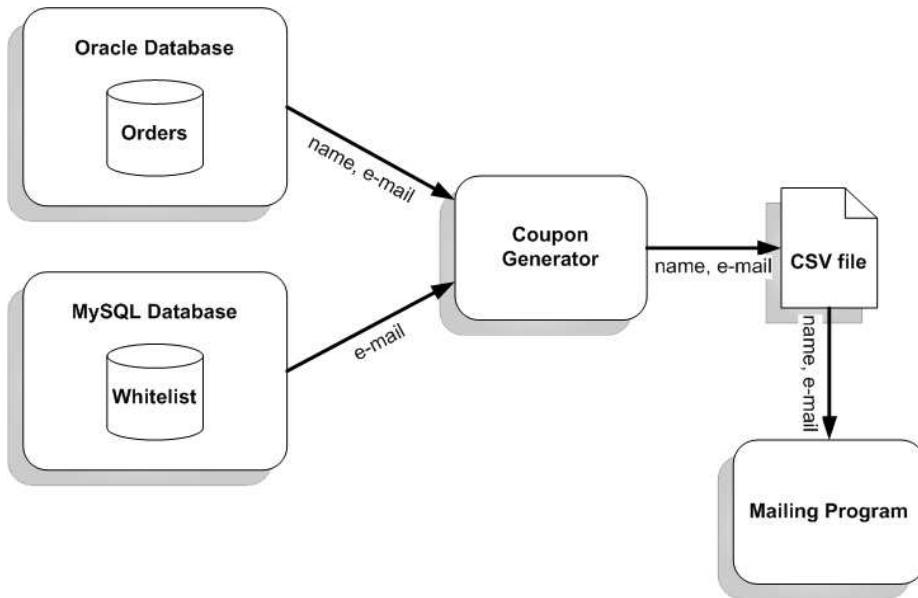
You decide to start with the Oracle part. Before moving on you want to have a closer look at the structure of the order database. Your database administrator told you that the relevant tables are called customers and orders. He gave you plenty of Microsoft Word documents describing every single table in the order database. Despite this you have a look at the current state of affairs yourself using SQL\*Plus, Oracle's SQL shell.

```
C:\> sqlplus scott
```

```
SQL*Plus: Release 9.2.0.1.0 - Production on Sat Jun 4 16:00:04 2005
```

```
Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.
```

```
Enter password: XXXX
```




---

Figure 2.1: Coupon Application Workflow

---

### Why didn't we use a standard product?

You might be asking yourself if it's a good idea for PragBoquet to have created its own customer and order databases? Wouldn't it be much easier to buy a solution off the shelf? Customer data is at the core of every enterprise and many processes rely upon it. It's needed for billing, for statistics, for troubleshooting, and so on. Although many big companies offer software for customer relationship management, it's never a bad idea to think about building your own customer database. No product will fit your needs better than your own and no product will ever be as flexible as yours.



```

Connected to:
Personal Oracle9i Release 9.2.0.1.0 - Production
With the Partitioning, OLAP and Oracle Data Mining options
JServer Release 9.2.0.1.0 - Production

```

```
SQL> describe customers
```

| Name         | Null?    | Type          |
|--------------|----------|---------------|
| -----        | -----    | -----         |
| ID           | NOT NULL | NUMBER(38)    |
| NAME         | NOT NULL | VARCHAR2(64)  |
| SURNAME      | NOT NULL | VARCHAR2(128) |
| STREET       | NOT NULL | VARCHAR2(128) |
| HOUSE_NUMBER | NOT NULL | VARCHAR2(10)  |
| POSTAL_CODE  | NOT NULL | VARCHAR2(10)  |
| CITY         | NOT NULL | VARCHAR2(128) |
| STATE        |          | VARCHAR2(20)  |
| COUNTRY_CODE | NOT NULL | VARCHAR2(2)   |
| EMAIL        | NOT NULL | VARCHAR2(128) |
| CREATED      |          | DATE          |

```
SQL> describe orders
```

| Name        | Null?    | Type         |
|-------------|----------|--------------|
| -----       | -----    | -----        |
| ID          | NOT NULL | NUMBER(38)   |
| CUSTOMER_ID | NOT NULL | NUMBER(38)   |
| STATE       | NOT NULL | NUMBER(38)   |
| CREATED     |          | TIMESTAMP(6) |

No big surprises here. Obviously, customers are characterized mainly by their address data and we guess that the tables are connected using column `customer_id` in table orders.

## Determine the Winners

If we're going to use a Ruby program to extract information from an Oracle database, we'll need a library that connects our code to the underlying Oracle API. There are currently three Ruby modules for Oracle:

- *Oracle* by Yoshida Masato <sup>1</sup>
- *Ruby/OCI8* by Kubo Takehiro <sup>2</sup>
- *Ruby9i* by Jim Kain <sup>3</sup>

---

<sup>1</sup><http://raa.ruby-lang.org/project/oracle>

<sup>2</sup><http://rubyforge.org/projects/ruby-oci8>

<sup>3</sup><http://rubyforge.org/projects/ruby9i>

### **Storing Addresses—A Plea from the Rest of the World**

Even though addresses are critical for many purposes, their data representation is often performed carelessly and without foresight. In particular, aspects of internationalization are often forgotten, because designers and developers normally do not know a lot about the administrative characteristics of their neighbors.

For example, Germany is a federal country divided into 16 states, but to the Germans the different states do not mean a lot. They aren't part of an address, they do not occur on envelopes, and you do not have to put them into a web form when ordering something from an internet shop. It's not surprising that German customers get annoyed by web forms insisting on a state. When working in an international environment, it's better to make the state optional.

There is no international standard for the representation of an address. In Germany, for example, a street address is *street name* followed by a blank followed by the house number. In Italy, there's a comma between the street name and the house number. Other countries put the number before the name.

It's nearly impossible to automatically separate street names and house numbers afterwards, because house numbers can contain nearly arbitrary characters.

Another aspect of addresses that is forgotten surprisingly often in this context is that addresses represent geographical objects. Geographical objects have coordinates, locations that are becoming increasingly important as we move into a world using location-based services. If you want to offer location based services to your customers some day you'll have to determine the geographical position of their addresses. For many cities it's possible to locate an object down to the individual house number.

Please, don't misunderstand me: you should not try to come up with a solution that will work with every possible address format in the world (I think that would probably be impossible), but you should at least have a closer look at the countries you're potentially working in.

The main difference between these libraries is their support (or lack thereof) for new data types. Gone are the days when you could only store small strings and numbers in your database. Nowadays you can store complete books or MP3 files in CLOB (*Character Large Object*) or BLOB (*Binary Large Object*) columns. Major versions of the *Oracle Call Interface* (OCI) also differ in other areas, such as security, performance etc.

*Character Large Object*

*Binary Large Object*

*Oracle Call Interface*

In this book we'll use Kubo Takehiro's Ruby/OCI8 driver—it's actively maintained, runs on many platforms, and provides a lot of functionality. It comes in two flavors: A low-level and a high-level API. The low-level API directly reflects the Oracle C library and we will not show its usage, as the high-level API is probably more convenient to use.

Let's dive into Ruby now and see how we can identify the customers who should get a coupon.

File

```
Line 1  require 'oci8'
-
-   connection = OCI8.new('maik', 'maik')
-   cursor = connection.exec(<<-SQL)
5   select a.id, a.name, a.surname, a.email
-   from   customers a, orders b
-   where  a.id = b.customer_id
-   and    b.created < sysdate - 180
-   and    b.created = (
10      select max(created)
-      from   orders
-      where  customer_id = a.id
-   )
-   SQL
15
-   while row = cursor.fetch do
-     puts row[3]
-   end
-
20  cursor.close
-   connection.logoff
```

produces something like this:

```
homer@example.com
seymour@example.com
```

Here we have a typical example of accessing a database. It would look similar in every modern programming language. First we establish a database connection by calling the `new()` method of class `OCI8` (`connect()` would have been a much better name, but for the moment we have to live with it). The `new()` method returns a connection object,

that can be used to communicate with the database server and to create other database objects, such as statements and cursors.

The SQL statement joins the tables `customers` and `orders` and returns only those customers whose last order is older than 180 days. The sub-select identifies the most current entry for each customer and makes sure that every customer is returned only once.

As you can see, SQL statements can be executed directly by calling the `exec()` method of an OCI8 connection. For `SELECT` statements, `exec()` returns a so-called cursor representing a result set on the database server. Clients can move through a result set by calling `fetch()` on the cursor object. After the last row has been read from the cursor `fetch()` returns `nil`.

Eventually, we close our cursor to free valuable resources on the database server. Cursors are resources like file handles, and are in limited supply. If you're a bad citizen and failed to free off these resources, Oracle will raise an exception sooner or later.

Admittedly, our example is concise and expressive, but using Ruby's iterators automatically leads to a more elegant solution with less explicit resource management:

```
File
Line 1  require 'oci8'
-
-   connection = OCI8.new('maik', 'maik')
-   sql = <<-SQL
5   select a.id, a.name, a.surname, a.email
-   from   customers a, orders b
-   where  a.id = b.customer_id
-   and    b.created < sysdate - 180
-   and    b.created = (
10      select max(created)
-      from   orders
-      where  customer_id = a.id
-   )
-   SQL
-
15
-   num_customers = connection.exec(sql) do |row|
-     puts row[3]
-   end
-
20  puts "Found #{num_customers} coupon recipients."
-   connection.logoff
```

produces:

```
homer@example.com
```

```
seymour@example.com
Found 2 coupon recipients.
```

When `exec()` is called as an iterator—with a code block—it returns the number of rows selected. The code block automatically gets each row fetched as a parameter and you no longer have to close the cursor explicitly. Actually, you don't even notice that you're working with a cursor.

## Enhancing Flexibility

OK, our first example works. We know where to get the data from and we know how to get it, so let's turn our little script into software. First of all, we have to replace the constant 180 days with something more dynamic. To do this, we could create the string containing the SQL statement on the fly, substituting in the time value, but this approach has some serious drawbacks.

As we already know, the SQL statement gets transferred over the network to the database server whenever we call `exec()`. Then it gets parsed, analyzed, optimized, executed, and eventually the result is sent back to the client.

Actually, modern database servers try to optimize a lot. Part of this process is the creation of a so-called *query execution plan* for every statement they receive. Current Oracle versions even try to compress the result sets before sending it back to the client to decrease bandwidth and processing time. For SQL statements that are executed often this means that we could gain a lot if the statement could be parsed, analyzed, and optimized only once.

*query execution plan*

Furthermore, building SQL statements on the fly often creates dangerous security holes. What if someone uses a web form to pass us the following string for the number of days?

```
'180; delete from customers; commit;'
```

In the worst case the database server will happily execute the malicious statement giving you an excellent opportunity to check if your backup system is working properly. This common kind of attack is called *SQL injection*.

*SQL injection*

Fortunately, it is possible to circumvent all these disadvantages by using so called *prepared statements*. We transmit a statement template to the server, where it is parsed, analyzed, and optimized. The server then sends back a statement handle. All the dynamic portions

*prepared statements*

of our statement are replaced by placeholders. Whenever we want to execute our statement, we only send the server the handle and the actual values for our placeholders.

File

```
Line 1  require 'oci8'
-
-  Customer = Struct.new(:id, :name, :surname, :email)
```

File

```
Line 1  class CustomerFinder
-      def initialize(connection)
-          @find_stmt = connection.parse(<<-SQL)
-          select a.id, a.name, a.surname, a.email
5         from   customers a, orders b
-          where  a.id = b.customer_id
-          and    b.created < sysdate - :days
-          and    b.created = (
-              select max(created)
10         from   orders
-          where  customer_id = a.id
-          )
-          SQL
-      end
15
-      def find(days)
-          @find_stmt.bind_param(':days', days)
-          @find_stmt.exec
-          customers = []
20         while row = @find_stmt.fetch do
-             customers << Customer.new(*row)
-         end
-         customers
-     end
25 end
```

First of all, we have inserted a placeholder (:days) into the SELECT statement. Then we create a prepared statement by calling `parse(sql)` on our connection. This method returns a handle identifying our statement on the server.

Calling `bind_param()` in line 17 binds the :days placeholder to its actual value and in the following line we finally execute the SELECT statement `@find_stmt` is referring to. The rest is business as usual. Using the `CustomerFinder` looks like this:

File

```
Line 1  ora_connection = OCI8.new('maik', 'maik')
-      finder = CustomerFinder.new(ora_connection)
-      customers = finder.find(180)
-      customers.each { |c| puts c.email }
5      ora_connection.logoff
```

## Respecting Customer Privacy

So far, so good. We can create a list of all customers that should potentially get a coupon, but we still have to sort out those who do not want to receive e-mails from PragBouquet. As we've already learned, this information is stored in the web shop's MySQL database. There we can find a table called `whitelist` containing a list of all e-mail addresses that we are allowed to use.

MySQL, created by Monty Widenius, is one of the most popular Open Source databases at the moment. It started as a thin wrapper for the `mSQL` database and has grown over the years into a full-blown transactional database management system. MySQL support in Ruby was made possible by the great work of Tomita Masahiro. He has developed both a C library binding called `MySQL/Ruby`<sup>4</sup> and a pure Ruby binding called `Ruby/MySQL`<sup>5</sup>. Thanks to a patch written by Matt Mower, `Ruby/MySQL` now also works with MySQL version 4.1.1 and later.<sup>6</sup>

In this book we'll use the pure Ruby implementation (for no special reason). As with our order database we first examine the webshop database using the MySQL shell:

```
C:\>mysql webshop
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3 to server version: 4.0.22-nt
```

```
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
```

```
mysql> describe whitelist;
+-----+-----+-----+-----+-----+-----+
| Field | Type                | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| id    | int(10) unsigned    |      | PRI | 0        |       |
| email | varchar(255)        |      | UNI |          |       |
| created | timestamp(14)      | YES  |     | NULL     |       |
+-----+-----+-----+-----+-----+-----+
3 rows in set (0.16 sec)
```

```
mysql>
```

As a first exercise we try to connect to the MySQL server and print the whole `whitelist`.

---

<sup>4</sup><http://tmtm.org/en/mysql/ruby/README.html>

<sup>5</sup>[http://www.tmtm.org/en/ruby/mysql/README\\_en.html](http://www.tmtm.org/en/ruby/mysql/README_en.html)

<sup>6</sup>This patch is part of Rails' ActiveRecord module, and can be found under <http://www.rubyonrails.org>.

File

```

Line 1  require 'mysql'
-
-  connection = Mysql.new('localhost', '', '', 'webshop')
-  whitelist = connection.query('select * from whitelist');
5  whitelist.each_hash { |h| puts h['email'] }
-  connection.close

```

produces something like this:

```

homer@example.com
info@example.net
...
c-m-burns@example.org

```

Here we have a textbook example of database use: create a connection, execute a query, print its result, and finally close the connection. What more could we say that hasn't already been expressed in the code? Alright, we have some details for you. Calling the `query(sql)` method returns an object of class `Mysql::Result` that represents a complete result set. You can read the single rows of a result set using various methods—here we chose `each_hash()`. It returns a Hash for every row where the column names are the hash keys with the data as the corresponding values.

Printing the whole whitelist was not exactly what we wanted. Instead we have to check whether a certain email address is contained in the whitelist. That means we have to execute a statement such as

```

select count(*)
  from whitelist
 where email = 'email@example.com'

```

and see if it returns 1. Obviously, the email address in the where clause of our statement is variable and from what we've learned in Section 2.1, *Enhancing Flexibility*, on page 15, you might assume it would be a good idea to use a prepared statement for this purpose. You are absolutely right: it would be a good idea, but unfortunately support for prepared statements in MySQL is a rather new feature. It was introduced in version 4.1 and the current Ruby drivers do not support it.

File

```

Line 1  require 'mysql'
-
-  class Whitelist
-    def initialize(connection) @connection = connection; end
5
-    def contains?(email)
-      sql = "select * from whitelist where email = '#{email}'"
-      result = @connection.query(sql)
-      result.num_rows == 1

```



```

10     end
-   end

```

Obviously, `num_rows()` returns the number of rows in a result set (which is what we wanted to determine). In use, our `Whitelist` class looks as follows.

File

```

Line 1  connection = Mysql.new('localhost', '', '', 'webshop')
-   whitelist = Whitelist.new(connection)
-   puts whitelist.contains?('homer@example.com')
-   puts whitelist.contains?('unknown_address')
5   connection.close

```

produces:

```

true
false

```

We've created our SQL statement using strings. Does it make you feel comfortable? Although the coupon application is an internal project, the e-mail addresses come from an external source and so you should never trust them. In addition, it's really wasteful to execute an SQL statement for every single e-mail address. So, we will trade some space for time and read all e-mail addresses into a Hash initially.

File

```

Line 1  require 'mysql'
-
-   class Whitelist
-   def initialize(connection)
5       @whitelist = {}
-       result = connection.query('select email from whitelist');
-       result.each_hash { |h| @whitelist[h['email']] = true }
-   end
-
10      def contains?(email)
-       @whitelist.has_key?(email)
-   end
-   end

```

That's a really good compromise. Even if we have to read several thousand e-mail addresses into memory, it's still a low price for the performance and security we get.

## Joining Forces

We have everything available now to create the list of our lucky coupon recipients: we can read all potential customers from the Oracle order database and can look them up on the white list stored in the MySQL webshop database. Because the mailing program expects data as CSV

we reopen the `Customer` class and add an appropriate method (see Section 3.5, *Comma-Separated Values (CSV)*, on page 129, to learn more about Ruby's CSV library).

File

```
require 'csv'
class Customer
  def to_csv(del = ',')
    str = ''
    CSV::Writer.generate(str, del) do |csv|
      csv << [name, surname, email]
    end
    str
  end
end
```

The following program then prints CSV data to the console so it can be easily redirected to the mass mailing program.

File

```
Line 1  require 'cusfinder'
-       require 'whitelist'
-
-       # Read all potential customers
5       ora_connection = OCI8.new('maik', 'maik')
-       finder = CustomerFinder.new(ora_connection)
-       customers = finder.find(180)
-       ora_connection.logoff
-
10      # Sort out customers not in whitelist
-       mysql_connection = Mysql.new('localhost', '', '', 'webshop')
-       whitelist = Whitelist.new(mysql_connection)
-
-       customers.each do |c|
15         puts c.to_csv if whitelist.contains?(c.email)
-       end
-
-       mysql_connection.close
```

produces:

```
Homer,Simpson,homer@example.com
Barney,Gumble,barney_gumble@example.org
...
Ned,Flanders,nflanders@example.net
```

That's it. We could happily move to the next project. But wouldn't it be interesting to know how many customers actually convert their coupon? To do this, we have to store at least the customer ids of all coupon recipients somewhere. Let's put it into the order database in a new table called `coupon_recipients`. This will let us check to see how many of the customers on this list placed an order after the coupon mailing.

File

```
create table coupon_recipients (
  customer_id int not null,
  created timestamp default sysdate
);
```

For the first time in this chapter we're going to write data into the database. It's nearly the same as reading information, but there are a few subtleties we have to take care of.

File

```
Line 1  require 'oci8'
-
-  class Recipient
-    def initialize(connection)
5      sql = 'insert into coupon_recipients (customer_id) values(:1)'
-      @insert_rec = connection.parse(sql)
-    end
-
-    def create(customer)
10     @insert_rec.bind_param(1, customer.id)
-     @insert_rec.exec
-    end
-  end
```

Here, we've used another form of bind variable, numbering them instead of naming them explicitly. It's more or less a matter of taste whether you bind parameters by name or by number, but you have to be consistent. If you've used numbers as placeholders for the parameters in the SQL statement, you have to bind them by number later. That's especially important for output parameters:

File

```
Line 1  connection = OCI8.new('maik', 'maik')
-  cursor = connection.parse("begin :now := sysdate; end;")
-  cursor.bind_param(':now', Time.mktime(1972, 9, 30), Date)
-  puts cursor[':now']
5  puts cursor[1]
-  cursor.exec
-  puts cursor[':now']
-  puts cursor[1]
-  cursor.close
10  connection.logoff
```

at the time of this writing produces:

```
1972-09-30
nil
2005-04-03
nil
```

There's something even more critical hidden in our Recipient class.

File

```
Line 1  require 'cusfinder'
-  connection = OCI8.new('maik', 'maik')
```

```

- connection.autocommit = true
- recipients = Recipient.new(connection)
5 customer = Customer.new(1, 'Selma', 'Bouvier', 'selma@example.com')
- recipients.create(customer)
- connection.logoff

```

See that we've enabled the *auto-commit* feature of the connection object on line 3. This makes sure that every SQL statement gets committed immediately, saving any changes to the database when the statement is executed. That's what we'd normally expect to happen.

*auto-commit*

Oracle is a transactional database—you can group several SQL statements as if they were one. If any of the statements fail, all the statements will be ignored—the database content will not be changed. The current transaction can be committed by executing the COMMIT command or it can be rolled back by calling ROLLBACK. Setting autocommit to true is like calling COMMIT after every single SQL statement. Without it, nothing would ever get written to the database. You wouldn't even notice it, because from the database's point of view it's not an error.

Our final version of the coupon application differs only slightly from our original approach:

File

```

Line 1  require 'cusfinder'
-       require 'whitelist'
-       require 'recipient'
-
5       # Read all potential customers
-       ora_connection = OCI8.new('maik', 'maik')
-       ora_connection.autocommit = true
-       recipients = Recipient.new(ora_connection)
-       finder = CustomerFinder.new(ora_connection)
10      customers = finder.find(180)
-
-       # Sort out customers not in whitelist
-       mysql_connection = Mysql.new('localhost', '', '', 'webshop')
-       whitelist = Whitelist.new(mysql_connection)
15
-       customers.each do |c|
-         if whitelist.contains?(c.email)
-           puts c.to_csv
-           recipients.create(c)
20       end
-       end
-
-       ora_connection.logoff
-       mysql_connection.close

```

The most important changes affect the Oracle connection object. We've

set its autocommit feature to true. We also defer closing the connection until the end of the program, as it's needed during the whole runtime.

## The Fruits of Our Labor

Two weeks ago the coupons were sent to their lucky recipients. Today started like any other: switched on your PC and went into the kitchen to get a (free) cup of coffee. As you came back to your desk to create yet more extraordinary code, one of the marketing guys was waiting for you. “You’re the techie that sent out the coupons two weeks ago, aren’t you?” he asks. Before you can say a word he proceeds: “Although we worked several weeks on the functional specification of the coupon application, we somehow forgot to define some statistics requirements. Now we’re afraid that we can’t find out how successful our *marvelous* and *groundbreaking* coupon idea was. Is there any way you could create some statistics, anyhow?”

Mostly, you’re surprised that something like a functional specification exists—it’s the first you heard of it. But, when you recover, you remember the `coupon_recipients` table and open an SQL\*Plus shell:

```
SQL> select count(*) from coupon_recipients;

COUNT(*)
-----
      3145

SQL> select count(*) from orders where customer_id in (
2   select customer_id from coupon_recipients
3 ) and created > sysdate - 14;

COUNT(*)
-----
       917

SQL> select 917 * 100 / 3145 from dual;

917*100/3145
-----
      29.1573927

SQL>
```

Turning around to the marketing guy you say: “29.16% of the coupon recipients placed an order during the last two weeks. Do you need anything else?” He is obviously impressed: “No, thank you very much!”

## Managing Database Resources

So far, our examples have been simple and we didn't care about performance and optimization. But opening a new database connection is expensive and should not be performed unnecessarily. If you only need a single connection, databases can be represented as a singleton object. A singleton object is available everywhere in your program and can be created only once. Thanks to the Ruby standard library it's a piece of cake to create a singleton encapsulating our OCI8 driver:

File

```
require 'oci8'
require 'singleton'

class Database
  include Singleton

  attr_reader :connection

  def initialize
    @connection = nil
  end

  def connect(usr, pwd, dbname = nil)
    @connection = OCI8.new(usr, pwd, dbname)
    @connection.autocommit = true
    @connection
  end

  def disconnect
    if !@connection.nil?
      @connection.logoff
      @connection = nil
    end
  end
end
```

Class Database makes a connection to our database available wherever we need it and we get access to the one and only instance by calling `Database.instance()`. At program start we have to call `Database.instance.connect(usr,pwd)` once and from then on `Database.instance.connection` contains our connection.

You did an awesome job and I wouldn't be surprised, if you get a corner office soon." You lean back and take a sip of your coffee. It's still hot.

## 2.2 Database Interface (DBI)

It's a bit annoying that the information we needed for our coupon application is spread across two databases—it might be a good idea to change this situation someday. Anticipating this change, it might be advantageous to make our application more independent of the underlying drivers. As we've seen in the previous sections, accessing databases using native drivers in principle differs only slightly from vendor to vendor: you have to obtain a connection, create or prepare statements, execute statements, and retrieve results eventually. Technically, though, there are many subtle (and sometimes not so subtle) differences. Countless attempts have been made to standardize this interface. For example, on the Microsoft Windows platform there is ODBC, OLE DB, and ADO.NET, to name just a few. Java has its JDBC and dynamic languages such as Perl, Python, and Ruby use an approach called *DBI* (Database Interface).<sup>7</sup>

DBI

All *database abstraction layers* work in a similar fashion: they define an abstract interface to the database, and a concrete implementation, called a *database driver*, is implemented for each specific database. For the Ruby DBI library, these drivers are known as DBD modules.<sup>8</sup> These drivers are accessed by your program through a standard interface,<sup>9</sup> so you do not have to remember if the method to get a new connection was called `new()`, `connect()`, `create_connection()`, or whatever. In DBI it's called `connect(driver_url, user=nil, auth=nil, params=nil)` for every database supported and it always expects the same parameters in the same order.

database abstraction  
layers

database driver

Compared to other database abstraction layers, DBI is extremely simple. To use it you only have to know two classes, `DatabaseHandle` and `StatementHandle`. A database handle represents a connection to the database, while a statement handle represents an active SQL statement. To examine whether we can benefit from using DBI in our PragBouquet application, we'll change the `Whitelist` class to use it.

File

```
Line 1  require 'dbi'
-
-  DBI.connect('DBI:MySQL:webshop', '', '') do |conn|
-    conn.select_all('select * from whitelist') { |row| p row }
```

<sup>7</sup>This list proves the old adage: the good thing about standards is that there're so many to choose from.

<sup>8</sup>[http://ruby-dbi.rubyforge.org/DBD\\_SPEC.html](http://ruby-dbi.rubyforge.org/DBD_SPEC.html)

<sup>9</sup>[http://ruby-dbi.rubyforge.org/DBI\\_SPEC.html](http://ruby-dbi.rubyforge.org/DBI_SPEC.html)

5    **end**

Because of the block syntax supported by the DBI methods, our demonstration program became extremely compact. In line 3, `DBI.connect()` returns a database handle that gets passed into the block. When the program reaches the end of the block, the connection is closed automatically. Within the block we call `select_all()`, which executes a `SELECT` statement and calls a code block for every row that was returned. Again, we do not have to care about resource management—the statement will be released at the end of the block. The only thing left to do is to integrate the code into the `Whitelist` class.

```
File
Line 1  require 'dbi'
-
-  class Whitelist
-    def initialize(connection)
5      @whitelist = {}
-      connection.select_all('select email from whitelist') do |row|
-        @whitelist[row[0]] = true
-      end
-    end
-  end
10
-  def contains?(email) @whitelist.has_key?(email); end
-  end
```

We did not change the interface and only the connection object has to be instantiated differently to use the `Whitelist` class:

```
File
Line 1  connection = DBI.connect('DBI:Mysql:webshop', '', '')
-  whitelist = Whitelist.new(connection)
-  puts whitelist.contains?('homer@example.com')
-  connection.disconnect
```

Should we move the `whitelist` table from MySQL to our Oracle database, we only have to change the string “Mysql” to “Oracle” and the program will still work.

Encouraged by our success, we’ll change the Oracle stuff in our `CustomerFinder` class to use DBI too.

```
File
Line 1  class CustomerFinder
-    def initialize(connection)
-      @find_stmt = connection.prepare(<<-SQL)
-      select a.id, a.name, a.surname, a.email
5      from   customers a, orders b
-      where  a.id = b.customer_id
-      and    b.created < sysdate - :days
-      and    b.created = (
-        select max(created)
10      from   orders
```



```

-         where customer_id = a.id
-     )
-     SQL
- end
15
- def find(days)
-     @find_stmt.bind_param(':days', days)
-     @find_stmt.execute
-     customers = []
20     while row = @find_stmt.fetch do
-         customers << Customer.new(*row)
-     end
-     customers
- end
25 end

```

As with the previous example, we did not have to change a lot. Instead of calling `parse()` on our connection object in line 3, we have to call `prepare()` now. Similarly, `exec()` becomes `execute()` on line 18. We have to pass a DBI connection object now:

File

```

Line 1 connection = DBI.connect('DBI:Oracle', 'maik', 'maik')
- finder = CustomerFinder.new(connection)
- customers = finder.find(180)
- customers.each { |c| puts c.email }
5 connection.disconnect

```

Despite all this, the benefits of a database abstraction layer aren't as big as you might think. It's convenient to work with DBI when you have to access a database product that you haven't worked with before, but you shouldn't assume that you can easily replace your existing database by a completely different one only because you're using an abstraction layer. Moving from one database to another is one of the most complicated things in developing enterprise software.

Because there are so many proprietary additions to SQL in every vendor's implementation, writing portable statements is nearly impossible. Often such statements look quite harmless. For example, look at the statement starting on line 3 in our `CustomerFinder` class. It contains at least three potential problems:

- Not all databases support sub-selects.
- `sysdate` is specific to Oracle. In MySQL you'd have to use `now()` and in DB2 it'd be `current timestamp`.
- The syntax of arithmetic expressions for dates (such as `sysdate-180`) differs from vendor to vendor.

Sometimes the problems aren't directly related to a SQL statement, but are caused by some side-effects like auto-generated identifiers which aren't available in every database. To support such database specific functions, the drivers used by DBI allow for some extensions, but if you want to write portable software, it's certainly not a good idea to use them. For example, to read the last auto-generated identifier from a MySQL database, you call the `last_insert_id()` method. This method is not available for Oracle databases and it's not easy to simulate the auto-generation feature in Oracle.

A last problem with DBI could be performance: the extra layers and the need to map features can decrease performance significantly. For example, accessing MySQL using the native driver is twice as fast as using the DBI layer.

There are much more important (and tricky) issues that might prevent you from easily changing your database. Consider, for example, C/C++ programs that contain *Embedded SQL*. Even if you're lucky and have access to the source code of all programs running in your environment, it still will be a lot of work to adjust them all.

*Embedded SQL*

So, if you know up front that you have to support multiple databases you can gain a lot by using an abstraction layer, but you have to plan for it carefully.

## 2.3 Object-Relational Mappers

A lot of people working in the software development department of Prag-Bouquet have been thinking about re-organizing the current database landscape for quite a long time. The design of many databases has become a bit messy over the years and it's a big problem that logic and data are spread across Oracle and MySQL databases. To save license costs, all the Oracle databases should be migrated to a MySQL database in the future and all new stuff should be implemented in the MySQL database right from the beginning.

The first thing that has to be added is an automatic management system for ordering flowers. Today flowers are ordered from a big wholesaler more or less manually by the buying department. The clerks get daily order reports and they can see how many flowers are still in stock. Then they do some simple calculations using a spreadsheet application and place new orders accordingly. It's your task now to automate this process as far as possible, i.e. to create a database for the flowers in

### Generating Unique Ids

It's really strange: mankind is talking about going to Mars, but creating artificial primary keys in databases is still a problem in the 21<sup>st</sup> century, because there's no standard.

From a design point of view, there are a lot of advantages to creating an artificial unique (numeric) primary key for every table in the database, even if a natural primary key does exist. Numeric values only need a small amount of space and can be indexed efficiently.

Although there's a need for unique ids in every database, all vendors come up with their own ideas and concepts to generate them. It's easy to generate them more or less portable by creating a table containing only two columns:

```
create table sequences (
    value int default 1 not null,
    table_name varchar(64)
);
```

To create a sequence for our customers table, we insert a new row into the sequences table:

```
insert into sequences (table_name) values ('customers');
```

Generating a new sequence value is straightforward, then:

```
begin
    update sequences set id = id + 1
    where table_name = 'customers';
    select id from sequences
    where table_name = 'customers';
end;
```

Unfortunately, this solution is not particularly efficient, because it has to be executed in a transaction that can slow down things a bit. Oh, and did I mention that not all databases support transactions?

Whenever your program relies upon auto-generated identifiers you should encapsulate this process carefully to prevent bad surprises when you have to migrate to another database.

stock and to remove flowers from stock whenever a new bouquet leaves PragBouquet.

Before opening your text editor you take a day off to think about the new database structure and after 24 hours of constant thinking you finally had this revolutionary idea: we need a table that represents flowers:

File

```
Line 1  create table flowers(
-       id int unsigned not null auto_increment primary key,
-       name varchar(64) not null,
-       price double not null
5  );
```

That should be sufficient for a first version: flowers have a name, a price, and an artificial primary key that is created by the database automatically. The “only” thing left to do is mapping the flowers table to a Flower class and mapping all its columns to the according attributes.

You have read Martin Fowler’s *Patterns of Enterprise Application Architecture* [?] and you still remember his Active Record pattern and its definition:

“An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data.”

Before programming an Active Record for the flowers table we encapsulate access to the MySQL database in a singleton first:

File

```
Line 1  require 'singleton'
-       require 'mysql'
-
-       class Database
5         include Singleton
-         attr_reader :connection
-
-         def initialize() @connection = nil; end
-
10        def connect(host, usr, pwd, db= nil)
-          @connection = Mysql.new(host, usr, pwd, db)
-          end
-
-         def disconnect
15          @connection.close if !@connection.nil?
-          @connection = nil
-          end
-         end
```

Fine, after calling Database.instance.connect() once, we can access the database connection calling Database.instance.connection() from any-

where we want. So, let's use it to create new flowers:

File

```

Line 1  class Flower
-       attr_reader :id
-       attr_accessor :name, :price
-
5       def Flower.create(name, price)
-       Database.instance.connection.query(<<-SQL)
-       insert into flowers (name, price)
-       values ('#{name}', #{price.to_f})
-       SQL
10      Flower.new(
-       Database.instance.connection.insert_id,
-       name,
-       price
-       )
15      end
-
-       def to_s
-       "A #{@name} (#{@id}) costs $#{@price}."
-       end
20
-       private
-
-       def initialize(id, name, price)
-       @id, @name, @price = id, name, price
25      end
-      end

```

Virtually planting a rose looks like this:

File

```

Line 1  Database.instance.connect('localhost', '', '', 'webshop')
-       rose = Flower.create('rose', 1.99)
-       puts rose

```

and produces:

```
A rose (1) costs $1.99.
```

The first version of the Flower class allows for creating new objects by calling `create(name,price)`. This method inserts a new row into the database, reads the id that has been generated by MySQL and returns a new Flower object. To make sure that no conflicts happen in the database because of duplicate id values, we have declared the `initialize()` method private. Hence, only methods of the Flower class can create new objects.

For the sake of completeness we add the remaining methods needed to be fully *CRUD* compliant (*CRUD* stands for Create, Retrieve, Update, Delete):

```

File
Line 1  class Flower
-       def Flower.find(name)
-         flower = Database.instance.connection.query(<<-SQL)
-         select id, name, price from flowers where name = '#{name}'
5         SQL
-         return nil if flower.num_rows == 0
-         attr = flower.fetch_hash
-         Flower.new(attr['id'], attr['name'], attr['price'])
-       end
10
-       def update
-         Database.instance.connection.query(<<-SQL)
-         update flowers set name = '#{@name}', price = '#{@price}'
-         where id = #{@id}
15         SQL
-       end
-
-       def destroy
-         Database.instance.connection.query(<<-SQL)
20         delete from flowers where id = #{@id}
-         SQL
-       end
-     end
end

```

Now we can retrieve, update, and delete Flower objects in the database:

```

File
Line 1  rose = Flower.find('rose')
-       rose.price = 2.49
-       rose.update
-
5       rose = Flower.find('rose')
-       puts rose
-       rose.destroy
-       puts Flower.find('rose')
-       Database.instance.disconnect

```

produces:

```

A rose (1) costs $2.49.
nil

```

It took less than an hour to create the Active Record and it works fine, but despite all this you still think that sometimes life isn't fair: all your friends are hanging around at the beach having fun and you're writing tons of boring SQL statements only to read and save Flower objects. Enough is enough and hence you decide to look for a tool that will do all this tedious stuff for you.

### Object-Relational Mappers for Ruby

Because of its dynamic nature Ruby is a perfect language for creating tools like object-relational mappers: you can easily create classes and methods on the fly and determining the structure of a database is not a big problem with most database systems, either.

Unsurprisingly, several projects have been initiated to implement an object-relational mapper\*, but ActiveRecord is by far the most popular and most advanced. It's much more than a simple mapper, it's fast, it supports nearly every database available, and it is constantly enhanced by a big community.

\*Kansas (<http://raa.ruby-lang.org/project/kansas>) is an interesting one, for example.

### ActiveRecord Basics

ActiveRecord is an enhanced implementation of Martin Fowler's *Active Record* object-relational mapping pattern.<sup>10</sup> ActiveRecord was created by David Heinemeier Hansson because he needed it for the famous Ruby on Rails project<sup>11</sup>. ActiveRecord now supports nearly every database system currently in use (MySQL, PostgreSQL, SQLite, Microsoft SQL Server, Oracle, and DB2).

Code always trumps prose, so instead of explaining academic persistence strategies, let's start by telling ActiveRecord to connect to our database:

File

```
Line 1  require 'rubygems'
-      require 'active_record'
-
-      ActiveRecord::Base.establish_connection(
5         :adapter => 'mysql',
-         :host => '127.0.0.1',
-         :database => 'webshop'
-      )
```

These statements load the ActiveRecord gem (see Section 6.4, *RubyGems*, on page 288 to learn more about RubyGems) then establish a connection to the webshop database running on localhost.

<sup>10</sup><http://www.martinfowler.com/eaCatalog/activeRecord.html>

<sup>11</sup><http://www.rubyonrails.com>

Now we have to map the flowers table to a Ruby class called Flower:

File

```
Line 1  class Flower < ActiveRecord::Base; end
```

That's it! All we had to do is derive our class from ActiveRecord::Base. Every instance of class Flower represents a single row of the flowers table.

ActiveRecord derives the name of the database table by taking the class name, turning it into lowercase, and pluralizing it. For example, Flower becomes flowers and PragmaticProgrammer becomes pragmatic\_programmers. If necessary, you can also set the table name explicitly, either because the built-in pluralization rules don't work for you or because you want to map to an existing table whose name doesn't meet ActiveRecord's expectations.

```
class LegacyTable < ActiveRecord::Base
  set_table_name 'xy12aj'
end
```

All Flower objects automatically have accessors for all the columns of the flowers table, so there'll be accessors named name() and price():

File

```
Line 1  flower = Flower.new
-       flower.name = 'primrose'
-       flower.price = 0.99
```

ActiveRecord stores all columns internally in a hash called attributes, but using this knowledge is dangerous, as it links us to ActiveRecord's implementation. Instead, we should access column values using just the attributes. For example, we could add a to\_s() method to our class.

File

```
Line 1  class Flower < ActiveRecord::Base
-       def to_s
-         "A #{self.name} (#{self.id}) costs $#{self.price}."
-       end
5       end
```

In addition, ActiveRecord creates methods for reading, updating, and deleting rows in the database. To initialize the flowers table with some lovely plants, we can do the following:

File

```
Line 1  [
-       ['rose', 1.10],
-       ['violet', 0.40],
-       ['sunflower', 0.40],
5       ['clove', 0.65],
-       ['lily', 0.80]
-     ].each do |name, price|
-       flower = Flower.new(:name => name, :price => price)
-       flower.save
10      end
```



The database will then contain the following rows:

```
mysql> select * from flowers;
+-----+-----+-----+
| id | name      | price |
+-----+-----+-----+
| 1 | rose      | 1.1   |
| 2 | violet    | 0.4   |
| 3 | sunflower | 0.3   |
| 4 | clove     | 0.65  |
| 5 | lily      | 0.8   |
+-----+-----+-----+
5 rows in set (0.00 sec)
```

It's important that every table that is mapped has an id column that gets filled automatically when a new row is created (the column name is id by default, but like the table name you can change it calling `set_primary_key(column_name)`).<sup>12</sup>

For reading *all* entities belonging to a particular class, we use ActiveRecord's `find(*args)` method and pass it the `:all` option:

File

```
Line 1  Flower.find(:all).each do |f|
-      puts "A #{f.name} costs ${f.price}."
-      end
```

This gives us a list of all flowers:

```
A rose costs $1.1.
A violet costs $0.4.
A sunflower costs $0.3.
A clove costs $0.65.
A lily costs $0.8.
```

As well as some methods that perform more fine-grained searches, ActiveRecord dynamically defines *finder methods* for each column and for all combinations of columns. This example finds a flower by its name:

File

```
Line 1  rose = Flower.find_by_name('rose')
```

The following statement

File

```
Line 1  Flower.find_all_by_price(0.4).each { |f| puts f }
```

returns a list of flowers costing \$0.40 each:

```
A sunflower (48) costs $0.4
A violet (47) costs $0.4
```

---

<sup>12</sup>Not all databases support columns that get incremented automatically. For details see the documentation of the ActiveRecord adapter for your database system.

Finally we can search for flowers having a particular name and price:

File

```
Line 1  clove = Flower.find_by_name_and_price('clove', 0.65)
```

All the `find()` methods we have used above are created automatically—there is a `method_missing()` handler that turns your method calls into SQL statements.

If searching for columns having certain values is not sufficient, you can use another variant of `find()` that accepts a list of conditions as they may appear in the `WHERE` clause of a `SELECT` statement:

File

```
Line 1  cheap = Flower.find(:all, :conditions => ['price < 0.8'])
-      cheap.each { |f| puts f }
```

prints:

```
A clove (49) costs $0.65
A sunflower (48) costs $0.4
A violet (47) costs $0.4
```

To prevent SQL injection attackss in the condition string you can use a variant of the `find()` method that supports placeholders. The following code produces the same result as the example above:

File

```
Line 1  cheap = Flower.find(:all, :conditions => ['price < ?', 0.8])
-      cheap.each { |f| puts f }
```

You don't like sunflowers? Destroy 'em:

File

```
Line 1  sunflower = Flower.find_by_name('sunflower')
-      sunflower.destroy
```

ActiveRecord gives us the CRUD methods (Create, Read, Update, and Delete) for free—this alone is a great benefit. But ActiveRecord does a lot more. In the next section we'll see how it helps us express the relationships between tables.

## Cultivating Relationships With Flowers

Now that we have so many flowers, we should try to make some wonderful bouquets out of them. We'll need another table to represent these. Because we know that we will access it with ActiveRecord, we'll call it `bouquets`. Bouquets have a name and a base price (which covers wrapping paper, florist work, and so on):

File

```
create table bouquets(
  id int unsigned not null auto_increment primary key,
  name varchar(64) not null,
  base_price double not null
);
```

The relationship between bouquets and flowers is simple: bouquets have many flowers and flowers may belong to several different bouquets. (Obviously what we mean here is that roses may appear in many bouquets, not that a particular rose is in many direct bouquets. The latter could lead to some interesting problems on Valentine's Day.) SQL pros call this a *many-to-many relationship*. To implement it, they usually introduce a *join-table* (see Figure 2.2, on the next page), that maps rows from one table to the other and vice versa:

*many-to-many  
relationship*  
*join-table*

File

```
create table bouquets_flowers(
  bouquet_id int not null,
  flower_id int not null
);
alter table bouquets_flowers add constraint fk1_bouquet
  foreign key (bouquet_id) references bouquets(id);
alter table bouquets_flowers add constraint fk2_flower
  foreign key (flower_id) references flowers(id);
```

The table name is no coincidence. We want to implement the relationship between the bouquets and the flowers table. By default, ActiveRecord searches for a table that is named after the tables to be joined. Concatenate the table names in lexical order and separate them by an underscore, and you have the name of your join table. In addition, the join columns follow a naming scheme, too. You have to append “\_id” to the class name in lowercase letters (camel case is turned into underscores).

When you adhere to the naming rules, ActiveRecord makes it easy to express the relationship between flowers and bouquets using special macros:

File

```
class Bouquet < ActiveRecord::Base
  has_and_belongs_to_many :flowers
end

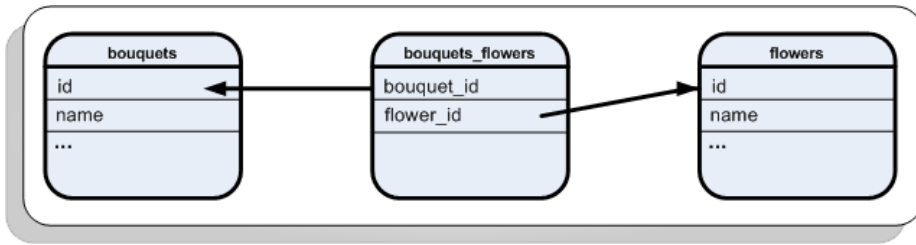
class Flower < ActiveRecord::Base
  has_and_belongs_to_many :bouquets
end
```

Many-to-many relationships are symmetrical, so we have to use the `has_and_belongs_to_many()` macro in both the Flower and Bouquet classes.

The following code snippet creates a new bouquet and stores it in the database:

File

```
Line 1  rose = Flower.find_by_name('rose')
- bouquet = Bouquet.new
- bouquet.name = 'Red Dream'
```




---

Figure 2.2: Join Table for Bouquets and Flowers

---

```

- bouquet.base_price = 2.49;
5 bouquet.save
- 6.times { bouquet.flowers << rose }
- bouquet.flowers << Flower.find_by_name('sunflower')

```

In line 6, we add six roses to our bouquet. In the following line we add a single sunflower. The flowers member of the Bouquet class was created automatically after we called `has_and_belongs_to_many()`. It acts as an array, except that whenever we add a flower to it it creates a new entry in the join-table.

After storing the bouquet the database content looks as follows:

```
mysql> select * from flowers where name in ('rose', 'sunflower');
```

```

+-----+-----+
| id | name      | price |
+-----+-----+
|  1 | rose      |  1.1  |
|  3 | sunflower |  0.3  |
+-----+-----+
2 rows in set (0.02 sec)

```

```
mysql> select * from bouquets;
```

```

+-----+-----+-----+
| id | name      | base_price |
+-----+-----+-----+
|  1 | Red Dream |    2.49    |
+-----+-----+-----+
1 row in set (0.00 sec)

```

```
mysql> select * from bouquets_flowers;
```

```

+-----+-----+
| bouquet_id | flower_id |
+-----+-----+
|           1 |          1 |
|           1 |          1 |
+-----+-----+

```

```

|          1 |          1 |
|          1 |          1 |
|          1 |          1 |
|          1 |          1 |
|          1 |          3 |
+-----+-----+
7 rows in set (0.00 sec)

```

As expected, the `bouquets_flowers` table contains seven rows describing the content of the “Red Dream” bouquet. Six of them refer to the rose entry in the `flowers` table, and one refers to the sunflower. To read all components with SQL we’d need a `SELECT` statement with a `JOIN` clause. ActiveRecord does this dirty work transparently.

File

```

b = Bouquet.find_by_name('Red Dream')
puts "#{b.name}' contains #{b.flowers.size} flowers:"
b.flowers.each { |f| puts f.name }

```

These statements disassemble our bunch of flowers:

```

'Red Dream' contains 7 flowers:
sunflower
rose
rose
rose
rose
rose
rose

```

This is nice and easy, but it’s also a bit messy. Instead of storing every rose in a single row, it would be better to store the information that there are six roses in this bouquet. To do this, we’ll add another piece of information to the relationship between flowers and bouquets:

File

```

alter table bouquets_flowers add (quantity int not null);

```

The join-table `bouquets_flowers` now also stores the quantity of a particular flower that belongs to a bouquet. To populate this column we use the `push_with_attributes()` method:

File

```

Line 1  rose = Flower.find_by_name('rose')
        - sunflower = Flower.find_by_name('sunflower')
        - bouquet = Bouquet.new
        - bouquet.name = 'Honeybunch'
5       bouquet.base_price = 1.37;
        - bouquet.save
        - bouquet.flowers.push_with_attributes(rose, :quantity => 1)
        - bouquet.flowers.push_with_attributes(sunflower, :quantity => 6)

```

We can read the information back from the database (remember, we’re still working with a database):

```
File
Line 1  b = Bouquet.find_by_name('Honeybunch')
        puts "#{b.name}' contains #{b.flowers.size} different flowers:"
        b.flowers.each { |f| puts "#{f.name} (#{f.quantity})" }
```

This prints a compact recipe for the “Honeybunch” bouquet:

```
'Honeybunch' contains 2 different flowers:
sunflower (6)
rose (1)
```

The join-table looks like this:

```
mysql> select * from bouquets_flowers;
+-----+-----+-----+
| bouquet_id | flower_id | quantity |
+-----+-----+-----+
|          2 |          3 |          6 |
|          2 |          1 |          1 |
+-----+-----+-----+
2 rows in set (0.00 sec)
```

It’s easy to add more logic to our domain classes, such as a method that calculates the price of a bouquet:

```
File
Line 1  class Bouquet
        def price
          flowers.inject(self.base_price) { |total, f|
            total += f.price * f.quantity.to_i
          }
        end
      end
      b = Bouquet.find_by_name('Honeybunch')
      puts "#{b.name}' costs ${b.price}."
```

For our “Honeybunch” bouquet it prints:

```
'Honeybunch' costs $4.87.
```

Let’s summarize what we’ve done so far: we created three database tables that represent flowers, bouquets, and their many-to-many relationship. The table names and the names of their columns conformed to a fairly natural convention, one that you may have used anyway. To map them to a Ruby class hierarchy that lets us manipulate them in every imaginable way, we needed six lines of code:

```
File
Line 1  class Bouquet < ActiveRecord::Base
        has_and_belongs_to_many :flowers
      end
      class Flower < ActiveRecord::Base
        has_and_belongs_to_many :bouquets
      end
```

## What Do We Have In Stock?

We shouldn't forget that we originally wanted to automate the ordering process for flowers. We still have to implement the concept of *stock*:

File

```
create table stock_items(
  id int unsigned not null auto_increment primary key,
  quantity int not null
);
```

Each row of the `stock_items` table represents the quantity of a particular flower that we currently have in stock—there is a stock item for every flower and vice versa.<sup>13</sup> This is a classic *one-to-one relationship*, that can be modeled in the database like this:

File

```
alter table flowers add (stock_item_id int not null);
alter table flowers add constraint fk1_flowers
  foreign key (stock_item_id) references stock_items(id);
```

Here's the Ruby model:

File

```
Line 1  class Bouquet < ActiveRecord::Base
-       has_and_belongs_to_many :flowers
-       end
-
5       class Flower < ActiveRecord::Base
-       has_and_belongs_to_many :bouquets
-       belongs_to :stock_item
-       end
-
10      class StockItem < ActiveRecord::Base
-       has_one :flower
-       end
```

It reads a bit like the notes you typically scribble on a piece of paper when thinking about a new database, doesn't it? Despite that, it's actual code that runs. In line 7 we declare that every flower belongs in stock and line 11 makes every stock item a flower.

Like the `has_and_belongs_to_many()` macro, the `has_one()` and `belongs_to()` macros automatically create some methods for the classes that use them. Let's fill our stock with some flowers:

File

```
Line 1  [
-       ['rose', 1.10, 1000],
-       ['violet', 0.40, 2000],
-       ['sunflower', 0.40, 500],
5       ['clove', 0.65, 2000],
```

<sup>13</sup>We could have added the quantity column to the `flowers` table, but in the future we will stock more than flowers.

```

-     ['lily', 0.80, 1500]
-   ].each do |name, price, quantity|
-     si = StockItem.new(:quantity => quantity)
-     si.save
10    Flower.new(
-      :stock_item => si,
-      :name => name,
-      :price => price
-    ).save
15  end

```

Note that in line 11 we have connected a particular flower with the current stock item by setting its `stock_item` attribute.

We can print a stock report like this:

File

```

Line 1  puts "Stock items:"
-      StockItem.find(:all).each { |si|
-        puts "#{si.flower.name}: #{si.quantity} in stock."
-      }

```

It produces:

```

Stock items:
rose: 1000 in stock.
violet: 2000 in stock.
sunflower: 500 in stock.
clove: 2000 in stock.
lily: 1500 in stock.

```

This time the magic is hidden in line 3. Obviously, ActiveRecord has added a `flower` member to the `StockItem` class that contains the `Flower` object that belongs to a certain instance.

Now for the real thing: the ultimate stock management system! First of all we map all database tables to Ruby classes and add bits of business logic where necessary:

File

```

Line 1  class Bouquet < ActiveRecord::Base
-      has_and_belongs_to_many :flowers
-
-      def price
5         flowers.inject(self.base_price) do |total, f|
-           total += f.price * f.quantity.to_i
-         end
-       end
-
10      def add(flower, quantity)
-        self.flowers.push_with_attributes(
-          flower,
-          :quantity => quantity
-        )

```



```

15     end
-   end
-
-   class Flower < ActiveRecord::Base
-     has_and_belongs_to_many :bouquets
20     belongs_to :stock_item
-   end
-
-   class StockItem < ActiveRecord::Base
-     has_one :flower
25   end

```

No big surprises here. The only new method is `add(flower,quantity)` method in `Bouquet`, and it was added only for convenience. More interesting is the `Stock` class:

File

```

Line 1  class Stock
-      def add_flower(name, price, quantity)
-        si = StockItem.new(:quantity => quantity)
-        si.save
5        si.create_flower(
-          :name => name,
-          :price => price
-        )
-      end
10
-      def remove(bouquet)
-        Bouquet.transaction do
-          bouquet.flowers.each do |f|
-            si = StockItem.find(f.stock_item_id)
15            if si.quantity < f.quantity.to_i
-              raise 'Not enough flowers left!'
-            end
-            si.quantity -= f.quantity.to_i
-            si.save
20          end
-        end
-      end
-
-      def print_report
25        StockItem.find(:all).each do |si|
-          puts "#{si.flower.name}: #{si.quantity} in stock."
-        end
-      end
-    end

```

It has everything we'd normally expect a `Stock` class to have: we can add a certain number of flowers having a certain price by calling `add_flower(name,price,quantity)`. The `remove(bouquet)` method will remove all flowers belonging to a bouquet from stock and the `print_report()` method will print something nice

for the clerks.

In line 12 we used another valuable ActiveRecord feature: transactions. When removing a bouquet from the database, we want to remove it completely—all the flower go, or none go. The `transaction()` method executes a code block and performs a rollback if an exception was raised in the block.

Let's fill our stock and print a first report:

File

```
Line 1  stock = Stock.new
-      [
-        ['rose', 1.10, 1000],
-        ['violet', 0.40, 2000],
5      ['sunflower', 0.40, 6]
-    ].each do |name, price, quantity|
-      stock.add_flower(name, price, quantity)
-    end
-    stock.print_report
```

produces:

```
rose: 1000 in stock.
violet: 2000 in stock.
sunflower: 500 in stock.
```

Then we create a new bouquet:

File

```
Line 1  bouquet = Bouquet.new(
-      :name => 'Honeybunch',
-      :base_price => 1.37
-    )
5  bouquet.save
-  bouquet.add(Flower.find_by_name('rose'), 1)
-  bouquet.add(Flower.find_by_name('sunflower'), 6)
-  puts "#{bouquet.name}' costs ${bouquet.price}."
```

and print its price:

```
'Honeybunch' costs $4.87.
```

Finally, we remove all flowers belonging to the bouquet from stock:

File

```
Line 1  stock.remove(bouquet)
-      stock.print_report
```

and print another report:

```
rose: 999 in stock.
violet: 2000 in stock.
sunflower: 494 in stock.
```

It's almost unbelievable: the only SQL statements we had to write were the table definitions (if you want to avoid writing table definitions, too,

have a look at Section 2.3, *Migrating Database Schemas*, on page 47). ActiveRecord let us implement PragBouquet's new stock management system in less than 80 lines of Ruby code that reads like a specification.

## Validation

As we all know, the most difficult things in software development are usually more or less unrelated to the actual problem being solved. The biggest problems are caused by error conditions: users enter invalid data, networks break down, hard disks crash, and so on.

This is also true in database programming. The most important technique for preventing errors in relational databases is to constrain columns so that they only contain valid values. As a special case of this, we need to maintain referential integrity—to make sure that foreign keys always refer to existing table rows in the other table.

All relational database systems allow you to put constraints on columns and tables somehow, but there are huge differences between their capabilities (many versions of MySQL, for example, allow you to define foreign key constraints, but they do not actually check them). To overcome these incompatibilities, ActiveRecord lets you define validation for your database objects in Ruby code.

For our Flower and Bouquet object the rules are simple:

```
File
Line 1  class Bouquet < ActiveRecord::Base
-       has_and_belongs_to_many :flowers
-
-       def validate
5         errors.add_on_empty %(name base_price)
-         unless base_price >= 0.0
-           errors.add('base_price', 'is negative')
-         end
-         errors.add('flowers', 'must exist') unless flowers.size > 0
10      end
-    end
-
-    class Flower < ActiveRecord::Base
-      has_and_belongs_to_many :bouquets
15
-      def validate
-        errors.add_on_empty %(name price)
-        errors.add(:price, 'is negative') unless price >= 0.0
-      end
20    end
```

### Where to Put Constraints?

Putting constraints on columns and on the relationships between tables is a two-edged sword. On the one hand, you have constraints that apply globally and can only be checked by the database system itself. It does not make sense—and often is impossible—to implement primary key, foreign key, or unique constraints in your application.

On the other hand you have things such as column value constraints like “prices should not be negative.” To assert that a numerical column value always is in a certain range can be easily achieved in both the database and the application. ActiveRecord supports this with its validation mechanism.

When you know for sure that a database will be used exclusively by your application, you may put the constraints on your data into the database. In all other cases try to put as much of your business logic into your application, because it can be found more easily (you don’t have to look into the database *and* the application) and you won’t affect other (future) applications that might have different or additional constraints on some columns.

No matter which way you choose, don’t forget the DRY principle—Don’t Repeat Yourself.\* Define every constraint just once, either in the application or in the database.

\*<http://www.artima.com/intv/dry.html>

Every ActiveRecord object has a member called `errors` of type `ActiveRecord::Errors`. Before writing an object to the database using `save()`, ActiveRecord calls the `validate()` method of the object. After `validate()` returns, ActiveRecord checks to see if any error has been written to the `errors` object. If any error occurred, the database will not be updated, and `save()` returns `false` (you can also use the `save!` variant that will raise an exception if an error occurred).

The rules for our domain objects are simple: the name and base price of a bouquet may not be empty, the base price has to be non-negative, and every bouquet must have at least one flower. Flowers must have a name and a price, too, and their price has to be greater than or equal to zero. The validation for this is shown in the following snippet:

File

```

Line 1  print 'Trying to save nameless flower: '
-      puts Flower.new(:name => nil, :price => 0).save
-      print 'Trying to save really cheap flower: '
-      puts Flower.new(:name => 'cheap', :price => -1).save
5      print 'Trying to save a valid flower: '
-      puts Flower.new(:name => 'daisy', :price => 0.02).save

```

produces:

```

Trying to save nameless flower: false
Trying to save really cheap flower: false
Trying to save a valid flower: true

```

Now we can prevent ourselves from storing imaginary bouquets:

File

```

Line 1  rose = Flower.find_by_name('rose')
-      sunflower = Flower.find_by_name('sunflower')
-      bouquet = Bouquet.new
-      bouquet.name = 'Honeybunch'
5      bouquet.base_price = 1.37
-      print 'Trying to save bouquet without flowers: '
-      puts bouquet.save
-      bouquet.flowers.push_with_attributes(rose, :quantity => 1)
-      bouquet.flowers.push_with_attributes(sunflower, :quantity => 6)
10     print 'Trying to save valid bouquet: '
-      puts bouquet.save

```

This produces:

```

Trying to save bouquet without flowers: false
Trying to save valid bouquet: true

```

## Migrating Database Schemas

One of the biggest problems with relational databases is maintaining their schemas and content. For example, in the development phase you often have to rename, drop, or add a column to a table. On your local machine this is not a big deal—you can mess with your private database installation whatever you want. However, when you check your code in, your fellow developers might be surprised to see that a new column is used in your Ruby code that is not present in their database.

ActiveRecord addresses this problem in a pragmatic way. The Migration class<sup>14</sup> defines methods for manipulating database schemas without writing a single DDL (Data Definition Language) statement. You can create and drop tables and indexes, and you can add, rename, or drop

<sup>14</sup><http://api.rubyonrails.com/classes/ActiveRecord/Migration.html>

columns, without knowing anything about the nitty-gritty details of the CREATE TABLE and ALTER TABLE statements of the database system you're using. Migration handles all this for you. This has significant advantages:

- All changes to your database schema are documented in Ruby code.
- The application can be easily migrated to every database system that is supported by Migration (at the time of this writing it works on MySQL, PostgreSQL, Microsoft SQL Server, and Oracle. Support for DB2 is in the works.).

To make the automatic flower ordering process even more intelligent, let's add a new column containing the average lifetime of a flower to the flowers table. When we have a lot of roses in stock and know that they only survive a few days in the cold store, it could make sense to make bouquets containing roses a bit cheaper in the shop.

Let's call our new column `life_time`. Using migrations, we'll add it like this:

```
File
Line 1  class AddLifeTime < ActiveRecord::Migration
-       def self.up
-         add_column :flowers, :life_time, :integer, :default => 5
-       end
5
-       def self.down
-         remove_column :flowers, :life_time
-       end
-     end
```

Every migration step is encapsulated in a class (AddLifeTime in our case) that is derived from ActiveRecord::Migration. The migration class has to implement the `up()` and `down()` methods for upgrading and downgrading the database schema. In these methods you can use methods such as `create_table()`, `drop_table()`, `add_column()`, `rename_column()`, `remove_column()`, `add_index()`, and `remove_index()` to modify your schema any way you'd like. In addition, you can call `execute(sql)` to run arbitrary SQL statements.

Let's test our migration class:

```
File
Line 1  AddLifeTime.up
-       Flower.reset_column_information
-
-       [
5       ['rose', 1.10, 4],
```

```

-     ['sunflower', 0.40, 10],
-     ['lily', 0.80, 5]
- ].each do |name, price, life_time|
-     flower = Flower.new(
10       :name => name,
-       :price => price,
-       :life_time => life_time
-     )
-     flower.save
15 end
-
- Flower.find(:all).each do |f|
-     puts "A #{f.name} costs $#{f.price} and lives " +
-         "for #{f.life_time} days."
20 end

```

produces:

```

A rose costs $1.1 and lives for 4 days.
A sunflower costs $0.4 and lives for 10 days.
A lily costs $0.8 and lives for 5 days.

```

After calling the `up()` method of class `AddLifeTime`, we can use the `life_time` attribute like any other attribute. The only important thing to remember is to call `reset_column_information()` (as we did in line 2) after changing a database schema. This will clear all cached column information. This is only necessary in this test case, as the program changes the schema while it is running.

Now the flowers table looks as follows:

```
mysql> describe flowers;
+-----+-----+-----+-----+-----+-----+
| Field      | Type                | Null | Key | Default | Extra          |
+-----+-----+-----+-----+-----+-----+
| id         | int(10) unsigned    |      | PRI | NULL    | auto_increment |
| name       | varchar(64)         |      |     |         |                |
| price      | double              |      |     | 0        |                |
| life_time  | int(11)              | YES  |     | 5        |                |
+-----+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)

```

As expected, Migration has added a new `life_time` column behind the scenes that we could remove just as easily by calling `AddLifeTime.down`.

The default lifetime value is 5 days. What if we wanted to set it to another value for existing rows right after we have added the new column? Simple answer: set it to whatever value you like in the `up()` method:

File

```
Line 1  class AddLifeTime < ActiveRecord::Migration
```

```

-   def self.up
-     add_column :flowers, :life_time, :integer, :default => 5
-     Flower.reset_column_information
5    Flower.find(:all).each do |f|
-      f.life_time = 3
-      f.save
-    end
-  end
10
-   def self.down
-     remove_column :flowers, :life_time
-   end
- end

```

Right after adding the new column, we call `reset_column_information()` in line 4 to activate it. Then, in line 5, we use the ordinary `find(*args)` method to iterate over all flowers and set their lifetime to 3 days.

To implement a real migration strategy, you'd need to write support scripts that work out which of your migrations to apply, but `ActiveRecord::Migration` will simplify this task significantly.<sup>15</sup>

## Conclusion

ActiveRecord can seem pretty magic, shielding you from the low-level SQL needed to perform CRUD operations and joins between tables. And, for much of the time, that's good enough. But ActiveRecord can't do everything. SQL is still a very important technology. The examples in this chapter were chosen to demonstrate ActiveRecord's strengths. If you cannot develop a complete database from scratch, and instead have to integrate with a legacy schema, things can get much more complicated. ActiveRecord relies upon a lot of conventions, and modeling complicated relationships between tables that have primary keys spanning several columns can be tricky to impossible.<sup>16</sup>

In any case you should study the ActiveRecord documentation carefully (or have a look at *Agile Web Development With Rails* [?]), because it has many useful features (such as table inheritance) that we did not cover here. Additionally, its capabilities differ from database system to database system so, for example, not all the features that are available with MySQL are also available if you use DB2.

---

<sup>15</sup>The Ruby On Rails project has a sophisticated solution for migrating database schemas. You should have a look at it.

<sup>16</sup>Read what the pros have to say: <http://blogs.pragprog.com/cgi-bin/pragdave.cgi/Tech/Ruby/IsRailsRe>



## 2.4 Lightweight Directory Access Protocol (LDAP)

We use directories all the time: telephone books, lists of network accounts, address books, the domain name service (DNS), and so on. Typically, directories are organized hierarchically—as trees—and their entries are often read and rarely modified.

Implementing directories with relational database systems can be a bit complicated. Even though many database vendors added tools for hierarchical queries to their products, they are still far from being convenient (some vendors, including Oracle, even ship a separate directory service that is based on their relational database product).

Hence, a standard for accessing directories was created as part of the *X.500 directory specification*. It was called *Directory Access Protocol* (DAP). Unfortunately, it was both complex and complicated, and no one implemented it completely.

*X.500 directory  
specification*

*Directory Access Protocol*

As a consequence, an easier standard was defined: the Lightweight Directory Access Protocol (LDAP)<sup>17</sup>, the most widespread directory service in use today.

Simply put, LDAP is for directories what SQL is for relational databases. It helps you to model real-world entities as so-called *object classes* (not as tables) that have different attributes. These can inherit additional attributes from other classes. Hence, an LDAP repository is a hierarchical tree of objects belonging to several object classes. In addition, LDAP defines a query language that allows you to read, modify, and delete subtrees and single nodes of your directories.

*object classes*

But the LDAP standard defines more than the protocol that is spoken between LDAP servers and their clients. It also specifies dozens of base classes for all the things you typically find in directories: *person*, *residentialPerson*, *organizationalPerson*, and so on.

It's not difficult to build your own object classes (it's similar to creating tables in SQL), but it's certainly a good idea first to check if the object class you need has not already be defined. Often it's sufficient to derive a new class from an existing one adding just a few attributes. For example, if you need to store address data containing the geographical position of the address, you can derive a new *geoPerson* class from *residentialPerson*, adding longitude and latitude attributes.

---

<sup>17</sup><http://www.faqs.org/rfcs/rfc2251.html>

Although a lot of directory services work more or less invisibly, touched only by your system administrators, chances are good that you'll have to integrate with one someday, because LDAP is gaining popularity among application developers, too. In the following sections we'll show how to manipulate a directory service based on OpenLDAP with Ruby.

## An Address Book For PragBouquet Customers

The marketing department made another astonishing observation: there are people who celebrate their birthdays every year! Wouldn't it be great if PragBouquet customers could easily send them a bouquet on those birthdays? And wouldn't it be nice if PragBouquet customers could be spared the extra work of entering the same address data for the recipients, over and over again?

So, marketing came up with an ingenious idea. All PragBouquet customers should have their own address book where they can store the addresses of the people they've ever sent a bunch of flowers.

The web shop team said that it's not a big deal to create an user interface for the address book, but they asked you to create the corresponding backend services. Fortunately, they want to give Ruby on Rails<sup>18</sup> a try, so you can use Ruby for implementing the address book logic.

When thinking about things like address books, LDAP immediately comes to mind, so you decide to implement the address book as a directory service using the the OpenLDAP<sup>19</sup> system. It has everything you need, it's available for free, it works on top of several database systems, and it ships with several utilities for reading and manipulating data.

For the development phase we install an OpenLDAP server on our local machine and configure it using this configuration file:

File

```
Line 1  include /sw/etc/openldap/schema/core.schema
-
-  database bdb
-  suffix "dc=pragbouquet,dc=com"
5  rootdn "cn=root,dc=pragbouquet,dc=com"
-  rootpw secret
-  directory /sw/var/openldap-data
-  index objectclass eq
```

<sup>18</sup><http://www.rubyonrails.com>

<sup>19</sup><http://www.openldap.org>

That is really all we need to get our address book application up and running. We have to include the core schema, because we'll need some of its definitions (person, residentialPerson, and uidObject). In addition, we have to define the database we want to use (the LDAP standard does not define how the directory is to be stored). It's a Berkeley database (bdb)<sup>20</sup> with all data files stored in directory /sw/var/openldap-data. The distinguished name of our root node (needed for administrative purposes only) is cn=root,dc=pragbouquet,dc=com. We have to authenticate ourselves using the nearly unbreakable plain text password "secret" whenever we want to write to the database.

LDAP allows us to create a sophisticated directory layout for our address books comprising lots of organizational units or even define our own object classes, but we will use a more modern and simpler approach. We will organize our directory in a flat way using domain components and uid attributes.<sup>21</sup>

Before diving into Ruby code, let's have a closer look at the directory structure and then initialize our repository with some sample data stored in init.ldif:

File

```

Line 1  # Create the PragBouquet organization.
-   dn: dc=pragbouquet,dc=com
-   objectclass: dcObject
-   objectclass: organization
5   o: PragBouquet
-   dc: pragbouquet
-
-   # Create an address book for customer 4711.
-   dn:uid=4711,dc=pragbouquet,dc=com
10  objectclass: top
-   objectclass: person
-   objectclass: uidObject
-   uid: 4711
-   cn: John Jackson
-   sn: Jackson
15  description: Address book of John Jackson.
-
-   # Create the first address book entry for customer 4711.
-   dn:cn=Marge Jackson,uid=4711,dc=pragbouquet,dc=com
20  objectclass: top
-   objectclass: residentialPerson
-   cn: Marge Jackson
-   sn: Jackson

```

<sup>20</sup><http://sleepycat.com/>

<sup>21</sup><http://www.faqs.org/rfcs/rfc2377.html>

```

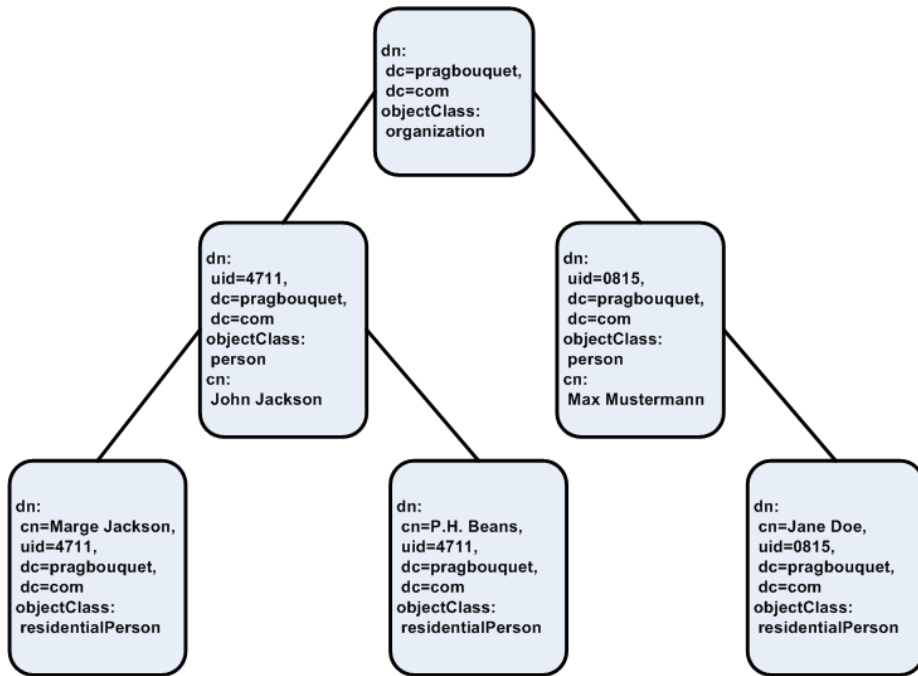
-   l: Springfield
25  st: IL
-   street: Evergreen Terrace 42
-   postalCode: 62701
-   description: Don't forget our wedding anniversary!
-
30  # Create the second address book entry for customer 4711.
-   dn:cn=P.H. Beans,uid=4711,dc=pragbouquet,dc=com
-   objectclass: top
-   objectclass: residentialPerson
-   cn: P.H. Beans
35  sn: Beans
-   l: Springfield
-   st: MO
-   street: Nuclear Powerplant Road 1
-   postalCode: 65801
40  description: My boss.
-
-   # Create an address book for customer 0815.
-   dn:uid=0815,dc=pragbouquet,dc=com
-   objectclass: top
45  objectclass: person
-   objectclass: uidObject
-   uid: 0815
-   cn: Max Mustermann
-   sn: Mustermann
50  description: Address book of Max Mustermann.
-
-   # Create the first address book entry for customer 0815.
-   dn:cn=Jane Doe,uid=0815,dc=pragbouquet,dc=com
-   objectclass: top
55  objectclass: residentialPerson
-   cn: Jane Doe
-   sn: Doe
-   street: 125 N. Arbitrary Street
-   st: DC
60  l: Washington
-   postalCode: 20500
-   description: My Sweetheart!

```

The LDIF file above should be nearly self-explanatory (comment lines start with a '#' character). Every entry has a so-called *distinguished name* (DN). All its other attributes are listed as “key: value” pairs. All attributes are potentially multi-dimensional, so they may appear several times.

*distinguished name*

Note that we use the attribute `uid` to structure our address books. Every web shop user is identified by a particular identifier (it might be a customer id, an e-mail address, or something similar). Whenever a customer creates a completely new address book (not an address book




---

Figure 2.3: Address Book Layout

---

entry) a new directory entry for her user id will be added. The directory belonging to our `init.ldif` file looks like Figure 2.3 (we have left out most attributes for brevity).

Let's start our server and load the initial data using the `ldapadd` command:

```
mschmidt:~/ldap> sudo slapd
Password:
mschmidt:~/ldap> ldapadd -c -x -D "cn=root,dc=pragbouquet,dc=com" \
> -W -f init.ldif
Enter LDAP Password:
adding new entry "dc=pragbouquet,dc=com"

adding new entry "uid=4711,dc=pragbouquet,dc=com"

adding new entry "cn=Marge Jackson,uid=4711,dc=pragbouquet,dc=com"

adding new entry "cn=P.H. Beans,uid=4711,dc=pragbouquet,dc=com"

adding new entry "uid=0815,dc=pragbouquet,dc=com"
```

adding new entry "cn=Jane Doe,uid=0815,dc=pragbouquet,dc=com"

Our .ldif file didn't contain any errors, and six new entries have been created.

OpenLDAP's `ldapsearch` command allows us to query the repository. It prints its results in LDIF. To become a bit more familiar with our directory, let's print the address book of the user identified by uid 4711:

```
mschmidt:~/ldap> ldapsearch -x -s one
> -b 'uid=4711,dc=pragbouquet,dc=com'
> '(objectclass=*)'
# extended LDIF
#
# LDAPv3
# base <uid=4711,dc=pragbouquet,dc=com> with scope one
# filter: (objectclass=*)
# requesting: ALL
#
# Marge Jackson, 4711, pragbouquet.com
dn: cn=Marge Jackson,uid=4711,dc=pragbouquet,dc=com
objectClass: top
objectClass: residentialPerson
cn: Marge Jackson
sn: Jackson
l: Springfield
st: IL
street: Evergreen Terrace 42
postalCode: 62701
description: Don't forget our wedding anniversary!

# P.H. Beans, 4711, pragbouquet.com
dn: cn=P.H. Beans,uid=4711,dc=pragbouquet,dc=com
objectClass: top
objectClass: residentialPerson
cn: P.H. Beans
sn: Beans
l: Springfield
st: MO
street: Nuclear Powerplant Road 1
postalCode: 65801
description: My boss.

# search result
search: 2
result: 0 Success

# numResponses: 3
# numEntries: 2
```

Obviously, everything is up and running. Our query returned the two address book entries that belong to the customer identified by user id 4711. But what are those options we passed to the command?

- `-x` uses the simple authentication mechanism. In our case the communication is unencrypted and no password is needed.
- `-s one` will search the directory “one level beyond base”, so it returns all entries below our search base, but not the base itself. `-s base` would have returned the base object only and `-s sub` would have returned the base object and all its descendants.
- `-b 'uid=4711,dc=pragbouquet,dc=com'` sets the search base to the distinguished name `uid=4711,dc=pragbouquet,dc=com`, so all entries of the subtree belonging to this DN will be returned.
- `(objectclass=*)` specifies a filter for the entries to be returned. The `(objectclass=*)` filter is comparable to SQL’s `SELECT *` statement and selects all entries not matter what attributes they have. If we were interested in entries from Illinois only, we could have set the filter to `(st=IL)`.

In the following sections we’ll see how to manipulate our repository with Ruby.

## Ruby/LDAP

The Ruby/LDAP<sup>22</sup> library was initially created by Takaaki Tateishi, and is currently maintained by Ian Macdonald. It supports all LDAP clients that comply with the LDAP Application Program Interface.<sup>23</sup> So, you can use it with OpenLDAP, Netscape, and ActiveDirectory, e.g.

As a first exercise we’ll try to read John Jackson’s address book. It should not be too surprising that accessing a directory service looks similar to accessing a relational database system:

File

```
Line 1  require 'pp'
-       require 'ldap'
-       include LDAP
-
5       begin
-         connection = Conn.new
-         connection.set_option(LDAP_OPT_PROTOCOL_VERSION, 3)
-         connection.bind do
```

<sup>22</sup><http://ruby-ldap.sourceforge.net>

<sup>23</sup><http://www.faqs.org/rfcs/rfc1823.html>

```

-     base_dn = 'uid=4711,dc=pragbouquet,dc=com'
10    scope = LDAP_SCOPE_ONELEVEL
-     filter = '(objectClass=*)'
-     connection.search(base_dn, scope, filter) do |entry|
-       pp entry.to_hash
-     end
15  end
-  rescue Exception => ex
-    puts ex
-  end

```

prints:

```

{"cn"=>[ "Marge Jackson"],
 "st"=>[ "IL"],
 "l"=>[ "Springfield"],
 "sn"=>[ "Jackson"],
 "description"=>[ "Don't forget our wedding anniversary!"],
 "postalCode"=>[ "62701"],
 "street"=>[ "Evergreen Terrace 42"],
 "objectClass"=>[ "top", "residentialPerson"],
 "dn"=>[ "cn=Marge Jackson,uid=4711,dc=pragbouquet,dc=com"]}
{"cn"=>[ "P.H. Beans"],
 "st"=>[ "MO"],
 "l"=>[ "Springfield"],
 "sn"=>[ "Beans"],
 "description"=>[ "My boss."],
 "postalCode"=>[ "65801"],
 "street"=>[ "Nuclear Powerplant Road 1"],
 "objectClass"=>[ "top", "residentialPerson"],
 "dn"=>[ "cn=P.H. Beans,uid=4711,dc=pragbouquet,dc=com"]}

```

First, we create a new connection to the LDAP service by calling `LDAP::Conn.new(host='localhost',port:389)`. We then set the `LDAP_OPT_PROTOCOL_VERSION` option, because we have set up a LDAPv3 service (it's OpenLDAP's default).

In line 8 we bind our connection object to the server. The real work is performed in the code block we pass to the `bind(dn=nil,password=nil,method=LDAP_AUTH_SIMPLE)` method. The heart of our “program logic” is the `search()` method. It expects the following parameters:

1. `base_dn` contains the base DN of the subtree to search in.
2. `scope` defines the search scope. It can be set to `LDAP_SCOPE_ONELEVEL`, `LDAP_SCOPE_SUBTREE`, or `LDAP_SCOPE_BASE`.

In our example we have used `LDAP_SCOPE_ONELEVEL`, which means “one level beyond base.” We are not interested in the base object (the address book owner) itself. If we had set the scope to `LDAP_SCOPE_SUBTREE`



the program would have printed the entry for the address book owner, too:

```
{ "cn"=>[ "John Jackson" ],
  "sn"=>[ "Jackson" ],
  "uid"=>[ "4711" ],
  "description"=>[ "Address book of John Jackson." ],
  "objectClass"=>[ "top", "person", "uidObject" ],
  "dn"=>[ "uid=4711,dc=pragbouquet,dc=com" ] }
...

```

LDAP\_SCOPE\_BASE only returns the base object (the address book owner in our case).

3. filter contains the LDAP search filter to be used.
4. The attributes array contains the name of the attributes to be returned. If it is empty or nil (the default), all attributes are returned.
5. The attributes\_only flag indicates whether only the names of the attributes should be returned (true). When it is set to false (the default), it returns both names and values.
6. seconds specifies the seconds portion of the search timeout. It defaults to 0. If this parameter or the useconds parameter is greater than 0, the timeout mechanism will be activated.
7. useconds specifies the microseconds portion of the search timeout. It defaults to 0. If this parameter or the seconds parameter is greater than 0, the timeout mechanism will be activated. To set a timeout of 2.5 seconds, set seconds to 2 and useconds to 500.
8. sort\_attribute specifies the attribute to sort the search result entries by. If no sort attribute is specified (the default), the order of the result entries is unpredictable.
9. sort\_proc may contain a code block that is used for sorting the entries returned by the server. It defaults to nil, so the order of the result entries is unpredictable.

search() is an iterator. It expects a code block that gets passed the current entry as LDAP::Entry object. In line 13 we turn these objects into hashes and print them, nicely formatted.

Reading LDAP entries seems to be fairly easy. Let's try to create new ones, now. First, let's add an empty address book for Jane Doe (she is already a member of Max Mustermann's address book, but that doesn't matter, because for us they are two different customers):

File

```

Line 1  User = Struct.new(:uid, :forename, :surname)
-      class AddressBook
-        BASE_DC = 'dc=pragbouquet,dc=com'
-
-      attr_reader :user
-
-      def initialize(connection, user)
-        @connection, @user = connection, user
-      end
10
-      def AddressBook.create(connection, user)
-        cn = user.forename + ' ' + user.surname
-        adr_book = []
-        [
15          ['objectclass', %w(top person uidObject)],
-          ['uid', [user.uid]],
-          ['cn', [cn]],
-          ['sn', [user.surname]],
-          ['description', ['Address book of ' + cn]]
20        ].each do |attr, values|
-          adr_book << LDAP.mod(LDAP_MOD_ADD, attr, values)
-        end
-
-        connection.add(
25          'uid=' + user.uid + ',' + BASE_DC,
-          adr_book
-        )
-        AddressBook.new(connection, user)
-      end
30  end

```

On line 1 we declare a User class that contains all the attributes we need to manage an address book for a web shop user: a user id (typically something like a customer id or an e-mail address), a first name and a surname.

New address books can be created with the create(connection,user) method. Objects returned by initialize(connection,user) refer to address books that have to exist already. This time we did not make our connection object available as a singleton. If we decide to do so later, we can easily change our class.

The interesting stuff happens in the create(connection,user) method. Here we iterate over an array of two-dimensional arrays. Because LDAP attributes can often appear multiple times, each entry contains the name of an attribute to be added and an array of values.

LDAP::Mod objects represent modifications to a single attribute. These modifications can be of type LDAP\_MOD\_ADD, LDAP\_MOD\_REPLACE, or

LDAP\_MOD\_DELETE (some LDAP servers define extension types).<sup>24</sup> In line 21 we create LDAP::Mod objects for adding attributes with the LDAP::mod(mod\_type,attr,values) method and append them to the `odr_book` array. The result is an array that contains “add modifications” for all the attributes we’re storing for a new address book.

Finally, in line 24, we send our modifications list to the server and can create a new AddressBook object:

File

```
Line 1  connection = Conn.new
-      connection.set_option(LDAP_OPT_PROTOCOL_VERSION, 3)
-      connection.bind('cn=root,dc=pragbouquet,dc=com', 'secret')
-
5      user = User.new('23', 'Jane', 'Doe')
-      address_book = AddressBook.create(connection, user)
```

We had to use a variant of the `bind()` method in line 3. Because we want to use our connection for writing, we have to authenticate ourselves, passing the root DN and the corresponding password.

Now that we’ve seen how to add entries to a LDAP database in principle, let’s define a method for adding bouquet recipients to an existing address book.

File

```
Line 1  Recipient = Struct.new(
-      :forename, :surname, :street, :postal_code,
-      :city, :state, :description
-    )
5
-  class AddressBook
-    def udn
-      'uid=' + @user.uid + ',' + BASE_DC
-    end
10
-    def add(recipient)
-      cn = recipient.forename + ' ' + recipient.surname
-      entry = {
-        'objectclass' => %w(top residentialPerson),
15      'cn' => [cn],
-      'sn' => [recipient.surname],
-      'l' => [recipient.city],
-      'street' => [recipient.street],
-      'postalCode' => [recipient.postal_code],
20      'st' => [recipient.state || ''],
-      'description' => [recipient.description || '']
```

---

<sup>24</sup>If you want to add binary values you have to logically or the modification type with LDAP\_MOD\_BVALUES, so to add a binary value you’d use LDAP\_MOD\_ADD | LDAP\_MOD\_BVALUES.

```

-     }
-     @connection.add('cn=' + cn + ',' + udn(), entry)
-   end
25 end

```

There is not much to say here. We declared a `Recipient` class that represents the happy recipients of a bouquet, and we defined an `add(recipient)` method that adds a `Recipient` object to the address book of a particular user. This time we did not create an array of `LDAP::Mod` objects. Instead we used a hash containing all the attributes (and their corresponding values) for the new recipient. `LDAP::Conn.add(dn, entry)` can handle a hash object, too. For convenience we defined the `udn()` method that returns the distinguished name for a directory entry belonging to a particular user id.

So, let's add a new entry to Jane Doe's address book.

File

```

Line 1  user = User.new('23', 'Jane', 'Doe')
-       address_book = AddressBook.new(connection, user)
-       recipient = Recipient.new(
-         'Jose', 'Rodriguez',
5       'Casanova Street 6', '77002',
-       'Houston', 'TX',
-       'Rrrrrrrr!'
-     )
-     address_book.add(recipient)

```

Hmmm, did it work? We'd better implement a method to print the whole address book:

File

```

Line 1  class AddressBook
-       def each
-         @connection.search(
-           udn,
5         LDAP_SCOPE_ONELEVEL,
-         '(objectClass=residentialPerson)',
-         nil, false, 0, 0,
-         'sn'
-       ) do |recipient|
10        yield recipient
-       end
-     end
-
-     def each_recipient
15     each do |entry|
-       rec_data = entry.to_hash
-       sn = rec_data['sn'][0]
-       cn = rec_data['cn'][0]
-       cn.sub!(Regexp.new(' ' + sn + '$'), '')
20     yield Recipient.new(

```

```

-         cn,
-         sn,
-         rec_data['street'][0],
-         rec_data['postalCode'][0],
25      rec_data['l'][0],
-         rec_data['st'][0],
-         rec_data['description'][0]
-     )
-     end
30 end
- end

```

Yeah, that's Ruby code! Starting on line 2, we define an iterator that passes every recipient to a code block. In line 6 we define an object filter for the first time—we are only interested in objects of class `residentialPerson`—and in line 8 we set the sort attribute to `sn`, so the results are sorted by the recipients surname attribute (in the preceding line we had to explicitly pass the default values for all remaining parameters. Hopefully, named parameters will be introduced in Ruby 2.0...).

The `each_recipient()` method is a piece of cake. In line 16 we turn each `LDAP::Entry` object into a hash and use its entries to initialize a new `Recipient` object. Line 19 is a bit more interesting: here we extract the recipient's first name from the entries' common name.

Now let's print the address book of Jane Doe:

File

```

Line 1  user = User.new('23', 'Jane', 'Doe')
-      address_book = AddressBook.new(connection, user)
-      address_book.each_recipient { |r| pp r}

```

This produces:

```

#<struct Recipient
  forename="Jose",
  surname="Rodriguez",
  street="Casanova Street 6",
  postal_code="77002",
  city="Houston",
  state="TX",
  description="Rrrrrrr!">

```

That's correct, and Jose Rodriguez's entry was the one we expected. But the formatting needs some adjustments...

File

```

Line 1  require 'ldap/ldif'
-      class AddressBook
-        include Enumerable
-
-      def to_ldif
-        inject("") { |ldif,e| ldif << e.to_ldif }

```

### Why Isn't AddressBook#each Private?

Back in the dark and ancient days of object-oriented programming, a lot of people were obsessed with declaring everything private whenever possible (not when it made sense!).

Of course, purists may say that the `each()` method of the `AddressBook` class should be declared private, because it passes `LDAP::Entry` objects to its code block and reveals some of its innards. What if you change your address book backend? What if you're using a relational database instead of a LDAP repository?

They are right—in a way—but at the moment it's more important to create an API that's easy to use and—more importantly—easy to test. It's more difficult to test a private `each()` method. The users of your `AddressBook` class won't care about method visibility, but they will care about bugs that could have been prevented by simple unit tests...

As a rule of thumb, you should only care about the visibility of your methods when programming libraries that will be used by a large number of programmers. When writing code for applications, it's usually more important to think about things like testability.

```
-   end
-   end
```

That's a perfect example for idiomatic Ruby: because we have defined an `each()` method, we can include the `Enumerable` module. Because we called `include Enumerable`, we can use the `inject(initial)` method to create an LDIF representation of the whole address book with a single line of code (note that we had to require `'ldap/ldif'`, too). So, let's do it:

File

```
Line 1  user = User.new('23', 'Jane', 'Doe')
-       address_book = AddressBook.new(connection, user)
-       puts address_book.to_ldif
```

Here's the corresponding LDIF output:

```
dn: cn=Jose Rodriguez,uid=23,dc=pragbouquet,dc=com
cn: Jose Rodriguez
st: TX
l: Houston
objectClass: top
```

```
objectClass: residentialPerson
sn: Rodriguez
description: Rrrrrrr!
postalCode: 77002
street: Casanova Street 6
```

That looks like something that could be imported by any modern address book application, doesn't it?

Jane just realized that Jose did not get the last present she sent (the biggest bouquet that has ever been produced by PragBouquet) because Jane got the wrong house number: it's 8 not 6. We need to change the entry:

File

```
Line 1  class AddressBook
-      def modify(recipient)
-          cn = recipient.forename + ' ' + recipient.surname
-          entry = {
5          'l' => [recipient.city],
-          'street' => [recipient.street],
-          'postalCode' => [recipient.postal_code],
-          'st' => [recipient.state],
-          'description' => [recipient.description]
10         }
-         @connection.modify('cn=' + cn + ',' + udn(), entry)
-      end
-  end
```

LDAP::Conn.modify(dn,attributes) modifies the object identified by the distinguished name dn, setting its attributes to the content of the attributes parameter. This parameter can be a hash or an array of LDAP::Mod objects. The following snippet corrects Jose's address:

File

```
Line 1  user = User.new('23', 'Jane', 'Doe')
-      address_book = AddressBook.new(connection, user)
-      recipient = Recipient.new(
-          'Jose', 'Rodriguez',
5      'Casanova Street 8', '77002',
-      'Houston', 'TX',
-      'Rrrrrrr!'
-      )
-      address_book.modify(recipient)
```

Although Jose definitely got her present this time, he was not as enthusiastic as Jane would have liked, so she decides to remove him from her address book. Fortunately, we have defined an appropriate method already:

File

```
Line 1  class AddressBook
-      def remove(recipient)
```

```

-     cn = recipient.forename + ' ' + recipient.surname
-     @connection.delete('cn=' + cn + ',' + udn())
5   end
-   end

```

The only new thing is the `delete(dn)` call we used in line 4. Now removing Jose is easy:

File

```

Line 1  user = User.new('23', 'Jane', 'Doe')
-     address_book = AddressBook.new(connection, user)
-     recipient = Recipient.new(
-       'Jose', 'Rodriguez',
5       'Casanova Street 6', '77002',
-       'Houston', 'TX',
-       'Rrrrrrr!'
-     )
-     address_book.remove(recipient)

```

Two weeks later Jane met a guy called Ron, and now she doesn't need her PragBouquet address book any longer...

File

```

Line 1  class AddressBook
-     def delete() @connection.delete(udn); end
-   end

```

Good luck, Jane!

## ActiveLDAP

After reading the preceding sections about databases, ActiveRecord, and LDAP, you might be thinking that it would be a good idea to develop an ActiveRecord equivalent for LDAP. What would you call it? ActiveLDAP, perhaps?

I've got good news for you: Will Drewry had the same idea. He's released a small library called ActiveLDAP that maps LDAP repository structures to Ruby classes and vice versa.<sup>25</sup> Under the hood it is based on Ian Macdonald's Ruby/LDAP library.

Let's use ActiveLDAP to clean up our customer account data. Users of PragBouquet's web shop have to enter their e-mail address and a password to create an account. These two values are stored in an LDAP repository. Later, they can be used to log into the web shop, allowing them to place new orders or have a look at current orders.

People who have an account do not necessarily have to be customers. Unless someone actually ordered something from the shop, PragBou-

<sup>25</sup><http://ruby-activeldap.rubyforge.org>



quet will not ask for address data, and so on. As a result, there are nominal accounts that have been created but never have been used for shopping. The marketing department wants to know how many there are, and it wants the email addresses belonging to these accounts. (Maybe they can find out why these people never bought anything.)

PragBouquet runs an OpenLDAP server that manages the account data of all customers and of all employees. The server has been configured like this:

File

```

Line 1  include /sw/etc/openldap/schema/core.schema
-      include /sw/etc/openldap/schema/cosine.schema
-      include /sw/etc/openldap/schema/nis.schema
-
5      database bdb
-      suffix "dc=pragbouquet,dc=com"
-      rootdn "cn=root,dc=pragbouquet,dc=com"
-      rootpw secret
-      directory /sw/var/openldap-data
10     index objectclass eq

```

Because we store account data that depends on the posixGroup and posixAccount object classes we have to include the nis schema. This schema in turn depends on the cosine schema, so we include that, too.

Sample data in the repository looks like this:

File

```

Line 1  # Create the PragBouquet organization.
-      dn: dc=pragbouquet,dc=com
-      objectclass: dcObject
-      objectclass: organization
5      o: PragBouquet
-      dc: pragbouquet
-
-      # Create a container for all groups.
-      dn: ou=Groups,dc=pragbouquet,dc=com
10     objectclass: organizationalUnit
-      ou: Groups
-
-      # Create a group for employee accounts.
-      dn: cn=employees,ou=Groups,dc=pragbouquet,dc=com
15     objectclass: top
-      objectclass: posixGroup
-      cn: employees
-      gidNumber: 42
-
20     # Create a group for customer accounts.
-      dn: cn=customers,ou=Groups,dc=pragbouquet,dc=com
-      objectclass: top
-      objectclass: posixGroup

```

```

-   cn: customers
25  gidNumber: 23
-
-   # Create an account for employee Maik Schmidt
-   dn:uid=mschmidt,cn=employees,ou=Groups,dc=pragbouquet,dc=com
-   objectclass: top
30  objectclass: account
-   objectclass: posixAccount
-   cn: Maik Schmidt
-   uid: mschmidt
-   uidNumber: 1000
35  gidNumber: 42
-   userPassword: {SSHA}wFH8hVlIQKttNK2+334mh2K3PHBRv9Lt
-   homeDirectory: /home/mschmidt
-
-   # Create an account for employee Carl Coder
40  dn:uid=ccoder,cn=employees,ou=Groups,dc=pragbouquet,dc=com
-   objectclass: top
-   objectclass: account
-   objectclass: posixAccount
-   cn: Carl Coder
45  uid: ccoder
-   uidNumber: 1001
-   gidNumber: 42
-   userPassword: {SSHA}ytrSx9Sc8v3RmitArARTfPRdQRCGZR9
-   homeDirectory: /home/ccoder
50
-   # Create an account for Homer Simpson
-   dn:uid=homer@example.com,cn=customers,ou=Groups,dc=pragbouquet,dc=com
-   objectclass: top
-   objectclass: account
55  objectclass: posixAccount
-   cn: Homer Simpson
-   uid: homer@example.com
-   uidNumber: 2000
-   gidNumber: 23
60  userPassword: {SSHA}DFTpR8b5R+x+p5E9fj1NZwrQQLRgfeBn
-   homeDirectory: /tmp
-
-   # Create an account for Jane Doe
-   dn:uid=jane_doe@example.net,cn=customers,ou=Groups,dc=pragbouquet,dc=com
65  objectclass: top
-   objectclass: account
-   objectclass: posixAccount
-   cn: Jane Doe
-   uid: jane_doe@example.net
70  uidNumber: 2001
-   gidNumber: 23
-   userPassword: {SSHA}lc7hXzhDP9T8qS51TUSE/89oLfq4EWti
-   homeDirectory: /tmp

```

There are two groups called “customers” and “employees”. Each has two entries: Carl Coder and Maik Schmidt belong to the employees group, and Homer Simpson and Jane Doe are members of the customers group.

In Ruby, we’ll represent these groups using classes `Group` and `Customer`. Our hope is that (just as ActiveRecord does for database tables) ActiveLDAP will create them automatically for us. But before we can use ActiveLDAP we have to initialize it:

File

```
Line 1  require 'rubygems'
-       require 'activeldap'
-
-       ActiveLDAP::Base.connect(
5         :base => 'dc=pragbouquet,dc=com',
-         :bind_format => 'cn=root,dc=pragbouquet,dc=com',
-         :password_block => Proc.new { 'secret' },
-         :allow_anonymous => false
-       )
```

After loading the ActiveLDAP Gem, we call the `ActiveLDAP::Base.connect(options)` method passing it the following options:

- `base` defines the search base that will be appended to all distinguished name attributes.
- `bind_format` contains the distinguished name to be used for binding to the server. Usually, this is the `rootdn` defined in the OpenLDAP server configuration.
- `password_block` points to a code block that returns a password. Defaults to `nil`.
- If `allow_anonymous` is `true`, it is possible to bind to the server anonymously after all other bind methods fail. Otherwise it is not. Defaults to `true`.

The `connect()` method supports many more options, but we don’t need them here. If you don’t want to modify data in your LDAP repository and your server allows anonymous binding, you don’t have to call `connect()` at all—ActiveLDAP will do it for you automatically.

File

```
Line 1  class Group < ActiveLDAP::Base
-       ldap_mapping :dnattr => 'cn', :prefix => 'ou=Groups'
-       end
-
-
5       customers = Group.new('customers')
-       puts "The 'customers' group has the following attributes:"
-       customers.attributes.each { |a| puts "  #{a}" }
```

```

-
- puts "\nIts group id is #{customers.gidNumber}."
10
- puts "\nWe have the following groups:"
- Group.find_all('*').each { |g| puts "  #{g}" }

```

produces:

The 'customers' group has the following attributes:

```

gidNumber
cn
memberUid
commonName
description
userPassword
objectClass

```

Its group id is 23.

We have the following groups:

```

employees
customers

```

Although the example is short, there's a lot to be explained. In line 1 we derive our Group class from ActiveLDAP::Base. That would be sufficient if we were mapping a relational database using ActiveRecord, but for LDAP we have to call `ldap_mapping()` in line 2. It accepts the following parameters:

- `prefix` is the static (more or less) part of the distinguished name that will always be prepended to the base defined in the `connect()` call. It defaults to 'ou=class name' ('ou=Group' in our case).
- `dnattr` contains the variable part of the distinguished name of an object. It defaults to 'cn', so we could have left it out this time.
- `classes` contains an Array of all object classes whose attributes should be mapped to the ActiveLDAP object. Its default value is ['top'].

When we create our first ActiveLDAP object in line 5, it references the LDAP object with the DN `cn=customers,ou=Groups,dc=pragbouquet,dc=com`.

In line 7 we use the reflective features of ActiveLDAP for the first time and print a list of all attributes belonging to a Group object. ActiveLDAP gives you accessors for all these attributes for free.

Then, in line 12, we read some real data from our repository with the `find_all()` method. ActiveLDAP objects can use the following class meth-

ods to search for entries in a repository:

- `find(config={})` returns the `dnattr` of the first entry that matches a certain query. `config` can be a String or a Hash object. If it is a String, it will be matched against `dnattr`. For example, `Group.find('e*')` returns the common name attribute of the first group entry whose `cn` attribute starts with `e`. In our case, it returns `'employees'`.

If `config` is a Hash, it may contain the keys `:attribute`, `:value`, and `:objects`. The `:attribute` and `:value` parameters can be used to search for objects where a particular attribute has a particular value. To find the common name of the group having the group id 23 run the following statement:

```
Group.find(:attribute => 'gidNumber', :value => '23')
```

Usually, you are not only interested in the `dnattr` of an object but in the object itself. To get back complete objects, pass `:objects => true`:

```
Group.find(
  :attribute => 'gidNumber',
  :value => '23',
  :objects => true)
```

- `find_all(config={})` works like the `find()` method, but returns *all* the entries matching a certain query.
- `search(config={})` gives you more or less direct access to the `search()` method of the underlying Ruby/LDAP library. It implements the `:attrs`, `:base`, `:filter`, and `:scope` parameters, corresponding to the `attributes`, `base_dn`, `filter`, and `scope` parameters of the Ruby/LDAP `search()` method described on page 57. The return value has the same format, too:

```
Group.search(:filter => '(cn=cus*)')
```

returns:

```
[
  {
    "gidNumber" => ["23"],
    "cn" => ["customers"],
    "objectClass" => ["posixGroup"],
    "dn" => ["cn=customers,ou=Groups,dc=pragbouquet,dc=com"]
  }
]
```

The following code segment separates the customer account data from the employee account data:

File

```

Line 1  class Customer < ActiveLDAP::Base
-       ldap_mapping :dnattr => 'uid',
-               :prefix => 'cn=customers,ou=Groups'
-       end
5
-       puts "Our customers are:"
-       Customer.find_all('*').each { |c| puts "  #{c}" }
-       h = Customer.new('homer@example.com')
-       puts "\nCommon name attribute:"
10      p h.cn
-       p h.cn(true)

```

This produces:

```

Our customers are:
homer@example.com
jane_doe@example.net

Common name attribute:
["Homer Simpson"]
"Homer Simpson"

```

The definition of the `Customer` class does not differ much from the `Group` class—we just set the `dnattr` and `prefix` attributes differently. In lines 10 and 11, we print the common name attribute of a `Customer` object. Usually, all attribute values will be returned as an `Array`, as every LDAP attribute is potentially multidimensional. If you want to get back an attribute value as a `String` object, pass `true` to the accessor.

We have everything at hand now to solve our little problem. The following method expects an `Array` containing the e-mail addresses of all customers that have ever bought something in our web shop. It returns a list of email addresses of all people who have created an account, but never used it to send some flowers:

File

```

Line 1  def get_unused_accounts(used_accounts)
-       unused_accounts = []
-       Customer.find_all('*').each do |email|
-           unused_accounts << email if !used_accounts.include?(email)
5       end
-       unused_accounts
-       end
-       puts get_unused_accounts(['homer@example.com'])

```

Our original job is done, but since we have a test system up and running anyway, wouldn't it be fun to see what other interesting features `ActiveLDAP` has to offer?

Perhaps you want delete the unused accounts?

File

```

Line 1  def delete_unused_accounts(used_accounts)
-       Customer.find_all('*').each do |cn|
-         Customer.new(cn).delete if !used_accounts.include?(cn)
-       end
5       end

```

Or perhaps you want to tag them with a special note in their description attribute?

File

```

Line 1  def mark_unused_accounts(used_accounts)
-       Customer.find_all('*').each do |cn|
-         if !used_accounts.include?(cn)
-           c = Customer.new(cn)
5         c.description = 'unused'
-         c.write
-       end
-     end
-   end

```

ActiveLDAP gives us all the CRUD methods for free. This feature alone makes it a valuable tool. But it gives us even more, allowing us to define relationships between objects as if we were working with a relational database:

File

```

Line 1  class Customer < ActiveLDAP::Base
-       ldap_mapping :dnattr => 'uid',
-                   :prefix => 'cn=customers,ou=Groups'
-
5       belongs_to :groups,
-               :class_name => 'Group',
-               :foreign_key => 'gidNumber',
-               :local_key => 'gidNumber'
-     end
10
-     h = Customer.new('homer@example.com')
-     h.groups.each { |g| puts g.cn }

```

This prints:

```
customers
```

Great, isn't it? In line 5 we declare that every customer belongs to a group, with the relationship defined by the `gidNumber` attribute in both the `Group` and `Customer` class. The first parameter of the `belongs_to()` method is the name of the new member variable to be created in the `Customer` class. The second parameter is a Hash that contains up to three parameters: `:class_name` points to the class your current class belongs to. `:foreign_key` determines, which attribute of the “foreign” class should be used to build the relationship and `:local_key` determines which attribute of the local class will be considered.

The newly created groups method accepts an optional boolean argument. If you pass true, it returns an Array of Group objects. If you pass false, you'll get back an Array of Strings containing the dnattr of every group our customer belongs to.

If you know ActiveRecord, you'll know that if you have a method called belongs\_to(), there's likely to be a method called has\_many(), too:

File

```

Line 1  class Group < ActiveLDAP::Base
-       ldap_mapping :dnattr => 'cn', :prefix => 'ou=Groups'
-
-       has_many :members,
5         :class_name => 'Employee',
-         :local_key => 'gidNumber',
-         :foreign_key => 'gidNumber'
-     end
-
10  class Employee < ActiveLDAP::Base
-       ldap_mapping :dnattr => 'uid',
-                   :prefix => 'cn=employees,ou=Groups'
-     end
-
15  employees = Group.new('employees')
-     employees.members.each { |e| puts e.cn }

```

This outputs the following list:

```

Maik Schmidt
Carl Coder

```

There is really not much to say here: has\_many(member,options) works exactly as belongs\_to(member,options).

## Conclusion

ActiveLDAP is in an early stage of development and it will certainly take some time until all details are settled. Despite this it's an amazingly useful tool and makes working with LDAP a breeze.



## Chapter 3

# Processing XML

---

Exchanging data between processes, components, and companies has always been a vital part of enterprise software. Many attempts have been made to create a universal format for data exchange, but they all have failed for various technical and political reasons.

It's hard to believe that it took several decades before something like a standard for a platform-independent data representation was both created *and* accepted. The *eXtensible Markup Language* (XML) has, over the years, evolved into such an industry standard for portable data. That's because it has some useful characteristics:

*eXtensible Markup  
Language*

- It is plain text.
- It has been standardized by the W3C<sup>1</sup>.
- It is machine-independent (so low-level details such as byte ordering do not matter).
- It is easy to use.
- It supports international character sets.

XML is supported by all modern programming languages. The current Ruby distribution comes with good XML support, but compared to languages such as Java and C#, there is still a lot to be done. On the one hand it is easy to create and parse XML documents in Ruby. On the other hand Ruby lacks support for some important technologies such as Document Type Definitions (DTD), schema validation, and XSLT.

---

<sup>1</sup><http://www.w3.org/XML>

### REXML: What's The Difference?

Although several XML parsers exist for Ruby (for example, NQXML\* or xmlparser†), the most popular is Sean Russel's REXML.

The majority of XML parsers are based on either the SAX2 or DOM APIs. These have been standardized, and hence look the same in all programming languages. That's certainly a good thing, because if you're familiar with DOM programming in Java, you do not have to learn a lot to do DOM programming in C++ or Ruby.

The downside is that general approaches such as DOM are a compromise, and tend not to be tailored to exploit the strengths of a particular language. Sean Russel felt so, too, and while looking for better alternatives he found the Electric XML library for Java, created by a company called The Mind Electric.‡REXML is a pure Ruby implementation of the original Electric XML API.

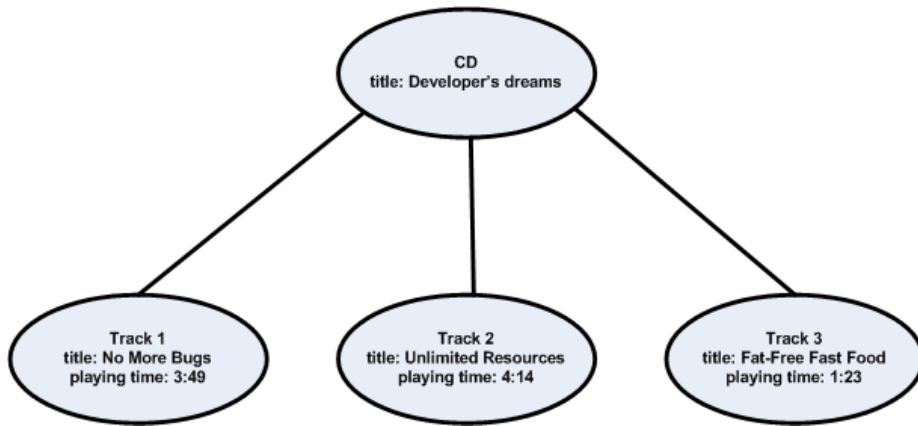
REXML is not a copy of the java API, but a genuine Ruby port. All classes and methods have been renamed to follow Ruby conventions and special Ruby features (such as iterators) have been used wherever it was possible and advantageous. In addition, REXML comes with a lot of things that were not part of the original Electric XML interface. There is support for SAX2, a proprietary stream parsing API, an experimental pull parser, and an experimental RELAX NG validator.

---

\*<http://nqxml.sourceforge.net>

†This is a binding for James Clark's expat XML parser. It can be found under <http://www.yoshidam.net/Ruby.html#xmlparser>.

‡The company is called Webmethods today and the Electric XML library is now integrated into their products. It's no longer available as a stand-alone product.




---

Figure 3.1: Tree Representation of a CD

---

It's unlikely that you can find a single company in this world that does not use XML by any means and a lot of enterprise data is no longer only stored in tables but between angle brackets. Hence, you better know how to extract it and in the following sections we'll cover the most important XML processing requirements: we'll show you how to create XML documents, how to parse them, and how to validate them.

### 3.1 A Short XML reminder

XML is a subset of the more flexible and more liberal *Standard Generalized Markup Language* (SGML). It allows you to define your own markup languages for describing data organized hierarchically, in a tree structure. For example, Figure 3.1 shows a possible tree representation of an audio CD. Its XML representation might look like this:

*Standard Generalized  
Markup Language*

```

Line 1  <?xml version='1.0' encoding='ISO-8859-1'?>
-
-      <!-- Comments look like this! -->
-      <cd title="Developer's Dreams">
5         <track id='1' title='No More Bugs' playing-time='3:49'/>
-         <track id='2' title='Unlimited Resources' playing-time='4:14'/>
-         <track id='3' title='Fat-Free Fast Food' playing-time='1:23'/>
-      </cd>
  
```

All XML documents must be well-formed, which roughly means:

- The document must have a single top-level element.

- All elements have to be closed explicitly and they have to be nested properly, i.e. `<a><b></a></b>` is not allowed, because element `<b>` must be closed before element `<a>`.
- Attributes always have a value and this value has to be set in single or double quotes. HTML attributes such as NOWRAP or `colspan=5` are not allowed in XML documents.

## Should I use elements or attributes?

Sometimes it's just a matter of taste, but more often it's a decision that should be made carefully.

The following cases force us to use elements:

- The information you want to describe can potentially occur more than once or can potentially have child elements. It's important to plan for such cases upfront—if you are not sure, use an element.
- Whitespace characters are significant.

In other cases, prefer attributes over elements:

- You do not have to worry about whitespace characters. Using attributes it's clear that “hello” differs from “\_hello\_”.
- Attributes often produce less noise and are more readable. For example, compare this

```
<person>
  <name>Homer</name>
  <middle-name>Jay</middle-name>
  <surname>Simpson</surname>
</person>
```

to this:

```
<person name="Homer" middle-name="Jay" surname="Simpson"/>
```

- Attributes are slightly faster, because they usually need less space than elements and therefore less text has to be processed by the XML parser (this is especially true for documents with long tag names for elements that get open and closed over and over again). In addition they increase parsing speed because of the inner structure of most XML parsers. Many XML parsers are event driven and use the SAX2 API. They search for the start tag of elements and whenever they find one, they call the `startElement()` method transmitting the element name and a list of all attributes belonging to the current element.

If you have a document fragment looking like this

```
<book>
  <title>Pragmatic Project Automation</title>
  <isbn>0974514039</isbn>
  <publisher>Pragmatic Bookshelf</publisher>
</book>
```

`startElement()` is called four times (for the elements `<book>`, `<title>`, `<isbn>`, and `<publisher>`) and calling methods in programming languages supporting polymorphism is expensive. When we use attributes instead of elements, our document will look like this:

```
<book title='Pragmatic Project Automation' isbn='0974514039'
      publisher='Pragmatic Bookshelf' />
```

Now `startElement()` is called only once for every `<book>` element. You might not consider this a big performance boost, but if you're Amazon.com and have to parse several hundred thousand books having dozens of elements, it certainly will matter.

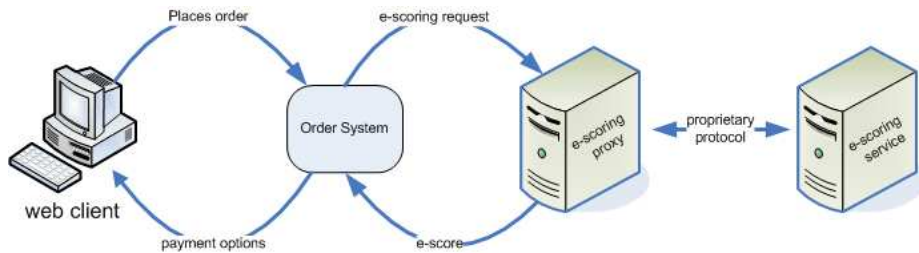
## 3.2 Generating XML documents

Generating XML documents is often necessary for communicating with other systems. If you are using technologies such as SOAP or XML-RPC you don't have to worry about the XML generation yourself, because it will be done under the hood by supporting libraries. But there are still many applications today expecting XML documents that you have to create "manually."

In this section we'll show you various techniques for generating XML documents. You'll see how to create documents using rather naive approaches (such as writing raw strings). We'll then look at more sophisticated technologies, such as the REXML API.

### To Score Well

From the beginning, customers using PragBouquet's web shop could freely choose from various payment methods: prepaid, invoice, or credit card. But after some months you came to realize that there were actually customers who'd cheated you. They've ordered flowers but never paid for them. Therefore the company decided to buy a so called e-score application. This assigns a risk score to each of your new customers. A customer with a low e-score will only be allowed to order if he or she pays upfront.




---

Figure 3.2: e-score architecture

---

The e-score provider uses a proprietary protocol, but gave you a proxy application that hides all that stuff behind an XML/HTTP layer. It expects an XML file containing a list of customers and returns a similar document where every customer was assigned a risk score. You can see the architecture in Figure 3.2.

It's your task to convert a customer address into an XML document acceptable to the e-scoring application. Based on the response, you'll then decide which payment options will be offered to the customer.

The input documents are simple: they consist of a list of `<person>` elements. Of course, the e-scoring company—like every company employing more than two people—defined its own XML markup, looking like this:

File

```

<persons>
  <person name='Max' surname='Mustermann'>
    <address>
      <street>Musterstr. 42</street>
      <city>Berlin</city>
      <postal-code>11011</postal-code>
    </address>
  </person>
</persons>

```

## Generating XML Documents Using Raw Strings

Because XML documents are nothing but text, it's tempting to generate them using strings. So, let's start with a simple helper function that tags a certain value:

File

```

Line 1 def tag(tag_name, value, attrs = {})
      -   tmp = "<#{tag_name}"
      -   if attrs

```

```

-       attrs.each { |k,v| tmp += " #{k}='#{v}'" }
5     end
-     tmp + ">#{value}</#{tag_name}>\n"
-     end
-
-     puts tag('hello', 'world')
10    puts tag('a', 'b', { 'c' => 'd' })

```

produces:

```

<hello>world</hello>
<a c='d'>b</a>

```

For generating our `<person>` elements we'll take the object-oriented road—we'll create classes for both addresses and persons. Because they are only storage classes we use `Struct` to create them automatically, then add `to_xml()` methods to turn them into XML documents.

File

```

Line 1  Address = Struct.new(:street, :city, :postal_code)
-
-     class Address
-       def to_xml
5         tag('address',
-           tag('street', self.street) +
-           tag('city', self.city) +
-           tag('postal-code', self.postal_code)
-         )
10      end
-     end

```

One of the things that makes working with Ruby so much fun is reopening classes. After `Struct` created an `Address` class for us, we reopened its definition and added our `to_xml()` method. It works the same way for the `Person` class:

File

```

Line 1  Person = Struct.new(:name, :surname, :address)
-
-     class Person
-       def to_xml
5         tag('person',
-           self.address.to_xml, {
-             'name' => self.name,
-             'surname' => self.surname
-           }
10      )
-     end
-     end

```

Finally, we check if it all works together:

File

```

Line 1  address = Address.new(
-     'Musterstr. 42',

```

```

-     'Berlin',
-     '11011'
5  )
-     max_m = Person.new('Max', 'Mustermann', address)
-     puts max_m.to_xml

```

produces:

```

<person name='Max' surname='Mustermann'><address> \
  <street>Musterstr. 42</street>
<city>Berlin</city>
<postal-code>11011</postal-code>
</address>
</person>

```

Although everything looks fine, you should follow this approach only in the simplest cases, because it has some serious disadvantages. First of all, you cannot move around and refine document fragments. This is a pity, because XML is such a flexible format and it happens often that new elements or attributes get added to existing document structures. If you've worked exclusively with strings you have to either parse or manipulate them directly to add the new stuff.

Let's assume we have access to one of those brand-new localization services that determine the geographic coordinates of an address and you want to add this information to the XML representation of your Address objects without both adding a position attribute and changing the `to_xml()` method.

Accessing the localization service is easy: you give it the street, the house number, the postal code, and the city, and it returns a pair of coordinates:

```

address = Address.new(
  :street => 'Musterstr.',
  :house_number => '42',
  :postal_code => '11011',
  :city => 'Berlin'
)
coordinates = LocalizationService.locate(address)
puts coordinates.latitude   # -> 51.5245
puts coordinates.longitude  # -> 6.75

```

Representing the coordinates in XML would probably look like this:

```

<position latitude='51.5245' longitude='6.75'/>

```

How can you add this to an existing XML file containing `<address>` elements? You can try using regular expressions and all the fancy methods of the String class, but think about it for a moment: did you really



| Character | XML Entity | Character | XML Entity |
|-----------|------------|-----------|------------|
| '         | &apos;     | "         | &quot;     |
| <         | &lt;       | >         | &gt;       |
| &         | &amp;      |           |            |

---

Figure 3.3: XML Standard Entities

---

consider all special cases? What about comments or CDATA sections? What about `<address>` elements that do not belong to `<person>` elements? What about `<address>` elements that already have a `<position>` element?

You have to admit that it can get complicated. Sometimes it's nearly impossible to perform this kind of manipulation without parsing the document fragment, adding the new stuff using conventional DOM manipulation methods, and finally creating a new XML string again.

Believe it or not, but we still have some disadvantages left. For example, if you work with raw strings, chances are good that you forget to markup elements correctly as we did in our `tag()` function above. What if a person's address is "Main Street 7 & 8"? The resulting `<street>` element would be

```
<street>Main Street 7 & 8</street>
```

Every standards compliant XML parser will reject this, complaining that your document isn't well-formed—blanks are not allowed after an ampersand. Whenever the parser sees an ampersand, it assumes it introduces an entity reference, that has to have an alphanumeric name, ends with a semicolon, and has been defined in a *Document Type Definition* (DTD). Similar things will happen whenever you use one of XML's special characters. If you really want to use strings for generating your documents, you'll have to replace these characters with their standard entities shown in Figure 3.3.

*Document Type  
Definition*

Adding such a mechanism to our `tag()` method is easy:

```
def encode_markup(text)
    return " if text.nil? or text == "
    text.gsub!('&', '&amp;')
    text.gsub!("'", '&apos;')
    text.gsub!('"', '&quot;')
    text.gsub!('<', '&lt;')
    text.gsub!('>', '&gt;')
```

File

```

end

def tag(tag_name, value, attrs = nil)
  tmp = "<#{tag_name}"
  if attrs
    attrs.each { |k,v| tmp += " #{k}='#{encode_markup(v)}'" }
  end
  tmp + ">#{encode_markup(value)}</#{tag_name}>\n"
end

puts tag('favorite', 'Starsky & Hutch')

```

produces:

```
<favorite>Starsky & Hutch</favorite>
```

You'll face more subtle problems if you ignore character set issues (as we did in the `tag()` method). An XML document that does not explicitly specify a character set encoding in its header automatically is supposed to contain only UTF-8 characters. For ASCII texts this is perfect, but what if you have a customer from Germany having the popular surname "Müller?" In UTF-8 the German umlaut ü is represented as a two byte sequence (0xc2, 0x81), but it's a single byte (0xfc) in the character set ISO-8859-1.

Whenever you get text data from an external source, from a database, a file, a HTTP server, for example, you have to determine what character set encoding has been used.

If the specified encoding and the document's content do not match, your XML parser will reject it or—even worse—will misinterpret some characters. Before reading on, you should have a look at Joel Spolsky's awesome essay *The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)*<sup>2</sup>

For our documents we now have two choices: we can set the encoding attribute in the XML header correctly or we convert our final document into UTF-8. To convert texts between different character sets in Ruby, you can use the *Iconv* library. Its interface is simple: a single line of code converts a text encoded in the ISO-8859-1 character set into the UTF-8 character set.

```
Iconv.conv('utf-8', 'iso-8859-1', 'Müller')
```

---

<sup>2</sup><http://www.joelonsoftware.com/articles/Unicode.html>

Our final version of method `tag()` will use this, assuming that all element and attribute values are encoded in ISO-8859-1.

File

```

Line 1  require 'iconv'
-
-  def empty?(text) text.nil? or text == ""; end
-
5  def encode_markup(text)
-    return " if empty?(text)
-    text.gsub!('&', '&amp;')
-    text.gsub!('"', '&apos;')
-    text.gsub!("'", '&quot;')
10   text.gsub!('<', '&lt;')
-    text.gsub!('>', '&gt;')
-  end
-
-  def to_utf8(text)
15    Iconv.conv('utf-8', 'iso-8859-1', text)
-  end
-
-  def encode(value)
-    encode_markup(to_utf8(value))
20  end
-
-  def tag(tag_name, value, attrs = {})
-    tmp = "<#{tag_name}"
-    if !attrs.nil? and !attrs.empty?
25     attrs.each { |k,v| tmp += " #{k}='#{encode(v)}'" }
-    end
-    if !empty?(value)
-      tmp += ">#{encode(value)}</#{tag_name}>\n"
-    else
30     tmp += "/>\n"
-    end
-  end
-  end

```

This version is much better than our first one, but it's also much longer and it still has some flaws that cannot be solved easily. For example, it does not check if element and attribute names are valid according to the XML standard (be honest: do you know the rules?). Additionally, you cannot reformat the generated document—you do not have much control over indentation, line breaks, etc. Obviously, generating well-formed XML documents is not as simple as it seems.

## Generating XML Documents With REXML

Although REXML does not implement the original DOM interface, it offers an API based on trees. Using this API you can convert an XML document into trees and create trees that represent XML documents.

Everything starts with a document. With REXML you create it like this:

```
require 'rexml/document'
doc = REXML::Document.new
```

An empty document is as useful as an empty bottle of beer. Let's add an element to it.

```
root = REXML::Element.new('my-root')
doc.add_element(root)
puts doc.to_s
```

produces:

```
<my-root/>
```

Creating attributes makes our toolbox complete:

```
root.add_attribute('an-attribute', 'a-value')
puts doc.to_s
```

results in:

```
<my-root an-attribute='a-value'/>
```

Now we can turn our Address object into XML the right way.

```
File
Line 1  class Address
-       def to_xml
-         adr = REXML::Element.new('address')
-         adr.add_element('street').add_text(self.street)
5         adr.add_element('city').add_text(self.city)
-         adr.add_element('postal-code').add_text(self.postal_code)
-         adr
-       end
-     end
10
-     address.to_xml.write($stdout, 0)
```

produces:

```
<address>
  <street>Musterstr. 42</street>
  <city>Berlin</city>
  <postal-code>11011</postal-code>
</address>
```

That's how it should look: every element is created explicitly and the `to_xml()` method no longer returns a simple string, but a document fragment. In addition, we can now use the `write()` method. This allows us to control the string representation of an XML document. It expects an object derived from IO and the level of indentation to be used.

The result of the `to_xml()` method can be processed further by other methods now. Adding coordinates to an `Address` object, for example, can be done like this:

File

```
Line 1  adr = address.to_xml
-      pos = REXML::Element.new('position')
-      pos.add_attribute('longitude', '12.345')
-      pos.add_attribute('latitude', '56.789')
5      adr.add_element(pos)
-      adr.write($stdout, 0)
```

produces:

```
<address>
  <street>Musterstr. 42</street>
  <city>Berlin</city>
  <postal-code>11011</postal-code>
  <position latitude='56.789' longitude='12.345' />
</address>
```

REXML correctly encodes markup characters, but you still cannot ignore character set encoding issues. REXML internally uses the UTF-8 character set and so you have to encode all strings before inserting them into a REXML document and you have to decode them accordingly when reading them back.

We already saw how to achieve this using the `Iconv` library in Section 3.2, *Generating XML Documents Using Raw Strings*, on page 80. You can also use Ruby's `unpack()` and `pack()` methods. `"hello".unpack("C*").pack("U*")` turns a string into UTF-8 and `"hello".unpack("U*").pack("C*")` does the opposite.

## Builder

As we've seen, building well-formed XML documents is not a trivial task. So people constantly try to simplify it. One of those people is Jim Weirich, who produced the `Builder` library for Ruby.<sup>3</sup> Its core class is `Builder::XmlMarkup`, which provides everything you need to generate well-formed XML documents. For example,

File

```
Line 1  require 'rubygems'
-      require 'builder'
-
-      doc = Builder::XmlMarkup.new
5      doc.person(:name => 'Max', :surname => 'Mustermann')
-      puts doc.target!
```

---

<sup>3</sup><http://builder.rubyforge.org>

prints this little document:

```
<person surname="Mustermann" name="Max"/>
```

The technique is probably familiar to Ruby fans: Builder defines a `method_missing()` handler that catches calls to unknown methods and turns them into XML tags with the same name as the method that was called originally. In addition, a hash of parameters is turned into attributes of the newly created element.

The resulting document can be obtained by calling `target!()` as we did in line 6.

To build hierarchical documents, `XmlMarkup` has a nice syntax: if you pass a code block to one of those “unknown” methods, it gets passed the current element automatically:

```
File
Line 1  xml = ''
        - doc = Builder::XmlMarkup.new(:target => xml)
        - doc.person(:name => 'Max', :surname => 'Mustermann') { |person|
        -   person.address { |address|
        5     address.street('Hauptstr. 42')
        -   }
        - }
        - puts xml
```

produces:

```
<person surname="Mustermann" name="Max"><address> \
  <street>Hauptstr. 42</street></address></person>
```

Intuitive, isn't it? Note that in line 2 we have specified the `target` option and set it explicitly to a String object. Hence, Builder fills up the `xml` variable with our document. The `target` option accepts any object that responds to the `<<(text)-operator`.

That's all very nice, but the formatting of the result document is, let's say, suboptimal. Fortunately, there is the `indent` option:

```
File
Line 1  xml = ''
        - doc = Builder::XmlMarkup.new(:target => xml, :indent => 2)
        - doc.person(:name => 'Max', :surname => 'Mustermann') { |person|
        -   person.address { |address|
        5     address.street('Hauptstr. 42')
        -     address.tag!('postal-code', '12345')
        -     address.city('Musterstadt')
        -   }
        - }
        10 puts xml
```

prints this beautiful XML document:

```

<person surname="Mustermann" name="Max">
  <address>
    <street>Hauptstr. 42</street>
    <postal-code>12345</postal-code>
    <city>Musterstadt</city>
  </address>
</person>

```

There's also a `margin` option, which specifies the indentation offset, so you can format your XML documents any way you like.

Did you notice the little trick with the postal code in line 6? `postal-code()` is not a valid method name in Ruby, but `<postal-code>` is a perfectly valid XML tag. To get around this, Builder lets you explicitly insert tags using the `tag!(sym,*args,&block)` method.

To make sure that we do not get into trouble when XML documents without an explicit encoding get prohibited by federal law, we better add another line of code:

```

File
Line 1  xml = ''
- doc = Builder::XmlMarkup.new(:target => xml, :indent => 2)
- doc.instruct!
- doc.person(:name => 'Max', :surname => 'Mustermann') { |person|
5   person.address { |address|
-     address.street('Hauptstr. 42')
-     address.tag!('postal-code', '12345')
-     address.city('Musterstadt')
-   }
10  person.position(:longitude => '12.345', :latitude => '56.789')
- }
- puts xml

```

produces this perfect XML document:

```

<?xml version="1.0" encoding="UTF-8"?>
<person surname="Mustermann" name="Max">
  <address>
    <street>Hauptstr. 42</street>
    <postal-code>12345</postal-code>
    <city>Musterstadt</city>
  </address>
  <position longitude="12.345" latitude="56.789"/>
</person>

```

That's nearly all you have to know to create XML documents with Builder, but for some special cases you'd might need the following methods, too:

`text!(text)`

allows you to create elements with mixed content:

```
File doc = Builder::XmlMarkup.new(:indent => 2)
doc.foo { |f|
  f.bar
  f.text! "I live outside the bar!\n"
}
puts doc.target!
```

prints:

```
<foo>
  <bar/>
  I live outside the bar!
</foo>
```

`cdata!(data)`

inserts a CDATA section into an XML document:

```
File doc = Builder::XmlMarkup.new
doc.cdata!('Do not run with scissors!')
puts doc.target!
```

prints:

```
<![CDATA[Do not run with scissors!]]>
```

`comment!(text)`

inserts a comment into an XML document:

```
File doc = Builder::XmlMarkup.new
doc.comment!('Some comments are totally useless!')
puts doc.target!
```

prints:

```
<!-- Some comments are totally useless! -->
```

`declare!(instruction,*args,&block)`

allows you to insert DTD declarations into your document:

```
File doc = Builder::XmlMarkup.new
doc.declare!(:ENTITY, :pp, 'Pragmatic Programmers')
puts doc.target!
```

prints:

```
<!ENTITY pp "Pragmatic Programmers">
```

## Conclusion

It should be clear by now that creating XML documents is by no means as simple as it seems. Because of this, you've probably already received or created documents that were not well-formed or that contained characters that did not match the document's encoding.



Although it is tempting to use a raw string approach, it does not scale very well and has a lot of drawbacks. Compared to using REXML or Builder it's even more complex and results in more code (and ugly code). From a pragmatic point of view it is certainly beneficial to do it correctly from the beginning. In *Refactoring to Patterns* [?] you will even find patterns that help you to refactor "I absolutely wanted to create XML myself" code.

### 3.3 Processing XML Documents

Imagine you've sent a bunch of flowers to a person you like. Wouldn't it be interesting to know if he or she got the flowers on time, or even if the person refused the flowers? Fortunately, all big parcel services offer their customers the opportunity to check the status of a parcel using the Internet.

PragBouquet works together with several parcel services. Although we could simply give our customers the tracking numbers assigned by the particular service to their order, that means that they'd then need to visit the shipping service's site and enter that number to track their flowers. That isn't great customer service on our part.

A much better solution would be periodically to track the status of all undelivered orders in our system and then inform our customers about all important events in the life of a deliver via e-mail or SMS messages. To do this, we'll have to take a closer look at the parcel service's web services.

A typical parcel tracking result looks like this:

```
File
<tracking-result>
  <parcel-history tracking-no='2X42'>
    <event timestamp='2005-05-02T04:05:00'
      state='picked-up'/>
    <event timestamp='2005-05-02T08:30:00'
      state='first-delivery-attempt'/>
    <event timestamp='2005-05-03T09:05:00'
      state='second-delivery-attempt'/>
    <event timestamp='2005-05-04T10:15:00'
      state='refused-damaged'>
      <consignee>Mrs. Smith</consignee>
    </event>
    <event timestamp='2005-05-04T19:07:00' state='returns-to-sender'/>
  </parcel-history>

  <parcel-history tracking-no='2X43'>
```

```

    <event timestamp='2005-05-02T04:25:00' state='picked-up'/>
    <event timestamp='2005-05-02T09:15:00' state='delivered'>
      <consignee>Mr. Gumble</consignee>
    </event>
  </parcel-history>

  <parcel-history tracking-no='2X44'>
    <event timestamp='2005-05-02T03:55:00' state='picked-up'/>
  </parcel-history>
</tracking-result>

```

Here we have the history of three different parcels, each identified by a unique tracking number:

- Parcel 2X42 was picked up by the parcel service at the production site at 4:05 am on May 2nd. They tried to deliver it on the same day and later on May 3rd without success. On May 4th the driver finally met Mrs. Smith, but she refused to accept the parcel because it was damaged. Now it's on its way back home.
- Parcel 2X43 is a textbook example: Picked up, delivered.
- Parcel 2X44 was picked up, but has not yet been delivered.

In the following sections we'll demonstrate how to parse a document such as this using different approaches.

## XML Processing With REXML

One of the key problems when dealing with XML documents is parsing them and representing the result in our programs. Currently, there are two major parsing schemes: tree parsing and stream parsing. A tree parser reads an XML document as a whole and represents it as a tree in memory. Stream parsers expect you to provide a so-called listener that is invoked whenever the parser finds a new element. Hence, *all* elements get processed in the order of their start tags and the listener is responsible for handling them.

Both approaches have their advantages and disadvantages. Tree parsers are significantly slower and consume much more memory. On the upside, they give you convenient access to the tree's nodes. On the other hand, stream parsers are fast and do not consume much memory, but they force you to organize the elements yourself. Choosing the right way to parse your documents depends to a large extent on your performance needs and on how you want to process the document. If you want to process every node, a stream parser is what you need. If

you often have to access different node sets, a tree parser might be a better choice.

There are standard APIs for both tree and stream parsing: DOM and SAX2 respectively. DOM has been standardized by the W3C and SAX2 is an open standard maintained by David Megginson.<sup>4</sup> Because it's so simple, there's not much discussion about the usefulness of the SAX2 API. DOM, though, has always been something of an *enfant terrible*. Its inventors wanted the DOM API to work in as many environments and programming languages as possible. Even COBOL programmers should have their DOM parser. This requirement turned DOM into the beast it is today. Ironically, nearly every programming platform now has an additional convenience API for tree parsing. Java, for example, has JDOM<sup>5</sup> and XOM<sup>6</sup>.

REXML is no exception to this rule and supports both tree and stream parsing. For reasons explained in the preceding paragraphs the only standard API it supports is SAX2. Its tree parsing API is completely proprietary, but meets the needs of a Ruby programmer much better than DOM.

## Tree Parsing

Easy things should be easy. Turning an XML document into a tree could not be much easier than this:

File

```
Line 1  require 'rexml/document'
-       include REXML
-
-       doc = Document.new("<sample>Our first example.</sample>")
5       print doc.root.name, ": ", doc.root.text, "\n"
```

produces:

```
sample: Our first example.
```

The first two lines load the REXML module and include its namespace into our code, so we don't have to prefix all its class names with REXML:: over and over again (from now on we're going to omit these lines). By calling `Document.new()` with our little example document in line 4 we turn a string into an instance of class `Document`. The `new()` method accepts parameters of different types:

---

<sup>4</sup><http://www.saxproject.org>

<sup>5</sup><http://www.jdom.org>

<sup>6</sup><http://www.xom.nu>

- Instances of class REXML::Document will get copied.
- Strings containing XML documents will be parsed and turned into instances of class REXML::Document.
- Instances of class IO will be read and then parsed. For example, to parse a file called example.xml you'd call

```
Document.new(File.new('example.xml'))
```

Finally, the last line of our example program prints the name and the content of our document's root element, which is an instance of class REXML::Element. Not surprisingly, the root element can be accessed using a document's root() method.

Our first successful parsing attempts should give us enough confidence to tackle our original problem: processing parcel tracking results. Let's load and parse our document and play around a bit with its elements and attributes.

```
File
Line 1  doc = Document.new(File.new('packages.xml'))
        - tracking_results = doc.root
        - first_parcel = tracking_results.elements[1]
        - tracking_no = first_parcel.attributes['tracking-no'] # -> 2X42
```

All REXML elements (including the root element) have an accessor called elements containing an array of the element's children. This array is indexed starting at 1, not 0, so to get the first child of the root element you have to call doc.root.elements[1] instead of doc.root.elements[0]. As a shortcut, you can index the element directly to access its children: doc.root[1].

Consistently, every element also has an attributes accessor that contains all the element's attributes. It can be used like a Hash, so the attribute tracking-no= of our first <parcel-history> element can be found in doc.root[1].attributes['tracking-no'].

It's nice that accessing elements and attributes is easy with REXML, but it's not exactly what we need, because we do not know upfront how many parcel histories we will get from our provider. It would be nice to be able to iterate over a set of elements matching certain criteria. For example, we might want to get at all the children of an element with a certain name. each\_element() to the rescue:

```
File
doc.root.each_element('parcel-history') do |ph|
  puts ph.attributes['tracking-no']
end
```

produces:

2X42  
2X43  
2X44

It seems that `each_element()` allows you to iterate over all children having a particular name, but that's only half the story. In fact, `each_element()` expects a so-called *XPath* expression. XPath can be used for describing nearly arbitrary node sets. We'll talk more about XPath in Section 3.3, *XPath*, on page 111, but for now we'll try to get by with what we have already.

```
File
Line 1  require 'time'
-       require 'rexml/document'
-       include REXML
-
5       Event = Struct.new(:timestamp, :state, :consignee)
-
-       class Event
-         def to_s
-           self.timestamp.strftime('%Y-%m-%d %H:%M') + ": " + self.state
10        end
-       end
-
-       class ParcelHistory
-         attr_reader :tracking_no, :events
15
-         def initialize(tracking_no)
-           @tracking_no = tracking_no
-           @events = []
-         end
20
-         def add_event(event) @events << event; end
-       end
-
-       class ParcelHistoryParser
25         def initialize(source)
-           @doc = Document.new(source)
-         end
-
-         def each_parcel
30           @doc.root.each_element('parcel-history') do |ph|
-             history = ParcelHistory.new(ph.attributes['tracking-no'])
-             ph.each_element("event") do |event_element|
-               history.add_event(to_event(event_element))
-             end
35           yield history
-         end
-       end
-     end
```

```

-   private
40
-   def to_event(element)
-       timestamp = Time::xmlschema(element.attributes['timestamp'])
-       state = element.attributes['state']
-       consignee = nil
45     if !element.elements['consignee'].nil?
-         element.elements['consignee'].text
-       end
-       Event.new(timestamp, state, consignee)
-   end
50 end
-
-   parser = ParcelHistoryParser.new(File.new('packages.xml'))
-   parser.each_parcel do |history|
-       puts history.tracking_no + ":"
55   history.events.each { |e| puts "  " + e.to_s }
-   end

```

produces:

```

2X42:
2005-05-02 04:05: picked-up
2005-05-02 08:30: first-delivery-attempt
2005-05-03 09:05: second-delivery-attempt
2005-05-04 10:15: refused-damaged
2005-05-04 19:07: returns-to-sender
2X43:
2005-05-02 04:25: picked-up
2005-05-02 09:15: delivered
2X44:
2005-05-02 03:55: picked-up

```

Admittedly, for Ruby this is a really big example, but we will dissect it class by class. Classes `Event` and `ParcelHistory` are only for storage purposes. `Event` encapsulates all attributes describing an event in the lifetime of a parcel and `ParcelHistory` accumulates a list of such events.

`Event` is pretty trivial, so we let class `Struct` create it for us automatically. In lines 8 to 10 we reopen it to add our own `to_s()` method. Because `ParcelHistory` contains an array (events), which has to be managed “manually”, we could not use `Struct` to create it, too. `Struct` should be used only for the simplest cases, when you have a list of atomic attributes that do not depend on each other and do not demand any logic while initializing, setting, and getting them.

The real fun begins on line 13. Class `ParcelHistoryParser` provides all the functionality that turns a parcel history document from all sources supported by the `REXML` parser into a list of `ParcelHistory` objects.

## Mixed Content

In line 46 of our parcel parser example we determine the name of a consignee by calling `text()` on an instance of `Element`. In our case this is perfect, because we expect any `<consignee>` element to have exactly one child node of type `text`. What would happen if it had more than one? The answer is simple: `text()` only returns the first child node of type `text`.

You have to be careful when dealing with so called mixed content—elements that have child nodes of different types. Given the following document:

File

```
doc = Document.new(<<-XML)
<root>
  <a>
    First!
    <b>xyz</b>
    Second!
    <b>abc</b>
    Third!
  </a>
</root>
XML
```

the code

```
puts doc.elements['root/a'].text
```

produces:

```
First!
```

If we use the `texts()` method, with code like this:

```
puts doc.elements['root/a'].texts
```

we'll instead see

```
First!
```

```
Second!
```

```
Third!
```

Sometimes you do not parse an XML document yourself, but get it as a `REXML::Document` object from another method. In these cases be prepared to get elements that do not contain mixed content, but more than one text node. Depending on the way the elements were built, it can happen easily:

File

```
e = Element.new('example')
e.add_text('foo')
e.add_text(Text.new('bar'))
puts e.texts.size # => 2
```

Right at the beginning of our code, in `initialize(source)`, you can see one of the biggest advantages of dynamic languages such as Ruby: we initialize the attribute `doc` with our parsed document without caring where the document came from originally. As we explained above, REXML accepts an XML string, an object derived from IO, or another REXML::Document instance. Automatically, our `initialize(source)` method behaves in the same liberal fashion. For unit testing we can feed our parser with constant strings or files and in production it will probably get a Socket object or something similar without changing a single line of code.

Our next method is `each_parcel()`. It's a Ruby *iterator*. Iterators invoke a block of code for every member of a collection. Within `each_parcel()` we iterate over all `<parcel-history>` elements and for each of these elements we iterate over all `<event>` elements. Every `<event>` element is converted into an Event object by calling `to_event(element)` and all the Event objects are added to their corresponding ParcelHistory object. The `yield()` call in line 35 invokes the code block that is associated with `each_parcel()`, passing it the current ParcelHistory object.

At the end of our program, we finally use our ParcelHistoryParser to write the content of our example document to the console. Note that we have solved only the mechanical part of our problem—we have parsed the tracking results and represented them in our own class hierarchy. Academics call this process *deserialization* or *XML data binding*. Because it really is mechanical there are tools out there that will do it automatically for you. We'll have a look at such a tool in Section 3.3, *XML Processing With XmlSimple*, on page 120.

For the final solution the results still have to be interpreted—we need a class that determines the actual state of each package, stores it in a database, and sends e-mails or SMS messages, but that has nothing to do with XML. In this section we wanted to show you how you can process an XML document using the basic functions of REXML.

## REXML Stream Parsing

Last Valentine's Day was chaotic: PragBouquet's parcel tracking system nearly broke down under the heavy load. Mother's Day is looming, so you did some performance tests and found out that one of the biggest bottlenecks was the XML processing. Obviously, loading several thou-



sand tracking results into memory was not a good idea, particularly as you then process all the tracking results serially anyway.

It would be much better if you could parse the XML documents serially, element by element. Fortunately REXML's stream parsing API allows you to do just that. Stream parsers always work the same way: they process an XML document character by character and invoke methods on a so called listener class whenever an interesting event occurs: when a start tag was found, text was found, a comment was found, and so on.

You can invoke the REXML stream parser by calling `REXML::Document.parse_stream(source,listener)`, where `source` is an XML source and `listener` is your listener class. Let's see what happens if we pass an empty listener.

File

```
class ParcelHistoryListener; end
Document.parse_stream(
  File.new('packages.xml'),
  ParcelHistoryListener.new
)
```

This produces:

```
c:/ruby/lib/ruby/1.8/rexml/parsers/streamparser.rb:37:in 'send':
undefined method 'xmldecl' for #<ParcelHistoryListener:0x2ad5620> \
(NoMethodError)
from c:/ruby/lib/ruby/1.8/rexml/parsers/streamparser.rb:37:in 'parse'
from c:/ruby/lib/ruby/1.8/rexml/document.rb:171:in 'parse_stream'
from sp_empty_listener.rb:8
```

Not surprisingly, the stream parser complains that it could not find the `xmldecl()` method in our `ParcelHistoryListener` class. Obviously, this method is invoked when the stream parser finds an XML declaration, and that's the first thing in our tracking results document.

Let's do some more reverse engineering (that's more exciting than reading API docs, isn't it?) by adding `method_missing(method_id,*args)` to our listener class. This is a Ruby standard method that's called whenever an undefined method is called in an object. `method_id` is the name of the method called (as a symbol) and `args` are any arguments that were passed to it.

File

```
class ParcelHistoryListener
  def method_missing(method_id, *args)
    puts "Method '#{method_id.id2name}' was called."
  end
end

Document.parse_stream(
```

```

    File.new('packages.xml'),
    ParcelHistoryListener.new
  )

```

This produces:

```

Method 'xmldecl' was called.
Method 'comment' was called.
Method 'tag_start' was called.
Method 'text' was called.
Method 'tag_start' was called.
Method 'text' was called.
Method 'tag_start' was called.
Method 'tag_end' was called.
Method 'text' was called.
Method 'tag_start' was called.
...
Method 'tag_end' was called.
Method 'text' was called.
Method 'comment' was called.
Method 'text' was called.
Method 'comment' was called.
Method 'text' was called.

```

At the very least, the stream parser expects us to implement methods called `xmldecl()`, `comment()`, `tag_start()`, `text()`, and `tag_end()`. We stop playing detective now and have a look at the complete list of possible events:

`xmldecl(version, encoding, standalone)`

Called when the parser encounters an XML declaration. The parameters directly reflect an XML declaration's attributes, so for the declaration

```
<?xml version='1.0' encoding='iso-8859-1'?>
```

the parser would invoke `xmldecl(version='1.0', encoding='iso-8859-1', standalone=nil)`. Note that the default value for encoding and standalone is `nil`, which is different from the XML standard's default values ('utf-8' and 'no').

`tag_start(name, attrs)`

Called when the beginning of an element is found. `name` contains the element's name and `attrs` is an array containing the element's attributes, where every list item is a two-dimensional array consisting of the attribute's name and its value. If the element has no attributes, `attrs` is an empty array. For example, given the element

```
<person name='Homer' age='45'/>
```

the parser invokes

```
tag_start('person', [['name', 'Homer'], ['age', '45']])
```

tag\_end(name)

Called when the end of element `<name>` was found.

text(text)

Called when text is encountered in the document. For nicely formatted documents, this method is invoked more often than you might think, because by default whitespace in your document is significant. The actual text is passed as the parameter.

cdata(content)

Called when a CDATA section has been found. Parameter content contains the content of the CDATA section including all whitespace characters.

comment(text)

Called when a comment has been found. The comment text is passed in text without the `<!--` and `-->` sequences.

instruction(target, instruction)

Processing instructions were introduced into XML for passing information to particular applications reading a document. They have the following format:

```
<?target [instructions depending on application]?>
```

Common targets are “xml-stylesheet” and “php”. Whenever the stream parser detects a processing instruction it calls `instruction()` setting target and instruction accordingly. For example, the processing instruction

```
<?xml-stylesheet href='sitestyle.css' type='text/css'?>
```

would call

```
instruction("xml-stylesheet", "href='sitestyle.css' type='text/css'")
```

doctype(root\_name, pub\_sys, long\_name, uri)

Documents may be associated with a DTD that can be used for validating the document’s content. DTDs come in different flavors:

- `<!DOCTYPE root SYSTEM "http://path/to/dtd">`
- `<!DOCTYPE rss PUBLIC`  
`"-//Netscape Communications//DTD RSS 0.91//EN"`  
`"http://my.netscape.com/publish/formats/rss-0.91.dtd">`

- `<!DOCTYPE books [  
     <!ELEMENT books (book*)>  
     <!ATTLIST book title CDATA #IMPLIED isbn CDATA #IMPLIED>  
 ]>`

If a document contains a document type declaration (there is only one allowed), the stream parser calls the `doctype()` method, setting all parameters accordingly. All attributes that have not been declared in the document type declaration will be set to `nil`.

`elementdecl(declaration)`

Called when an element declaration like “`<!ELEMENT books (book*)>`” was found in a document type definition.

`attlistdecl(element_name, attributes, raw_content)`

Called for every attribute list declaration in a DTD. Parameter `element_name` contains the name of the element the attribute list has been defined for, `attributes` is a Hash containing all the attributes and `raw_content` contains the original declaration from the DTD. For example, the declaration

```
<!ATTLIST book title CDATA #IMPLIED isbn CDATA #IMPLIED>
```

would result in the following method call:

```
attlistdecl(  
  'book',  
  { 'title' => '', 'isbn' => '' },  
  '<!ATTLIST book title CDATA #IMPLIED isbn CDATA #IMPLIED>'  
)
```

`entity(entity_name)`

Called when an entity reference (such as “`%shortcut;`”) is found in a document type declaration.

`entitydecl(declaration)`

Called when an entity declaration (such as “`<!ENTITY ms Maik Schmidt>`”) is found in a document type definition.

`notationdecl(content)`

If you want to embed non-XML content (such as images or audio files) in your document, you can describe it in more detail using so called *notations* in a DTD. The stream parser calls `notationdecl()` when it finds a notation type attribute in a DTD. For

```
<!NOTATION gif SYSTEM 'image/gif'>
```

the parser would invoke

```
notationdecl(['gif', 'SYSTEM', 'image/gif'])
```

This attribute type is rarely used in practice.

That's a lot of methods, but we already knew that XML is a fairly complex beast. For building our own listener classes it seems that we have only a few options: we could implement all of the methods above, leaving some of them empty, or we could implement only the methods we definitely need and define `method_missing()` to suppress the rest. Because this is such a common scenario, REXML has a predefined solution for it, namely the `REXML::StreamListener`.

`REXML::StreamListener` is a template module that can be used when building your own listeners. It provides an empty implementation for all methods that can potentially be invoked by the stream parser. You include it in your listener class:

File

```
Line 1  require 'rexml/streamlistener'
-
-  class ParcelHistoryListener
-    include StreamListener
5
-    def tag_start(tag_name, attrs)
-      puts tag_name
-    end
-  end
10
-  Document.parse_stream(
-    File.new('packages.xml'),
-    ParcelHistoryListener.new
-  )
```

produces:

```
tracking-result
parcel-history
event
event
event
event
consignee
event
parcel-history
event
event
consignee
parcel-history
event
```

That's highly convenient, especially because many of the stream parser events are related to DTDs, which aren't used often nowadays.

We should not forget that we still have a problem to solve: the tree parsing approach was too slow and we wanted to enhance it using a stream parser. Our listener class might look like this:

```

File
Line 1  class ParcelHistoryListener
-       include StreamListener
-
-       def initialize
5         @in_consignee = false
-       end
-
-       def tag_start(tag_name, attrs)
-         if tag_name == 'parcel-history'
10          @parcel_history = ParcelHistory.new(attrs['tracking-no'])
-         elsif tag_name == 'event'
-           timestamp = Time.xmlschema(attrs['timestamp'])
-           state = attrs['state']
-           @event = Event.new(timestamp, state, nil)
15        elsif tag_name == 'consignee'
-          @in_consignee = true
-        end
-      end
-
20      def tag_end(tag_name)
-        if tag_name == 'parcel-history'
-          puts @parcel_history.tracking_no + ":"
-          @parcel_history.events.each { |e| puts ' ' + e.to_s }
-        elsif tag_name == 'event'
25          @parcel_history.add_event(@event)
-        elsif tag_name == 'consignee'
-          @in_consignee = false
-        end
-      end
-
30      def text(value)
-        if @in_consignee
-          @event.consignee = " unless @event.consignee
-          @event.consignee << value
35        end
-      end
-    end
-  end
-
-  Document.parse_stream(
40    File.new('packages.xml'),
-    ParcelHistoryListener.new
-  )

```

produces:

```

2X42:
2005-05-02 04:05: picked-up
2005-05-02 08:30: first-delivery-attempt

```

```

2005-05-03 09:05: second-delivery-attempt
2005-05-04 10:15: refused-damaged
2005-05-04 19:07: returns-to-sender
2X43:
  2005-05-02 04:25: picked-up
  2005-05-02 09:15: delivered
2X44:
  2005-05-02 03:55: picked-up

```

Here we have a classical stream parsing example., We only had to implement three `StreamListener` methods: `tag_start()`, `tag_end()`, and `text()`. The biggest difference between tree parsing and stream parsing is that stream parsers force you to maintain state yourself. For example, as you can see in lines 32 to 35, we are only interested in text nodes if we are currently in a `<consignee>` element. Therefore we store this state in the boolean instance variable `@in_consignee`.

Method `tag_start()` is responsible for creating new elements. Whenever we encounter a new `<parcel-history>` or `<event>` element, we create a corresponding `ParcelHistory` or `Event` object and store it in an instance variable. The only exception to this are `<consignee>` elements.

### But it's not Standard

Long before XML became more popular among software developers than caffeine, there were no standards except the XML standard itself. If you wanted to process XML documents you were forced to write your own parser or to use one of countless proprietary solutions. To overcome this awkward situation, a group of developers collaboratively developed an API on the XML-DEV mailing list and called it Simple API for XML (SAX). This happened in 1998 and SAX version 1.0 quickly became the de facto industry standard for XML stream parsing. The current version is called SAX2 and was released in 2002. Since then nothing has changed in the API.

Due to the nature of stream parsing, the proprietary REXML approach and the SAX2 API are similar. You have to define a listener whose methods are invoked by the parser, whenever it encounters an interesting event.

For the sake of completeness we'll show you the whole `REXML::SAX2Listener` class, but we often refer to `REXML::StreamListener` to avoid redundancy.

```
start_document()
```

Called at the beginning of a document.

`end_document()`

Called at the end of a document.

`start_element(uri, localname, qname, attributes)`

Called when the beginning of a new element is encountered. `uri` contains the namespace URI of the element (an empty string when the element is not associated with a namespace). If the element is associated with a namespace, `localname` contains the element's name without the namespace prefix, otherwise it is empty. `qname` contains the element's qualified name—the name as it appeared in the XML document. It might be empty for elements associated with a namespace URI. `attributes` contains all the element's attributes.

`end_element(uri, localname, qname)`

Called when the end of an element was encountered.

`cdata(content)`

See the description in Section 3.3, *REXML Stream Parsing*, on page 98.

`characters(text)`

See description of `text()` in Section 3.3, *REXML Stream Parsing*, on page 98.

`comment(comment)`

See description in Section 3.3, *REXML Stream Parsing*, on page 98.

`start_prefix_mapping(prefix, uri)`

Called at the beginning of a namespace declaration. `prefix` is the namespace prefix and `uri` the namespace URI.

`end_prefix_mapping(prefix)`

Called at the end of a namespace declaration, when the current namespace goes out of scope.

`processing_instruction(target, data)`

See description of `instruction()` in Section 3.3, *REXML Stream Parsing*, on page 98.

`doctype(name, pub_sys, long_name, uri)`

See description in Section 3.3, *REXML Stream Parsing*, on page 98.

`elementdecl(content)`

See description in Section 3.3, *REXML Stream Parsing*, on page 98.



attlistdecl(element, pairs, contents)

See description in Section 3.3, *REXML Stream Parsing*, on page 98.

entitydecl(content)

See description in Section 3.3, *REXML Stream Parsing*, on page 98.

notationdecl(content)

See description in Section 3.3, *REXML Stream Parsing*, on page 98.

Let's rewrite our ParcelHistoryListener using the SAX2 parser.

File

```

Line 1  require 'time'
-       require 'rexml/parsers/sax2parser'
-       require 'rexml/sax2listener'
-
5       class ParcelHistoryListener
-         include REXML::SAX2Listener
-
-         def initialize
-           @in_consignee = false
10        end
-
-        def start_element(uri, localname, tag_name, attrs)
-          if tag_name == 'parcel-history'
-            @parcel_history = ParcelHistory.new(attrs['tracking-no'])
15          elsif tag_name == 'event'
-            timestamp = Time::xmlschema(attrs['timestamp'])
-            state = attrs['state']
-            @event = Event.new(timestamp, state, nil)
-          elsif tag_name == 'consignee'
20            @in_consignee = true
-          end
-        end
-
-        def end_element(uri, localname, tag_name)
25          if tag_name == 'parcel-history'
-            puts @parcel_history.tracking_no + ":"
-            @parcel_history.events.each { |e| puts ' ' + e.to_s }
-          elsif tag_name == 'event'
-            @parcel_history.add_event(@event)
30          elsif tag_name == 'consignee'
-            @in_consignee = false
-          end
-        end
-
35        def characters(value)
-          if @in_consignee
-            @event.consignee = " unless @event.consignee
-            @event.consignee << value
-          end
40        end

```

```

-   end
-
-   parser = REXML::Parsers::SAX2Parser.new(File.new('packages.xml'))
-   parser.listen(ParcelHistoryListener.new)
45  parser.parse

```

We already knew that both APIs were similar, but isn't it surprising how similar they really are? In the `ParcelHistoryListener` class itself we had to change only the names and the signature of some methods. `tag_start(tag_name,attrs)` became `start_element(uri,localname,tag_name,attrs)`, `tag_end(tag_name)` became `end_element(uri,localname,tag_name)`, and `text(value)` became `characters(value)`. Instead of `StreamListener` we had to include `SAX2Listener` but not a single line of business logic had to be changed.

Because it's not part of the `REXML::Document` class, invoking the parser is a bit different. Additional parsers can be found in the `rexml/parsers` directory and reside in the `REXML::Parsers` namespace. In the last three lines of our program, we create the `REXML::Parsers::SAX2Parser`, pass it our implementation of a `REXML::SAX2Listener`, and eventually start the parsing process.

SAX was originally defined to make XML processing easier for Java, so its design reflects the typical shortcomings of a more or less static programming language. Sean Russel realized this and added some sugar to REXML that make it more, let's say, Rubyesque.

For example, if you are only interested in a small subset of your document's nodes, you don't have to implement a complete listener class. It's possible to associate SAX2 events with a code block. To extract all text nodes from the following document:

File

```
<?xml version="1.0" encoding="iso-8859-1"?>
```

```

<todo-list>
  <to-do>Write a book.</to-do>
  <to-do>Learn a new programming language.</to-do>
  <to-do>Don't Repeat Yourself</to-do>
</todo-list>

```

We could do the following:

File

```

Line 1  require 'rexml/parsers/sax2parser'
-
-   todo_list = File.new('todo.xml')
-   parser = REXML::Parsers::SAX2Parser.new(todo_list)
5   parser.listen(:characters) { |text| puts "* #{text}" }
-   parser.parse

```

and it produces:

```

Line 1  *
-
-      * Write a book.
-      *
5
-      * Learn a new programming language.
-      *
-
-      * Don&apos;t Repeat Yourself
10     *
-     *

```

Hmm, that's not exactly what we expected, is it? The standard entity &apos; has not been resolved and obviously our code block was called for *all* text nodes, i.e. even for those containing only whitespace characters.

REXML's handling of all things related to DTDs is rudimentary and you have to do a lot of things yourself that other parsers won't expect you to do. It's a bit annoying, but at least for the XML standard entities we only have to perform simple text substitutions.

To solve our whitespace problem, we could check if the text that is passed to the code block is empty. But wouldn't it be much nicer if we could tell the parser to call our code block only for `<to-do>` elements? Here we go:

File

```

Line 1  def decode_markup(text)
-      text.gsub!(/&lt;/, '<')
-      text.gsub!(/&gt;/, '>')
-      text.gsub!(/&apos;/, "'")
5      text.gsub!(/&quot;/, '"')
-      text.gsub!(/&amp;/, '&')
-      text
-      end
-
10  todo_list = File.new('todo.xml')
-  parser = REXML::Parsers::SAX2Parser.new(todo_list)
-  parser.listen(:characters, ['to-do']) { |text|
-      puts " * #{decode_markup(text)} "
-  }
15  parser.parse

```

produces:

```

* Write a book.
* Learn a new programming language.
* Don't Repeat Yourself

```

As well as the event to listen for, the `listen()` method can take a list

of element names. It's even possible to pass regular expressions that describe the element names to be matched against.

In the following descriptions of the variants of the `listen()` method, the parameters have the following meaning:

- `symbol` can be one of: `:start_element`, `:end_element`, `:characters`, `:cdata`, `:start_prefix_mapping`, `:end_prefix_mapping`, `:processing_instruction`, `:doctype`, `:attlistdecl`, `:elementdecl`, `:entitydecl`, `:notationdecl`, `:xmldecl`, `:comment`.
- `array` contains regular expressions or strings which will be matched against fully qualified element names.
- `listener` is an object implementing all methods of the `REXML::SAX2Listener` class.
- `block` will be passed the same arguments as the according `REXML::SAX2Listener` method would get. The method name is the same as the matched symbol.

You can invoke `listen()` as follows:

`listen(symbol, array, &block)`

Listens to the symbol event for all elements in array. For example,

```
parser.listen(:characters, %w(name surname)) do |text|
  puts text
end
```

prints the text of all `<name>` and `<surname>` elements.

`listen(symbol, &block)`

Calls block whenever the symbol event occurs. For example,

```
parser.listen(:comment) { |text| puts text }
```

prints all comments in a document.

`listen(array, listener)`

Invokes methods of listener only for elements whose name is in array. For example,

```
class MyListener
  include REXML::SAX2Listener
  ...
end

parser.listen(%w(to-do), MyListener.new)
```

invokes listener methods only for `<to-do>` elements.

```
listen(array, &block)
```

block is called when the :start\_element event occurs and the name of the current element is in array. For example,

```
parser.listen(%w(order person)) do |uri, localname, qname, attr|
  attr.each { |k, v| puts "#{k}=#{v}" }
end
```

prints all attributes of all <order> and <person> elements.

```
listen(listener)
```

Listen to all events and invoke the according methods of the listener.

```
class MyListener
  include REXML::SAX2Listener
  ...
end
```

```
parser.listen(MyListener.new)
```

That's the standard way of SAX2 parsing.

## XPath

In Section 3.3, *Tree Parsing*, on page 93, we saw that the each\_element() method of class REXML::Element accepts an element name or a so-called XPath expression. According to the official specification<sup>7</sup> “XPath is a language for addressing parts of an XML document, designed to be used by both XSLT and XPointer.” And that’s exactly how it is: with XPath you can extract single nodes and node sets matching certain criteria from an XML document. Simply put, XPath is for XML documents what regular expressions are for strings.

XPath expressions are not XML documents or document fragments themselves: they have their own syntax. We will not fully cover XPath and all its nitty-gritty details, but we will at least show you enough of its syntax to move conveniently through typical XML documents using REXML (for a complete reference see *XPath and XPointer* [?]).

Besides XPath support in the Element class, REXML has a class called XPath. This class contains three methods that allow you to select arbitrary node sets from a certain element:<sup>8</sup>

<sup>7</sup><http://www.w3.org/TR/xpath>

<sup>8</sup>These methods potentially deal with objects of different classes. For example, match() may return objects of class Element, Attribute, etc.

`each(element, xpath=nil, namespaces={})`  
 iterates over all nodes matching `xpath` in the context of `element`.  
`xpath` defaults to `'*'`. The `namespaces Hash` may contain a namespace mapping.

`first(element, xpath=nil, namespaces={})`  
 returns the first node matching `xpath` in the context of `element`.  
`xpath` defaults to `'*'`. iterates over all nodes matching `xpath` in the context of `element`. `xpath` defaults to `'*'`.

`match(element, xpath=nil, namespaces={})`  
 returns all nodes matching `xpath` in the context of `element`. `xpath` defaults to `'*'`.

In the following shortened version of our tracking history document from Section 3.3, *Processing XML Documents*, on page 91, we use these methods to demonstrate how REXML's XPath support works.

File

```
<tracking-result>
  <parcel-history tracking-no='2X42'>
    <event timestamp='2005-05-02T04:05:00'
      state='picked-up' />
    <event timestamp='2005-05-02T08:30:00'
      state='first-delivery-attempt' />
    <event timestamp='2005-05-04T10:15:00'
      state='refused-damaged'>
      <consignee>Mrs. Smith</consignee>
    </event>
    <event timestamp='2005-05-04T19:07:00' state='returns-to-sender' />
  </parcel-history>

  <parcel-history tracking-no='2X43'>
    <event timestamp='2005-05-02T04:25:00' state='picked-up' />
    <event timestamp='2005-05-02T09:15:00' state='delivered'>
      <consignee>Mr. Gumble</consignee>
    </event>
  </parcel-history>
</tracking-result>
```

We assume that the `doc` variable has been initialized like this:

File

```
doc = Document.new(File.new('packages.xml'))
```

Let's start with some simple examples:

File

```
puts XPath.match(doc, '/tracking-result/parcel-history')[1]
```

produces:

```
<parcel-history tracking-no='2X43'>
  <event timestamp='2005-05-02T04:25:00' state='picked-up' />
```

```

    <event timestamp='2005-05-02T09:15:00' state='delivered'>
      <consignee>Mr. Gumble</consignee>
    </event>
  </parcel-history>

```

XPath expressions are made up of *location paths* and *location steps*. In the example above, the location path is `/tracking-result/package-history`. It consists of two location steps, namely `/tracking-result` and `package-history`.

*location paths*

*location steps*

As you might have guessed, location steps are separated by a slash, but the leading slash in the first location step is not a delimiter. Instead, it identifies the document root. Hence, the first location step identifies the `<tracking-result>` child of the document root—the root element. The second location step identifies all `<parcel-history>` children of the preceding element (the `<tracking-result>` element). Because there are three of them, `match()` returns them all as an array. We select the second one.

We can achieve the same result a bit differently by using XPath's index features. Node sets can be treated as arrays and can be indexed using square brackets:

File

```
puts XPath.first(doc, '/tracking-result/parcel-history[2]')
```

Did you notice that we used the index 2? It's true: XPath indices start at 1, not 0! That's why the indexing of the elements array of class `REXML::elements` starts at 1, too.

It's also possible to define relative paths: they do not start with a slash. A more or less obscure way to get all the consignees belonging to delivered packages could look like this:

File

```

events = XPath.match(
  doc,
  '/tracking-result/parcel-history/event[@state="delivered"]'
)
events.each do |event|
  puts XPath.match(event, 'consignee')
end

```

It produces:

```
<consignee>Mr. Gumble</consignee>
```

XPath expressions are similar to file and directory paths under Unix, but this analogy cannot be stretched too far. Even in our small example we have encountered the first big difference: every element may have an arbitrary number of children having the same name. For example, a `<tracking-result>` element may have more than one `<package-history>`

child element (which is impossible in file systems). The next big difference is that we can recursively select nodes. For example, the following statement

File

```
puts XPath.match(doc, '//consignee')
```

produces:

```
<consignee>Mrs. Smith</consignee>
<consignee>Mr. Gumble</consignee>
```

`//consignee` selects all `<consignee>` nodes in the document. The expression `'//'` is an abbreviation for the *descendant-or-self axis*. This contains all descendant nodes (not only direct children) belonging to a particular node, along with the node itself. At the beginning of a location path, `//` means “select all nodes in the document” and we refine that by adding the consignee location step.

*descendant-or-self axis*

The concepts of context and axes are important when working with XPath, because XPath interpreters see XML documents as trees where every node has a type and lives in a particular context that defines a relationship to all other nodes in the tree. For example, our tracking result document looks to an XPath interpreter like Figure 3.4, on the following page. It is a tree consisting of 30 nodes (actually, it has many more because of all the whitespace text nodes belonging to nearly every element, but we left them out for brevity). Twelve of them are element nodes, sixteen are attribute nodes and two are text nodes.

Given a particular node, you can easily define node sets that are somehow related to this node. For an element node, for example, you can create a set consisting of all its children, all its attributes, all its descendants, all its siblings, and so on. XPath defines some standard ways to create node sets from a given context node and calls them *axes* (you can see them in Figure 3.5, on page 116). To make things more complicated, all axes can not only select nodes by their position, but also by their type. For example, the descendant-or-self axis we used above selects all nodes below the current context node, but only if they are not attribute nodes or namespace nodes.

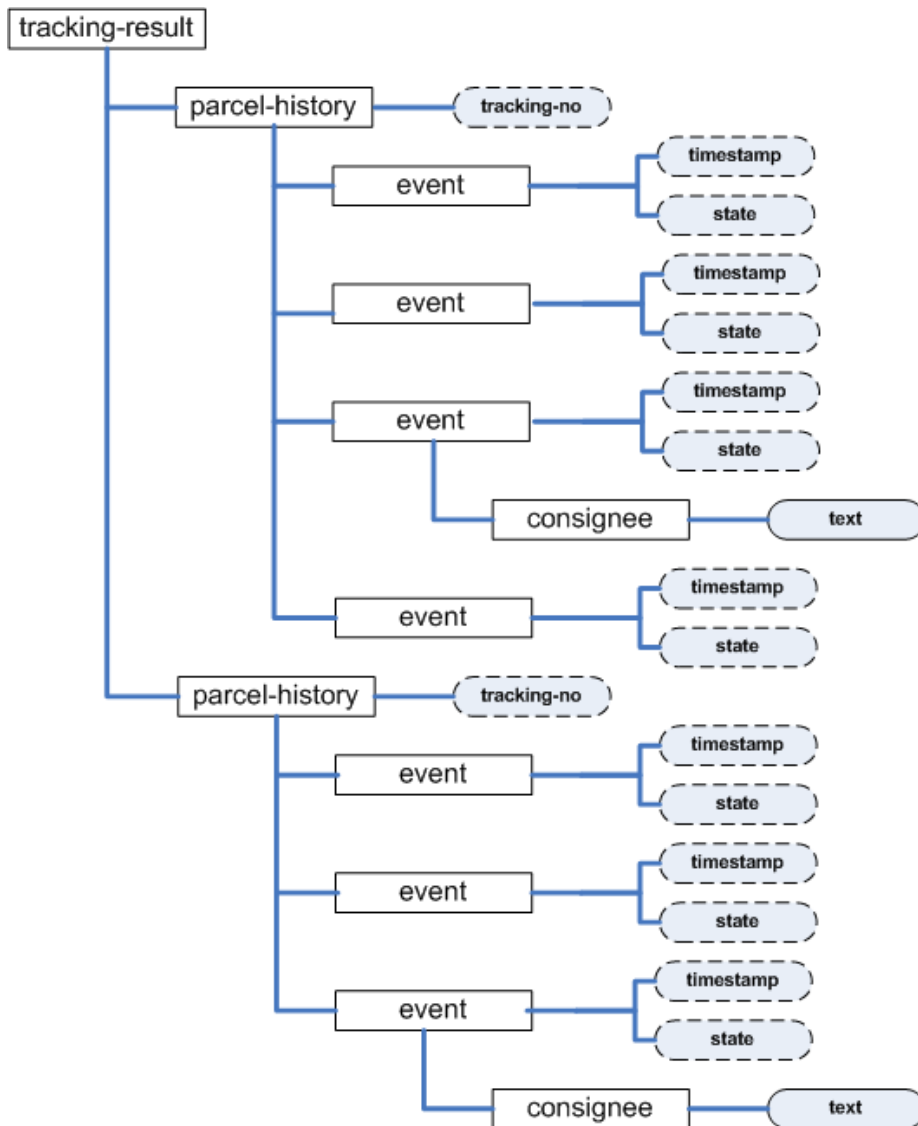
We could have selected all consignees more verbosely without using the `//` abbreviation:

File

```
puts XPath.match(doc, 'descendant-or-self::consignee')
```

It works the same with all other axes. For example, the following two statements select all tracking numbers:






---

Figure 3.4: Tree Representation of Tracking Results

---

|                                     |                                                                                                                                                                                                                                 |
|-------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| child                               | <p>All immediate children of the context node.</p> <p>Selects: element, processing instruction, comment, text</p>                                                                                                               |
| descendant                          | <p>All descendants of the context node, i.e. all children, grandchildren, and so on.</p> <p>Selects: element, processing instruction, comment, text</p>                                                                         |
| parent                              | <p>All nodes immediately above the context node. It may include the root node. Abbreviation: ..</p> <p>Selects: element</p>                                                                                                     |
| ancestor                            | <p>All ancestors of the context node, i.e. its parents, grandparents, and so on. Includes the root node.</p> <p>Selects: element</p>                                                                                            |
| preceding-sibling/following-sibling | <p>All siblings preceding (following) the context node having the same parent. Empty for attribute and namespace nodes. Never includes the root node.</p> <p>Selects: Any but attribute and namespace.</p>                      |
| preceding/following                 | <p>All nodes preceding (following) the context node excluding any ancestors (descendants) and excluding attribute nodes and namespace nodes. Never includes the root node.</p> <p>Selects: Any but attribute and namespace.</p> |
| attribute                           | <p>All attribute nodes of the context node. Abbreviation: @</p>                                                                                                                                                                 |
| namespace                           | <p>All namespace nodes of the context node.</p>                                                                                                                                                                                 |
| self                                | <p>The context node itself. Abbreviation: .</p>                                                                                                                                                                                 |
| descendant-or-self                  | <p>The context node and all its descendants. Abbreviation: //</p> <p>Selects: Any but attribute and namespace.</p>                                                                                                              |
| ancestor-or-self                    | <p>The context node and all its ancestors. Includes the root node.</p> <p>Selects: element</p>                                                                                                                                  |

---

Figure 3.5: XPath Axes

---

```
File puts XPath.match(doc, '//parcel-history/@tracking-no')
```

```
File puts XPath.match(doc, '//parcel-history/attribute::tracking-no')
```

Now we know how to access each node and some particular interesting neighbors of each of them. XPath is concise and elegant in many respects, but we could have achieved this using iterators and loops, too.

But XPath can do more for us, i.e. it allows us to use conditions—XPath calls them *predicates*—that refine node sets even further. For example, the following statement locates all events belonging to package 2X43:

```
File puts XPath.match(doc, '//parcel-history[@tracking-no="2X43"]/*')
```

and produces:

```
<event timestamp='2005-05-02T04:25:00' state='picked-up' />
<event timestamp='2005-05-02T09:15:00' state='delivered'>
  <consignee>Mr. Gumble</consignee>
</event>
```

Here we have selected nodes anonymously—without knowing their names—by using the asterisk (\*). It stands for “select every node you can find.” Predicates are put in square brackets and can occur after any location step. They are boolean expressions. Whenever a predicate evaluates to true the corresponding node becomes part of the result node set. XPath boolean expressions are usually of the form `lvalue op rvalue`, where `op` is one of the operators shown in Figure 3.6, on the following page.

Predicates can be stacked by chaining them together (with no separator). For example, the following statement selects all `<event>` elements that have been delivered on 2005-05-02.

```
File puts XPath.match(
  doc,
  '//event[starts-with(@timestamp, "2005-05-02")][@state="delivered"]'
)
```

produces:

```
<event timestamp='2005-05-02T09:15:00' state='delivered'>
  <consignee>Mr. Gumble</consignee>
</event>
```

But wait: what about that first predicate? Apparently, it's a boolean expression, but it does not contain an operator. As you might have guessed already, XPath defines functions, too, and `starts-with(string,substring)` is one of them. If `string` actually starts with `substring`, this method returns true, otherwise it returns false.

| Operator | Meaning               |
|----------|-----------------------|
| =        | Equal to              |
| !=       | Not equal to          |
| >        | Greater than          |
| >=       | Greater than or equal |
| <        | Less than             |
| <=       | Less than or equal    |

---

Figure 3.6: XPath Boolean Operators

---

XPath functions are divided into four groups:<sup>9</sup>

- *Node set functions* let you perform operations on node sets. For example, the `count(node-set)` function returns the number of nodes in the argument node-set.
- *String functions* manipulate and compare strings. For instance, the `normalize-space((string))` function returns a normalized representation of the argument string (it defaults to the current context node). It removes leading and trailing whitespace characters and replaces sequences of multiple whitespace characters by a single space.
- *Boolean functions* implement common predicates. For example, `not(expr)` returns true, if `expr` is false. Otherwise, it returns false.
- *Number functions* help you work with numeric expressions. The `sum(node-set)` function, for example, converts the string values of all nodes belonging to node-set into numbers and returns their overall sum.

---

<sup>9</sup>You can find a description of all functions under <http://www.w3.org/TR/xpath##corelib>.

REXML::Element has one more interesting method, `xpath()`. It returns one XPath belonging to a particular element (there's a nearly infinite number of XPaths leading to every XML document node). For example, the following statement

File

```
event = XPath.first(
  doc,
  '/tracking-result/parcel-history/event[@state="delivered"]'
)
puts event.xpath
```

produces:

```
/tracking-result/parcel-history[2]/event[2]
```

REXML's XPath support is comparatively good, but its implementation of the standard functions and some of the more complex features of the XPath specification is still a work in progress. So, before depending on a particular function, you'd better test REXML's support for it.

For example, the union operator does not work as expected. In XPath it's possible to join node sets using the `|` operator. The following expression should return all `state=` and all `tracking-no=` attributes:

File

```
puts XPath.match(doc, '//event/@state | //parcel-history/@tracking-no')
puts XPath.match(
  doc,
  '//event[@state="delivered"]|//event[@state="picked-up"]'
)
```

Currently, REXML only returns the node set on the left side of the `|`-operator. The previous code prints

```
picked-up
first-delivery-attempt
refused-damaged
returns-to-sender
picked-up
delivered
```

As with stored procedures in databases it's usually not a good idea to put too much of your business logic into XPath expressions (even simple length constraints can make up a vital part of your business logic). On the other hand, XPath is probably the most convenient way to access content of XML documents and the REXML way of life will certainly change the way you approach your next parsing problem.

## XML Processing With XmlSimple

The attentive reader (and that's all of you, isn't it?) might have noticed that processing XML is often a tedious job, mapping elements to classes and attributes to member variables. Usually, the classes have the same name as their corresponding XML elements and the member variables have the same names as their attribute counterparts. This calls for automation. Fortunately, a lot of tools exist to free you from this type of repetitive work.

Depending on the programming environment you're working in, you can choose from a variety of tools today ranging from small helper libraries supporting your own serialization strategy to sophisticated solutions that automatically create complete class hierarchies and serializer classes based on XmlSchema files. Java and .NET certainly have the most mature support for these kind of things, but there are some useful libraries for Ruby, too.

Grant McLean got fed up writing the same code over and over again and came up with a clever solution. His Perl module XML::Simple<sup>10</sup> converts an XML document into a structure of hashes and arrays according to some simple rules and conventions. I often used this model back in the old days when Ruby was nearly unknown so it seemed natural to port it across when I started using Ruby. The Ruby version, named *XmlSimple*,<sup>11</sup> implements nearly 100% of the original's functionality and fixes some of its major flaws.

Less talk, more examples! Here we have a configuration file called `app-config.xml`, as found in countless projects the world over:

File

```
<?xml version="1.0" encoding="iso-8859-1"?>
```

```
<app-config>
  <database env='test'>
    <usr>developer</usr>
    <pwd>foo</pwd>
  </database>

  <database env='production'>
    <usr>admin</usr>
    <pwd>secret</pwd>
  </database>
```

<sup>10</sup><http://www.cpan.org/modules/by-module/XML/XML-Simple-2.14.readme>

<sup>11</sup>It is available as a gem called `xml-simple` and from <http://raa.ruby-lang.org/project/xml-simple>

```

<http-proxy>
  <host>my.proxy.host</host>
  <port>8080</port>
</http-proxy>
</app-config>

```

Configuration files like this are often read when the application starts. Their content is then made available globally using a singleton class or something similar. It would be a complete overkill to deserialize its content manually and to create a corresponding class hierarchy. A hash containing all configuration parameters would be sufficient. To achieve this with XmlSimple, do something like the following.

File

```

require 'xmlsimple'

cfg = XmlSimple::xml_in('appconfig.xml')
p cfg

```

This produces:

```

{
  "http-proxy" => [
    {
      "port" => ["8080"],
      "host" => ["my.proxy.host"]
    }
  ],
  "database" => [
    {
      "env" => "test",
      "pwd" => ["foo"],
      "usr" => ["developer"]
    },
    {
      "env" => "production",
      "pwd" => ["secret"],
      "usr" => ["admin"]
    }
  ]
}

```

XmlSimple turns the XML document into a structure consisting of hashes and arrays, where every element and attribute name becomes a hash key pointing to its associated content. Because elements can potentially occur more than once, their values are stored in arrays by default, while attribute values are not.

Our first result looks promising, but it's far from being optimal. To find the password of our production database, for example, we have to

search for an entry in the array `cfg['database']`, where the value associated with the key `env` is “production”.

To adjust the output’s structure, `xml_in()` accepts an optional hash containing `name => value` option pairs. The option `key_attr` controls which elements should be used as the basis for the array folding. We’ll set it to “env”:

File

```
cfg = XmlSimple::xml_in('appconfig.xml', 'key_attr' => 'env')
p cfg
```

Now we get:

```
{
  "http-proxy" => [
    {
      "port" => ["8080"],
      "host" => ["my.proxy.host"]
    }
  ],
  "database" => {                                # -> No longer an Array.
    "production" => {
      "pwd" => ["secret"],
      "usr" => ["admin"]
    },
    "test" => {
      "pwd" => ["foo"],
      "usr" => ["developer"]
    }
  }
}
```

That’s much better. Now we can access the production database’s password using

```
cfg['database']['production']['pwd'][0]
```

The arrays containing a single element are still a bit annoying, but we can get rid of them by setting the parameter `force_array` to `false`.

File

```
cfg = XmlSimple::xml_in(nil, 'key_attr' => 'env', 'force_array' => false)
p cfg
```

The output is simpler:

```
{
  "http-proxy" => {
    "port" => "8080",
    "host" => "my.proxy.host"
  },
  "database" => {
    "production" => {
      "pwd" => "secret",

```



```

    "usr" => "admin"
  },
  "test" => {
    "pwd" => "foo",
    "usr" => "developer"
  }
}
}

```

Now the database password can be found in `cfg['database']['production']['pwd']`.

As we saw, `xml_in()` expects some kind of XML source and a parameter hash. The XML source can be:

- *a filename*, as in

```
cfg = XmlSimple.xml_in('appconfig.xml')
```

- `nil`

If there is no XML source, `xml_in()` will check the source program's directory for a file with the same name as the script but with the extension `.xml`. If you wish to specify options, you must specify the value `nil`:

```
cfg = XmlSimple.xml_in(nil, 'force_array' => false)
```

- *a string containing XML* (recognized by the presence of `<` and `>` characters) will be parsed directly. For example:

```
cfg = XmlSimple.xml_in('<cfg env="test" password="foo" />')
```

- *an IO object* will be read and its contents parsed. For example:

```
cfg = XmlSimple.xml_in(File.new('appconfig.xml'))
```

`XmlSimple` is not a full-blown XML data binding tool, but it does just what you need surprisingly often.

### 3.4 Validating XML Documents

Because you have to pay for every request to the e-scoring application even when it's syntactically invalid, you decide to validate all XML documents you're going to send before actually transmitting them (all documents we create using REXML are wellformed, but they do not have to be valid necessarily). In general, you could choose from various validation technologies, using DTDs (Document Type Definition), `XmlSchema`, and RELAX NG (Regular Language for XML, New Gener-

ation)<sup>12</sup>. Fortunately, the decision with Ruby is easy, because REXML only supports RELAX NG. That's good news, because RELAX NG is the most powerful and most convenient of the three. The bad news is that REXML's support for RELAX NG is highly experimental and a lot of important features do not work at all.

RELAX NG does not allow you to control the lexical appearance of a document, so you can't use it to control what kind of quotes you've used for attributes, or the comments or processing instructions that can be used.

RELAX NG comes in two different syntax styles: XML syntax and compact syntax. Both have their pros and cons, but again the choice is made easy for Ruby programmers, because REXML does not support the compact syntax style. To be honest: at the current stage of development, REXML's support for RELAX NG is barely useful, but it's still possible to perform some basic validation, if you obey some simple rules.

Before diving into the details of RELAX NG, let's revisit our example document from Section 3.2, *Generating XML documents*, on page 79.

File

```
<persons>
  <person name='Max' surname='Mustermann'>
    <address>
      <street>Musterstr. 42</street>
      <city>Berlin</city>
      <postal-code>11011</postal-code>
    </address>
  </person>
</persons>
```

Considering only a document's structure and not its content, a valid document fulfills the following requirements:

- There's exactly one root element called `<persons>`.
- The `<persons>` element may have one or more `<person>` children.
- A `<person>` element has exactly two attributes called `name` and `surname`.
- A `<person>` element has one `<address>` child.
- Every `<address>` element has a `<street>`, a `<city>`, and a `<postal-code>` child.

<sup>12</sup><http://www.oasis-open.org/committees/relax-ng>

- The `<street>`, `<city>`, and `<postal-code>` elements may contain text.

RELAX NG allows us to translate these requirements easily into the XML syntax:

```
<?xml version='1.0'?>

<element xmlns='http://relaxng.org/ns/structure/1.0' name='persons'>
  <oneOrMore>
    <element name='person'>
      <attribute name='name'/>
      <attribute name='surname'/>
      <element name='address'>
        <element name='street'><text/></element>
        <element name='city'><text/></element>
        <element name='postal-code'><text/></element>
      </element>
    </element>
  </oneOrMore>
</element>
```

To validate our example document against this schema, we do the following:

```
Line 1  require 'rexml/document'
-       require 'rexml/validation/relaxng'
-       include REXML
-
5       def read_doc(filename)
-       doc = IO.readlines(filename).to_s
-       doc.gsub!(/\s+</, '<')
-       doc.gsub!(/\s+$/ , "")
-       doc
10      end
-
-       begin
-       schema = File.new('person_schema.xml')
-       validator = Validation::RelaxNG.new(schema)
15      parser = Parsers::TreeParser.new(read_doc('persons.xml'))
-       parser.add_listener(validator.reset)
-       parser.parse
-       rescue Validation::ValidationException => e
-       puts "An error occurred during validation: #{e}"
20      rescue Exception => e
-       puts "An exception occurred: #{e}"
-       end
```

As we don't need to process the document any further, it doesn't matter which parser we use for our validation purposes and we arbitrarily chose the `TreeParser`. Most of the code should be self-explanatory: in

lines 14 to 17 we initialize a validator with our schema and pass it to the parser. If anything goes wrong while validating the document, a `ValidationException` exception will be thrown.

But what the heck does `read_doc()` do? As mentioned earlier, REXML's support for RELAX NG validation is full of flaws. One of them is the incorrect handling of whitespace. Whitespace handling has always been a complicated issue in XML. When validating documents, it becomes an even bigger problem. In a formatted XML document, the indentation before tags automatically is converted into text nodes. So, even if your element does not allow for text, it actually has some, if you format your document. The corresponding schema must support it too.

RELAX NG usually works pragmatically with this issue, as it considers whitespace and empty content equal—you do not have to clutter your schema document with `<text>` elements only because you want to lay out your source with white space. REXML currently does not implement this behavior, so instead of adding countless `<text>` elements, `read_doc()` eliminates all whitespace before `'<'` characters. Another annoying bug is REXML's comment handling. There's no way in RELAX NG to restrict comments in any way, but if you have a comment in an XML document you're trying to validate with REXML it will fail miserably, because REXML complains about the unexpected comment node. Finally, REXML does not support the XML header.

## xmllint

If you definitely need to validate XML documents against a particular schema, you can use `xmllint`.<sup>13</sup> `xmllint` is a command line tool that is available for nearly all operating systems. It checks that a given set of XML files are well-formed and if they are in compliance with a certain schema. The following class encapsulates access to the `xmllint` program. If there is native schema support someday in Ruby, you'll be able to replace the `system()` call easily.

File

```
Line 1  class Validator
-      class << self
-          def validate(document, schema)
-              option = case schema
5          when /\.dtd$/ then '-valid'
-          when /\.xml$/ then '-relaxng'
-          when /\.xsd$/ then '-schema'
```

<sup>13</sup><http://xmlsoft.org/xmllint>

```

-         end
-         exec_xmllint(document, schema, option)
10      end
-
-      private
-
-      def exec_xmllint(file, schema, option)
15          rc = system("xmllint -noout #{option} #{schema} #{file}")
-          status = $? >> 8
-          return true if rc && status == 0
-          raise 'ValidationError' if [3, 4].include?(status)
-          raise 'UnexpectedError' unless rc
20      end
-      end
-  end

```

This solution isn't particularly efficient, but it's good enough in many cases. `xmllint`'s validation support is excellent. From line 4 to 8 we determine which options have to be passed to `xmllint`. In line 15 we call `xmllint` and evaluate its return status. First we check if the `system()` call succeeded—whether it returned `true` or `false` (it will return `false`, for example, if `xmllint` could not be found). Then we determine `xmllint`'s return code. `xmllint` returns 0 if it could parse and validate a document successfully. If a validation error occurred, it returns 3 or 4. All other return codes indicate errors.

To validate our sample document you have to do this:

File

```

Line 1  begin
-      Validator.validate('persons.xml', 'person_schema.xml')
-  rescue => ex
-      puts ex
5  end

```

## Conclusion

Validation of XML documents using any kind of schema is not as useful as it might initially seem; as a careful and defensive programmer you will check the restrictions defined in the schema in your program anyway. Validation only makes sense when you generate the code completely from a schema or when you generate documents that are going to be consumed by other applications.

Ruby's support for any kind of XML schema language is currently weak. If you are in need of high performance XML validation, you will have to look for alternatives or—even better—implement a `libxml2` binding for Ruby.

### 3.5 Are There Alternatives to XML?

Data representation is one of the most important things in our business; you shouldn't take it lightly. Sometimes you do not have a choice. If you have to deal with legacy data, you usually have to eat what has been served.

Although XML is a comparatively young technology, for some people it seems hard to remember the dark and ancient times without XML. Today nearly every program offers functions for importing or exporting XML or can be configured by editing one or more XML files.

Many developers do not think about alternative file formats anymore. "XML is a standard and I get a parser for free, so why should I use something else or even invent something new?", they say. Certainly this is true under some circumstances, but more often than not you're better off, if you think about alternatives.

As a rule of thumb you should try to avoid XML, when you know that human beings have to read or edit the documents. It's OK for machines to do it, because that's what they were built for, but you do not want your customers to learn the secrets of wellformedness. Even if your customers are programmers, they will be deeply grateful, if you offer them the most convenient way to achieve their goals.

You should not use XML only because you get a parser for free. James Duncan Davidson did this when he created the Java build tool Ant and today he publicly regrets having done it <sup>14</sup>, because looking back XML wasn't an adequate choice. It does not fit the needs of a build tool and we'll see a much better approach in Section 6.5, *Project Automation with Rake*, on page 293.

XML is quite a misnomer, because it is not a language itself but a set of rules for describing markup languages. You should never consider using XML for the description of a programming language. Probably the most obvious example of this kind of misuse is the eXtensible Stylesheet Language (XSL) <sup>15</sup>.

Despite all this XML should be part of every developer's toolbox, because it is widespread and can be handy under certain circumstances, but you should always keep in mind: There's no such thing like "cool tech-

---

<sup>14</sup><http://web.archive.org/web/20040602210721/x180.net/Articles/Java/AntAndXML.html>

<sup>15</sup><http://www.w3.org/Style/XSL>

nology”. There are only tools that help you to get your job done, and there are tools that don’t.

## Comma-Separated Values (CSV)

Although there is no official standard, the *Comma-Separated Values* (CSV) format may well be the most popular data exchange format ever<sup>16</sup>, because countless spreadsheet and database applications use it for importing and exporting data.

*Comma-Separated  
Values*

Simply put, a CSV file is a plain text file containing tabular data where every line represents a table row and where all columns (fields) are separated by a delimiter. The lines themselves are usually separated by a carriage return. If a column contains the delimiting character or a carriage return, quotes will be put before and after its value. If the value contains quotes, too, the quotes will be doubled. Optionally the first line of the file may contain the column names (separated by the delimiter, too).

Here we have a CSV representation of some bunches of flowers:

File

```
id,name,price,description
1,Red Dream,79.99,"Lots of red roses saying ""Come back, please!""
2,Blue Velvet,29.99,Your Mom will love it.
```

Because the description of the “Red Dream” data set contains a comma, we had to set it in quotes. Consequently, we had to double all the quotes in the description, too.

The name Comma-Separated Values is quite a misnomer, because today it’s used for all kind of formats where a set of attributes is separated by special characters. For every file you have to be consistent about the delimiter, i.e. all attributes in every line have to be separated by the same character. Frequently used delimiters are ;, :, |, blanks, and tabulators. Hence, pedantic people call the format *Character-Separated Values* nowadays.

*Character-Separated  
Values*

You might be tempted to create and read CSV files using strings and regular expressions, but beware: it’s not as easy as it seems and there are a lot of pitfalls lurking. In addition, Hiroshi Nakamura’s excellent CSV library in the meantime became part of the standard distribution and makes both reading and writing CSV data a breeze.

<sup>16</sup>Under [http://en.wikipedia.org/wiki/Comma-separated\\_values](http://en.wikipedia.org/wiki/Comma-separated_values) you’ll find a lot of interesting background information.

## Generating Comma-Separated Values

For generating CSV data we use the `generate(stream,fs=',',rs=nil, &block)` method of class `CSV::Writer`. Parameter `stream` is an object supporting the `<<(string)`-operator, `fs` is the field separator to be used (comma by default), `rs` is the record separator to be used (carriage return by default), and `block` is a code block to be called for every line to be generated. The code block gets a single parameter of class `CSV::BasicWriter`.

To generate our list of flowers using a semicolon as delimiter, we have to do the following:

```
File
Line 1  require 'csv'
-      CSV::Writer.generate(STDOUT, ';') do |csv|
-        csv << [1, 'Red Dream', 'Red roses saying "Come back, please!"]
-        csv << [2, 'Blue Velvet', 'Your Mom will love it.']
5      end
```

produces:

```
1;Red Dream;"Red roses saying ""Come back, please!""
2;Blue Velvet;Your Mom will love it.
```

Did you notice that `CSV::Writer` enclosed our “Red Dream” description in quotes, although it does not contain the delimiter? Always keep in mind: There is no CSV standard and whenever you have to create or process CSV files, you should talk to the sender or recipient of the files upfront. Maybe Hiroshi’s library does not emit what your counterpart expects or vice versa.

## Processing Comma-Separated Values

You already guessed it: the CSV library contains a `CSV::Reader` class for reading CSV files. The following program produces a price list of our flavorsome products:

```
File
Line 1  require 'csv'
-      CSV::Reader.parse(File.open('products.csv', 'r')) do |row|
-        puts "#{row[1]}: #{row[2]}"
-      end
```

produces:

```
name: price
Red Dream: 79.99
Blue Velvet: 29.99
```

Obviously we forgot to ignore our header line and the `parse(stream,fs=',',rs=nil, &block)` method of class `CSV::Reader` does not offer a possibility to ignore



it. Hence, we will use `open(path, mode, fs=nil, rs=nil, &block)` to get an instance of `CSV::Reader`:

File

```
Line 1  csv = CSV.open('products.csv', 'r')
-      header = csv.shift # Ignore header line.
-      csv.each { |row| puts "#{row[1]}: #{row[2]}" }
-      csv.close
```

produces:

Red Dream: 79.99

Blue Velvet: 29.99

Normally, you would not ignore the header, but use it to dynamically create objects or reports. The following program dynamically creates a list of `Product` classes from our CSV file:

File

```
Line 1  require 'csv'
-      csv = CSV.open('products.csv', 'r')
-      Product = Struct.new(*(csv.shift.map { |f| f.to_sym }))
-      products = csv.inject([]) do |products, row|
5         products << Product[*row]
-      end
-      csv.close
-
-      products.each do |p|
10         puts "#{p.name}: #{p.price}"
-      end
```

produces:

Red Dream: 79.99

Blue Velvet: 29.99

That's the Ruby way of life and nearly all the magic comes from line 3, where we create a new `Product` class on the fly by using class `Struct`. Let's dissect this line from the inside out: `csv.shift.map { |f| f.to_sym }` reads the header line from our CSV file and turns all columns into a symbol by calling `to_sym()`. The result is an array of symbols that is converted into a parameter list using the asterisk. This parameter list is then used to create a new `Struct` called `Product`. Our newly created class has accessors for all the header columns (`id`, `name`, `price`, `description`) and new instances can be created using the `new()` or `[]`-operator (as shown in line 5).

Skimming the documentation of the CSV library is not a bad idea, because it contains some convenient methods (like `foreach()` and `read-line()`) we didn't cover here.

## Properties Files

*Properties files* can be often found in Java environments and it happens sometimes that you have to add a program to an existing enterprise Java project that has to use the same configuration as the project. For example, the HTTP proxy and database parameters are typically the same for many applications running on the same machine.

*Properties files*

Simply put, a property is a value associated with a unique key and a properties file is a file containing such key/value pairs. Keys and values are separated by “=” or “:” characters and logical lines can comprise more than one physical line. A typical properties file looks like this:

File

```
! Comments start with '!'...
# ... or with '#'.

ruby = pragmatic

# A property spanning two lines.
broken_value = This value is \
                long.

colon_prop: hello

# A property spanning three lines.
fruits:  apple, banana, pear, \
         cantaloupe, watermelon, \
         kiwi, mango

http.proxyHost=example.com
http.proxyPort=8080
```

Although the original specification is much more complicated (e.g., it allows “:” and “=” characters in keys and Unicode characters), the following class should understand enough of the Java properties syntax to process most properties files out there.

File

```
Line 1  class PropertiesFile
-       def initialize(file_name)
-         @properties = load_props(file_name)
-       end
5
-       def [](key) @properties[key]; end
-       def to_hash() @properties; end
-
-       private
10
-       def load_props(file_name)
-         properties = {}
-         file = File.new(file_name)
```

```

-       while !file.eof? do
15         line = get_line(file)
-         next if line =~ /^[#!]/ or line =~ /\s*$/
-         if line =~ /\s*(\S+)\s*[:]\s*(.*)$/
-           properties[$1] = $2
-         end
20       end
-       file.close
-       properties
-     end
-
25     def get_line(file)
-       current_line = file.readline.chomp
-       while current_line =~ /^(.*)\\$/
-         next_line = file.readline.chomp
-         current_line = $1 + next_line.gsub(/\s*(.*)$/, '\\\1')
30       end
-       current_line
-     end
-   end
end

```

Feeding our sample file to our class produces:

```

require 'propfile'
properties = PropertiesFile.new('regular.properties')
properties.to_hash.each do |k,v|
  puts "#{k}: #{v}"
end

```

```

broken_value: This value is long.
http.proxyPort: 8080
fruits: apple, banana, pear, cantaloupe, watermelon, kiwi, mango
colon_prop: hello
ruby: pragmatic
http.proxyHost: example.com

```

## Fixed Length Records

One of the oldest and simplest file formats in the history of computer science is fixed length records, where every line of a file is divided into several fields having a fixed length. A fixed length record file containing a credit card number (19 characters), a first name (20 characters) and a surname (30 characters) looks like this:

File

```

Line 1  1234-5678-9012-3456Barney          Gumble
-       1111-2222-3333-4444Homer         Simpson

```

Even in times of seemingly unlimited resources we still have to think about maximum sizes surprisingly often. For instance, relational databases still force us to define the maximum length of columns (How long can a surname or a street name be? In North America? In Iceland?)

and therefore database exports are often files containing fixed length records. But using fixed length records can also be a natural choice, because the size of a lot of data, e.g. social security or credit card numbers, is limited anyway.

Hence, files containing fixed length records are still in widespread use today and processing them with Ruby is easy:

File

```
class FixedLengthRecordFile
  def FixedLengthRecordFile.open(filename, field_sizes)
    field_pattern = 'a' + field_sizes.join('a')
    IO.foreach(filename) do |line|
      record = line.chomp.unpack(field_pattern)
      record.map { |f| f.strip! }
      yield record
    end
  end
end
```

File

```
FixedLengthRecordFile.open('customers.fix', [19, 20, 30]) do |row|
  puts "#{row[1]}, #{row[2]} (#{row[0]})"
end
```

produces:

```
Barney, Gumble (1234-5678-9012-3456)
Homer, Simpson (1111-2222-3333-4444)
```

The main work in our class is done by the `unpack(format)` method, which is a string tokenizer on steroids, that can be used whenever you have to dissect a string of bytes. It expects a format string consisting of single-letter commands and returns an array of all values extracted from the string.

In our case we dynamically construct a format string that extracts sequences of ASCII characters from a string. An “a” followed by a number means “extract this amount of ASCII characters”. For our customer example it is “a19a20a30”, i.e. extract 19 characters, then 20 characters, and finally 30 characters. It should be obvious that it’s not possible to process binary data using our class.

## YAML Ain’t Markup Language (YAML)

You probably won’t have to integrate your software with existing YAML documents in your enterprise. But, in the Ruby world, YAML is popular, maybe even more popular than XML. So let’s spend a few pages looking at YAML.

YAML stands for “YAML Ain’t Markup Language.” That’s no lie: YAML actually does not define markup sequences the way XML does. Instead, it specifies a small set of simple rules for formatting structured data without much clutter, using plain text. YAML is available not only for Ruby, but also for Python, PHP, and Perl.<sup>17</sup>

The following YAML file (`themes.yaml`) represents a list of wrapping paper themes. It shows examples of array elements, which start with a `-` symbol and are separated by newline characters.

File

```
- red hearts
- lovely smileys
- rainbows
```

To load and parse the file, we use the YAML parser written by *why the lucky stiff*:

File

```
Line 1  require 'yaml'
        themes = YAML::load_file('themes.yaml')
        puts themes.class
        puts themes.join("\n")
```

This prints:

```
Array
red hearts
lovely smileys
rainbows
```

Obviously, the parser knows how to turn our list back into a native Ruby Array, but does it work the other way around, too?

File

```
Line 1  include YAML
        puts %(a spectacular example).to_yaml
```

produces:

```
- a
- spectacular
- example
```

When you include YAML, every object gets a `to_yaml()` method that returns its YAML representation. That whets our appetite; let’s see how YAML treats other data types:

File

```
Line 1  [
        - 'PragBouquet',
        - :aSymbol,
        - 42,
5       3.14,
```

---

<sup>17</sup><http://www.yaml.org>

```

-     true,
-     { 'chunky' => 'bacon', 'answer' => 42},
-   ].each do |obj|
-     puts "#{obj.class}#to_yaml:\n#{obj.to_yaml}\n"
10  end

```

produces:

```
String#to_yaml:
PragBouquet
```

```
Symbol#to_yaml:
:aSymbol
```

```
Fixnum#to_yaml:
42
```

```
Float#to_yaml:
3.14
```

```
TrueClass#to_yaml:
true
```

```
Hash#to_yaml:
answer: 42
chunky: bacon
```

All the basic data types (Fixnum, Float, Boolean, and String) seem to be encoded as ordinary Ruby literals. The encoding of Hash objects looks intuitive, too.

By default, every sequence of characters starting with an alphanumeric character is a String object in YAML unless it matches the more fine-grained rules for other data types.

As usual, symbols start with a colon (keep in mind that YAML is not specific to Ruby, so it has no specific syntax for symbols, as they do not exist in all dynamic languages.). Integer objects are sequences of digits that may start with an optional sign character (+ or -). They may contain commas for better readability, so -1,234 will be interpreted as an integer. Boolean values are represented by true and false. Float objects are two sequences of digits separated by a period ('.'). They may start with an optional sign character and in contrast to Ruby's literals for Float objects, the scientific notation for floating point numbers is allowed in YAML.

Hash objects are represented as key/value pairs where key and value are separated by a colon and each pair is separated by a newline char-

acter.

Now we know how Ruby's standard types are represented in YAML, let's look at more complex structures. When running this little program

```
File
Line 1  [
-      [%w(a nested list)],
-      ['another', %w(more deeply), 'nested', 'list'],
-      ['a', { 'list' => 'of', 3 => 'different' }, 'objects'],
5      { 'that' => %w(is getting), 'really' => { 'very' => 'complex' } }
-    ].each do |obj|
-      puts "#{obj.inspect}.to_yaml:\n#{obj.to_yaml}\n"
-    end
```

we get the following result:

```
[[ "a", "nested", "list" ].to_yaml:
- - a
- - nested
- - list

["another", ["more", "deeply"], "nested", "list"].to_yaml:
- another
- - more
- - deeply
- nested
- list

["a", { "list" => "of", 6 => "different" }, "objects"].to_yaml:
- a
- list: of
- 3: different
- objects

{ "that" => ["is", "getting"], "really" => { "very" => "complex" } }.to_yaml:
really:
- very: complex
that:
- is
- getting
```

No big deal: nested Arrays and Hashes are encoded by indenting the nested elements with leading spaces. The number of characters used for indentation doesn't matter, but it has to be consistent, so if the first element is indented by four characters, the following elements have to be indented by four characters, also.

As there are no more standard types left, let's have a look at other interesting objects.

```
File
Line 1  Flower = Struct.new(:name, :price)
-      [
```

```

-   Time.now,
-   Flower.new('rose', 1.99),
5   /even regexes/i,
-   'a'..'f'
-   ].each do |obj|
-     puts "#{obj.class}#to_yaml:\n#{obj.to_yaml}\n"
-   end

```

This is how our objects are “yamlified”:

```

Time#to_yaml:
2005-11-06 13:13:41.536822 +01:00

```

```

Flower#to_yaml:
!ruby/struct:Flower
name: rose
price: 1.99

```

```

Regexp#to_yaml:
!ruby/regexp /even regexes/i

```

```

Range#to_yaml:
!ruby/range
begin: a
end: f
excl: false

```

Timestamps and dates are encoded in iso-8601 format or—as shown in the example above—a slight variation where you can put spaces between the date, time, and time zone. Struct objects begin with `!ruby/struct:<struct name>`, followed by the members of the Struct object, listed as colon-separated key/value pairs. Regular expressions are encoded by putting `!ruby/regexp` in front. Finally, the representation of Range objects starts with the `!ruby/range` sequence, followed by the three attributes `begin`, `end`, and `excl`, corresponding to the range definition.

Now let’s see how YAML encodes objects created from the classes you write.

File

```

Line 1  class CustomClass
-     attr_accessor :a_hash, :an_array, :a_timestamp
-     end
-
5  custom_class = CustomClass.new
-  custom_class.a_hash = { 'how' => 'boring' }
-  custom_class.an_array = %w(still awake?)
-  custom_class.a_timestamp = Time.now
-  puts custom_class.to_yaml

```

prints:



```
!ruby/object:CustomClass
a_hash:
  how: boring
a_timestamp: 2005-11-07 19:36:54.477093 +01:00
an_array:
- still
- awake?
```

That should be fairly self explanatory.

We're looked at YAML serializing data. It can also deserialize data. If, for example, the file `demo.yaml` is as follows

File

```
- 42
- ~
- null
- true
- 1.2345
- :aSymbol
- 2005-11-07
- 1972-09-30T03:42:17.0+01:00
- !ruby/object:CustomClass
  a_hash:
    how: boring
  a_timestamp: 2005-11-07 19:36:54.477093 +01:00
  an_array:
  - still
  - awake?
```

and we parse it like this

File

```
Line 1  p YAML::load_file('demo.yaml')
```

then we get the following (manually formatted) output:

```
[
  42, nil, nil, true, 1.2345, :aSymbol,
  #<Date: 4907363/2,0,2299161>, Sat Sep 30 03:42:17 CET 1972,
  #<CustomClass:0x392ffc
    @a_hash={ "how"=>"boring"},
    @a_timestamp=Mon Nov 07 19:36:54 CET 2005,
    @an_array=["still", "awake?"]>
]
```

## Conclusion

YAML is a useful technology and its Ruby support is excellent. It still has many more useful features we didn't cover here.<sup>18</sup> For example, you can put several documents into a single file and you can define

---

<sup>18</sup>Visit <http://yaml4r.sourceforge.net/cookbook> to learn every little detail.

references to particular entries in a document and use them wherever you want. The next time you need a data format, you should give YAML a chance...

## Chapter 4

# Low Ceremony Distributed Applications

---

Few technologies have changed the IT landscape the way networks have. Today networks are ubiquitous. Some key players claim that the network is the computer. Think about it for a moment: when was the last time you switched on your computer and did not immediately connect to the internet? When was the last time you started your office PC and did not log in to a server using ssh or telnet?

Yes, it's true: networks have changed the way we think about computers significantly. They've changed the way we think about applications, too. Nowadays, even small applications often depend on distributed architectures where parts of a program are made available using network technologies. That's obvious on the internet, where you can freely use services offered by Amazon.com or Google. By sending simple HTTP requests you get back information about a book, or you can search for news about your favorite football team.

Big companies in the banking or telecommunications business were the first to adapt to the distributed applications paradigm. Because these companies often are spread across different continents, they were motivated to find ways to implement a feature or a function only once and to reuse it wherever it is possible. Today it's so easy to implement inter-process communication that nearly every company can benefit from using the technology.

You do not have to use heavy-weight industry standards like SOAP or CORBA to make your processes talk to each other. In this section we'll

show you how to use plain sockets and pure HTTP to separate concerns and to distribute business logic across process boundaries.

## 4.1 “I’d Rather Use a Socket”

On a conference panel in 2003 someone asked Robert C. Martin about the future of SOAP. After listening to the answers of his panelists, Uncle Bob replied: “I’d rather use a socket.”<sup>1</sup>. He is right: more often than not it’s sufficient to use some plain text files and a socket instead of huge databases and complex middleware. So, let’s start by looking at this simple approach.

Using plain sockets in an efficient and platform-independent manner isn’t trivial; handling multi-threading issues and the like can become fairly complicated. If you’re really interested in the nitty-gritty details of socket programming, you should have a look at *Unix Network Programming* [?] and the appendix of *Programming Ruby* [?]. Fortunately, there are a off-the-shelf solutions available in Ruby’s standard distribution.

Ruby comes with a class called `GServer` which helps in the creation of generic TCP servers. Written by John W. Small, `GServer` deals with stuff like connection handling and distributing requests to different threads, leaving you to implement the business logic. The unavoidable echo server example that sends back everything it gets looks like this:

File

```
Line 1  require 'gserver'
-
-  class Parrot < GServer
-    def initialize(host = 'localhost', port = '3333')
5      super(port, host)
-    end
-
-    def serve(client)
-      text = client.gets
10     client.puts(text)
-    end
-  end
-
-  lora = Parrot.new
15  lora.start
-  lora.join
```

On line 3 we derive our class `Parrot` from `GServer`, initializing it on line 5. The `serve(client)` method gets called whenever a client connects to

<sup>1</sup><http://www.artima.com/weblogs/viewpost.jsp?thread=4846>

our server. After reading what the client has to say, it sends it back as an echo. As you might have guessed already, `start()` starts the server and the `join()` method ensures that all running threads finish their work before shutting down the server. Here we have a recording from a live performance:

```
mschmidt:/tmp> telnet localhost 3333
⇒ Trying ::1...
   Connected to localhost.
   Escape character is '^]'.
⇐ Do you wanna have a cookie
⇒ Do you wanna have a cookie
   Connection closed by foreign host.

mschmidt:/tmp>
```

## The PragBouquet Status Monitor

Because building a server with `GServer` is so easy, we will try to solve a problem that has been around for a long time now at PragBouquet. Nearly every application running at PragBouquet writes a log file, often used for trouble shooting. Unfortunately, not all of them (to be exact: no two of them) use the same format and the files are scattered across several file systems. Even if you’re lucky and eventually find the file that should contain the information you need to resolve a trouble ticket initiated by one of your best (and probably most choleric) customers, chances are good that it has been overwritten already and no backup is available.

To overcome this situation we will create a status monitor, a TCP server whose only purpose is to centrally store messages sent by PragBouquet applications. Each message belongs to a certain application and is tagged with a severity level (warn, error, fatal). You might be wondering “Hey, why rewrite `syslogd`?”, but our little server differs from `syslogd` in a lot of ways: it’s used exclusively by PragBouquet applications, we can decide where and how to store our log messages, special actions can be triggered for certain log levels, and it’s platform-independent—it will run on Unix boxes as well as on Windows PCs.

Release candidate 1 of status monitor V0.0.1b only writes messages to `STDOUT`. It looks like this:

File

```
Line 1  require 'logger'
-       require 'gserver'
-
-       class StatusMonitor < GServer
```

```

5      def initialize(host = '127.0.0.1', port = '3333')
-        super(port, host)
-        @level_map = {
-          'w' => 'warn', 'e' => 'error', 'f' => 'fatal'
-        }
10       @logger = Logger.new(STDOUT)
-     end
-
-     # Expects CSV data in the following format:
-     # level,application,message.
15     def serve(client)
-       level, app, msg = client.readline.chomp.split(',', 3)
-       if @level_map.has_key?(level)
-         @logger.send(@level_map[level], "#{app}: #{msg}")
-         client.puts('0')
20       else
-         client.puts('1')
-       end
-     end
-   end
- end
25   sm = StatusMonitor.new
-   sm.start
-   sm.join

```

Let’s test it using good old telnet:

```
mschmidt:/tmp> telnet localhost 3333
```

```

⇒ Trying ::1...
Connected to localhost.
Escape character is '^]'.
⇐ f,billing,Lost connection to payment gateway.
⇒ 0
Connection closed by foreign host.

```

```
mschmidt:/tmp>
```

Our request has been successful (0 was returned). The corresponding server output looks like this:

```
F, [2005-09-03T16:32:03.952707 #1124] FATAL -- : billing: \
Lost connection to payment gateway.
```

Line 18 shows a nice Ruby trick; it dynamically invokes a method on our logger object using `send(symbol, (, args...))` (to read more about the `Logger` class, see Section 6.2, *Logging*, on page 250).

By the way: if you don’t want to test your `GServer` objects using old-fashioned manual prodding with `telnet`, you can easily use ordinary unit tests, too. Thanks to the magic of duck typing (see the sidebar, on page 146), you can pass any object to the `server()` method that responds to the `readline()` and `puts()` methods. `StringIO` works well:

File

```

Line 1  require 'test/unit'
-       require 'stringio'
-       require 'status_monitor'
-
5       class StatusMonitorTest < Test::Unit::TestCase
-         def setup
-           @server = StatusMonitor.new
-         end
-
10      def test_empty_string
-        result = simulate_request("\n")
-        assert_equal('1', result)
-      end
-
15      def test_invalid_level
-        result = simulate_request("x,foo,invalid level\n")
-        assert_equal('1', result)
-      end
-
20      def test_normal_case
-        result = simulate_request("e,foo,normal case\n")
-        assert_equal('0', result)
-      end
-
25      def simulate_request(request)
-        client = StringIO.new(request)
-        @server.serve(client)
-        client.string[request.size - 2 .. -2]
-      end
30  end

```

produces:

```

Loaded suite sm_test
Started
..E, [2005-10-14T08:00:54.960679 #374] ERROR -- : foo: normal case
.
Finished in 0.012851 seconds.

3 tests, 3 assertions, 0 failures, 0 errors

```

For the unit tests, we initialize StringIO objects with messages that potential clients could send to the status monitor. The StatusMonitor's serve() method doesn't care what class its clients belong to, and happily reads requests from a StringIO object and writes results to it.

The main work is done by the simulate\_request() method. This calls the serve() method, passing it a StringIO object. To read the result on line 28, we have to ignore the message that is still in the StringIO object and we have to ignore the line feed the server sends at the end of the message.

### **Duck Typing**

Ruby is an object-oriented language. Where there are objects, classes and types are usually not far away. However, in contrast to Java or C++ programs Ruby programs are not cluttered with type declarations. In Ruby, the type of variables, methods, and method parameters do not have to be explicitly declared. Despite this, all Ruby objects have a certain type.

However, when you’re programming in a dynamic language you soon realize that the most important question about a particular object is normally not “What’s its class, and what are *all* the methods it responds to?” but instead “Does the object at hand respond to `foo()`?”

That’s where the duck analogy comes from: if it walks like a duck and talks like a duck, then it will be treated as if it were a duck. No matter, if it actually is one or not.

This situation is not uncommon even in apparently statically typed languages. For example, Java programmers expect every object to have a `toString()` method. In Java, this is because all objects are derived from the omnipresent `Object` class. The implementation is different, but the principle is the same: you want a particular object to respond to a certain message; its class doesn’t matter much to you.

Our implementation of the `GServer`’s `serve(client)` method doesn’t care about `client`’s class: it doesn’t declare its type and it doesn’t check if it is actually some kind of TCP socket. It only expects `client` to respond to methods called `readline()` and `puts(text)`.



## Status Monitor Clients

The biggest problem with servers is that they are totally useless without a client. In our case, we have to create one for every programming language PragBouquet uses. Although this may seem like a lot of redundancy, it's always a good idea to provide programmers with the highest degree of flexibility. If there is no status monitor client library for Java, they will not use it (the status monitor, of course. Not Java). If you think that the status monitor is a GOOD THING and you want your colleagues to use it, then you have to make it as painless as possible to do so.

Currently, there are not many Ruby applications in the PragBouquet environment, but we want to be prepared. Feeding our status monitor from Ruby programs can be accomplished using class TCPSocket:

File

```
Line 1  require 'socket'
-
-  class StatusMonitorClient
-    WARN = 'w'
5    ERROR = 'e'
-    FATAL = 'f'
-
-    def initialize(host, port)
-      @sm = TCPSocket.new(host, port)
10    end
-
-    def warn(app, msg) log(WARN, app, msg); end
-    def error(app, msg) log(ERROR, app, msg); end
-    def fatal(app, msg) log(FATAL, app, msg); end
15
-    def terminate() @sm.close; end
-
-    private
-
20    def log(level, app, msg)
-      @sm.puts [level, app, msg].join(',')
-      @sm.readline
-    end
-  end
```

At the heart of class StatusMonitorClient is the log(level,app,msg) method. It sends a string to the status monitor and reads back the result. Because TCPSocket objects behave like any other IO instance, it's easy to implement.

You can use this class as follows:

File

```
Line 1  sm = StatusMonitorClient.new('127.0.0.1', 3333)
```

```
- # ...
- sm.fatal('billing', 'Lost connection to payment gateway.')
- # ...
5 sm.terminate
```

One of the most important clients will be the Java client, because it has been the programming language of choice for a long time at Prag-Bouquet. As we did with the Ruby client, we won't go abstracting too much. At the same time, adding a little layer on top of the TCP stack will certainly pay off in the future.

File

```
Line 1  import java.io.*;
-       import java.net.*;
-
-       public class StatusMonitor {
5         public static final String WARN = "w";
-         public static final String ERROR = "e";
-         public static final String FATAL = "f";
-
-         public StatusMonitor(final String host, final int port)
10          throws UnknownHostException, IOException
-         {
-             _sm = new Socket(host, port);
-             _out = new PrintWriter(_sm.getOutputStream(), true);
-             _in = new BufferedReader(
15              new InputStreamReader(_sm.getInputStream())
-             );
-         }
-
-         public int warn(final String app, final String msg)
20          throws IOException
-         {
-             return log(WARN, app, msg);
-         }
-
-         public int error(final String app, final String msg)
25          throws IOException
-         {
-             return log(ERROR, app, msg);
-         }
-
30         public int fatal(final String app, final String msg)
-          throws IOException
-         {
-             return log(FATAL, app, msg);
35        }
-
-         public void terminate() throws IOException {
-             _out.close();
-             _in.close();
40            _sm.close();
```

```

-     }
-
-     private int log(
-         final String level,
45         final String app,
-         final String msg) throws IOException
-     {
-         final String DEL = ",";
-         _out.println(level + DEL + app + DEL + msg);
50         final String response = _in.readLine();
-         int return_code = 1;
-         try {
-             return_code = Integer.parseInt(response);
-         }
55         catch(NumberFormatException ignoreMe) {}
-         return return_code;
-     }
-
-     private Socket _sm;
60     private PrintWriter _out;
-     private BufferedReader _in;
- }

```

Again, most of the work is done in the `log(level,app,msg)` method. Because this is a book about Ruby we will not go into the details of the `StatusMonitor` class. However, we will demonstrate how to use it:

```

File
Line 1  public class StatusMonitorTest {
-     public static void main(String[] args) {
-         try {
-             final StatusMonitor sm = new StatusMonitor(
5             "127.0.0.1",
-             3333
-             );
-             final int result = sm.debug("billing", "ALAAARM!!");
-             System.out.println(result);
10            sm.terminate();
-         }
-         catch(Exception e) {
-             System.err.println(
-                 "An error occurred: " + e.getMessage()
15            );
-         }
-     }
- }

```

Oh, and we shouldn't forget the hordes of Perl programmers that inhabited our IT department in the past. All the poor guys who have to maintain their legacy code have the right to use our amazing status monitor, too.

```

File
Line 1  use strict;
-       use IO::Socket;
-
-       package StatusMonitor;
5
-       use constant WARN => 'w';
-       use constant ERROR => 'e';
-       use constant FATAL => 'f';
-
10      sub new {
-          my $class = shift;
-          my ($host, $port) = @_;
-          my $self = {};
-          my $sm = IO::Socket::INET->new("$host:$port");
15      die "Could not connect to status monitor: $!\n" unless $sm;
-          $self->{'sm'} = $sm;
-          bless $self, $class;
-      }
-
20      sub warn() {
-          my ($self, $app, $msg) = @_;
-          $self->_log(WARN, $app, $msg);
-      }
-
25      sub error() {
-          my ($self, $app, $msg) = @_;
-          $self->_log(ERROR, $app, $msg);
-      }
-
30      sub fatal() {
-          my ($self, $app, $msg) = @_;
-          $self->_log(FATAL, $app, $msg);
-      }
-
35      sub _log() {
-          my ($self, $level, $app, $msg) = @_;
-          print { $self->{'sm'} } "$level,$app,$msg\n";
-          my $sm = $self->{'sm'};
-          <$sm>;
40      }
-
-      sub DESTROY {
-          close($_[0]->{'sm'});
-      }
45
-      1;

```

For those who are familiar with Perl (the Pathological Eclectic Rubbish Lister as its inventor Larry Wall sometimes calls it) the lines above should be no problem. For all the others it will look like line noise

until they have studied Perl for several years. Just believe it: this code achieves the same results as our Ruby and Java code. Here’s a little test program to prove it:

File

```
use strict;
use status_monitor;

my $sm = StatusMonitor->new('127.0.0.1', 3333);
$sm->error('tracking', 'Lost connection to service.');
```

This produces:

```
E, [2005-09-17T17:23:43.216794 #1419] ERROR -- : tracking: \
Lost connection to service.
```

Admittedly, this is a lot of code that has to be maintained from now on, but it has some valuable benefits. Although we had to implement TCP client code for various languages, we can now add features to the status monitor without touching the client libraries. For example, we can send an e-mail to the operations department whenever an application logs a fatal error. Additionally, to change the transport layer, only the different `log(level,app,msg)` methods have to be changed.

## Adding Better Persistence

Let’s illustrate that flexibility by replacing our cheap logger by a full-blown MySQL database. At the same time, we’ll add an e-mail feature, too. Our little database (called `smon`) consists of only one table, `log_entries`. Its structure should be fairly clear:

File

```
create table log_entries(
  id int unsigned not null primary key,
  application varchar(64) not null,
  level enum('warn', 'error', 'fatal'),
  message text,
  created timestamp not null
);
```

For convenience, we access the status monitor database using `ActiveRecord`.

File

```
Line 1  require 'rubygems'
-       require 'active_record'
-       require 'gserver'
-
5  ActiveRecord::Base.establish_connection(
-    :adapter => 'mysql',
-    :host => '127.0.0.1',
-    :database => 'smon'
```

```

-   )
10
-   class LogEntry < ActiveRecord::Base; end
-
-   class StatusMonitor < GServer
-       def initialize(host = '127.0.0.1', port = '3333')
15           super(port, host)
-           @level_map = {
-               'w' => 'warn', 'e' => 'error', 'f' => 'fatal'
-           }
-       end
20
-       # Expects CSV data in the following format:
-       # level,application,message.
-       def serve(client)
-           level, app, msg = client.readline.chomp.split(',', 3)
25           if @level_map.has_key?(level)
-               entry = LogEntry.new
-               entry.application = app
-               entry.level = @level_map[level]
-               entry.message = msg
30               entry.save
-               client.puts('0')
-           else
-               client.puts('1')
-           end
35       end
-   end
end

```

First, we removed the `Logger` class (maybe we will use it again for logging the status monitor's own status, but for now we try to keep the examples short). Next, we added initialization code for `ActiveRecord` and the `LogEntry` class (to learn more about `ActiveRecord`, see Section 2.3, *ActiveRecord Basics*, on page 33). In our `serve(client)` method we didn't have to change a lot either. All the logging stuff was replaced by initialization code for a new `LogEntry` object that gets saved for each request.

None of the client libraries had to be touched. By changing only a few lines of Ruby code, we solved one of our biggest problems—important information is now stored centrally. From now on we do not have to search tons of log files to find vital information; a simple `SELECT` statement will be sufficient. Provided that your database and system administrators earn their salary by creating regular database backups, this information will be safe forever.

## Sending E-Mails

At this point we're unstoppable. Let's add some more code to the status monitor that sends an e-mail to the operations department whenever an application logs a fatal error.

File

```

Line 1  require 'rubygems'
-       require 'active_record'
-       require 'gserver'
-       require 'tmail'
5       require 'net/smtp'
-
-       ActiveRecord::Base.establish_connection(
-         :adapter => 'mysql',
-         :host => '127.0.0.1',
10      :database => 'smon'
-     )
-
-     class LogEntry < ActiveRecord::Base; end
-
15    class StatusMonitor < GServer
-      def initialize(host = '127.0.0.1', port = '3333')
-        super(port, host)
-        @level_map = {
-          'w' => 'warn', 'e' => 'error', 'f' => 'fatal'
20        }
-      end
-
-      # Expects CSV data in the following format:
-      # level,application,message.
25      def serve(client)
-        level, app, msg = client.readline.chomp.split(',', 3)
-        if @level_map.has_key?(level)
-          entry = LogEntry.new
-          entry.application = app
-          entry.level = @level_map[level]
30          entry.message = msg
-          entry.save
-          puts "Entry was saved."
-          inform_helpdesk(app, level, msg) if level == 'f'
35          client.puts('0')
-        else
-          client.puts('1')
-        end
-      end
-
40      private
-
-      def inform_helpdesk(app, level, msg)
-        subject = "A fatal error occured in #{app}!"
45        subject << " Regret your sins!"

```

```

-         mail = create_mail(
-             'helpdesk@pragbouquet.com',
-             subject,
-             msg
50         )
-         send_mail('localhost', mail)
-     end
-
-     def create_mail(to, subject, body)
55         mail = TMail::Mail.new
-         mail.date = Time.now
-         mail.mime_version = '1.0'
-         mail.set_content_type 'text', 'plain'
-         mail.from = 'status_monitor@pragbouquet.com'
60         mail.to = to
-         mail.subject = subject
-         mail.body = body
-         mail
-     end
65
-     def send_mail(host, mail)
-         msg = mail.encoded
-         Net::SMTP.start(host, 25) do |smtp|
-             smtp.send_mail(msg, mail.from_address, mail.destinations)
70         end
-     end
- end

```

Sending e-mail can be divided into two parts: creating the SMTP (Simple Mail Transfer Protocol) formatted e-mail and sending that e-mail over a network. To create the SMTP representation, we used Minero Aoki’s excellent *tmail* library<sup>2</sup> (you can see how to obtain and install it in Section 6.4, *Deploying With setup.rb*, on page 277.).

In the `create_mail(to,subject,body)` method we assemble a `TMail::Mail` object that contains all the attributes we’d expect an e-mail to have. Finally, on line 67, we call `encoded()`. It returns something like this:

```

Date: Sat, 17 Sep 2005 14:48:30 +0200
From: status_monitor@pragbouquet.com
To: helpdesk@pragbouquet.com
Subject: A fatal error occurred in billing! Regret your sins!
Mime-Version: 1.0
Content-Type: text/plain

```

Lost connection to payment gateway.

---

<sup>2</sup><http://raa.ruby-lang.org/project/tmail>



Ruby's standard class `Net::SMTP` comes quite handy to send the e-mail we just created. Lines 68 to 70 of method `send_mail(host,mail)` show how to accomplish this.

We're done! The application is now backed by a database and sends an e-mail to the operations department whenever a fatal application error occurs.

## 4.2 Remote Procedure Calls Using HTTP

Our new status monitor has been a huge success—the system failure rate has dropped down to an all-time low of 2 per day, each lasting for less than 10 minutes. Unfortunately, this is true only for the time between 8 am and 5 pm, Monday to Friday. Every night 3 to 5 fatal application errors still occur, and they don't get solved until the following morning.

Obviously, the status monitor e-mails are only read during the office hours. It would increase our quality of service significantly if we could inform our operations department's employees of failures wherever they are, and whatever time it is. A good way to do this is to use the Short Messages (SMS) supported by cellular networks: to send the message you simply need a cellular number, and to receive it the employee need only keep their cell phone with them.

### Waking Up the Operator

To send short messages from a computer to a cell phone, you can:

1. Serially connect a computer and a cellular phone or modem that can be controlled via an AT cellular command interface.
2. Connect to a Short Message Service Center (SMSC) at your network provider. You send messages using a protocol called Short message peer-to-peer protocol (SMPP)<sup>3</sup> to the SMSC, and the SMSC is responsible for delivering them to the according mobile devices.
3. Use an existing web service on the Internet.

PragBouquet chose the SMSC connection. We already have a server that is capable of sending SMS. This particular piece of software offers a simple interface via HTTP: it supports the single function `send(recipient,sender,type,data)`.

---

<sup>3</sup><http://en.wikipedia.org/wiki/SMPP>

You have to distinguish between Short Messages containing textual or binary data. Set type to “text” or “binary” accordingly (“text” is the default). Binary data has to be transmitted in two-digit hexadecimal ASCII representation, so a binary zero is transmitted as 00 and binary 63 is transmitted as 3f.

The remaining parameters contain the data to be sent and the recipient’s and the sender’s phone numbers in international format. If the sender parameter is not set, it will be set to the recipient parameter automatically. This way the recipient thinks he has sent himself a message—in our case that is alright.

The service listens on port 4242 under path “/send”. It returns its result using the HTTP status code, so 200 means everything went fine and 500 means an error occurred. Let’s try it using telnet:

```
mschmidt:/tmp> telnet localhost 4242

⇒ Trying ::1...
   Connected to localhost.
   Escape character is '^J'.
⇐ GET /send?type=text&recipient=+011234123456&data=hello HTTP/1.0
⇒
HTTP/1.1 200 OK
Connection: close
Date: Wed, 14 Sep 2005 20:26:47 GMT
Content-Type: text/plain
Server: WEBrick/1.3.1 (Ruby/1.8.2/2004-12-25)
Content-Length: 0

Connection closed by foreign host.

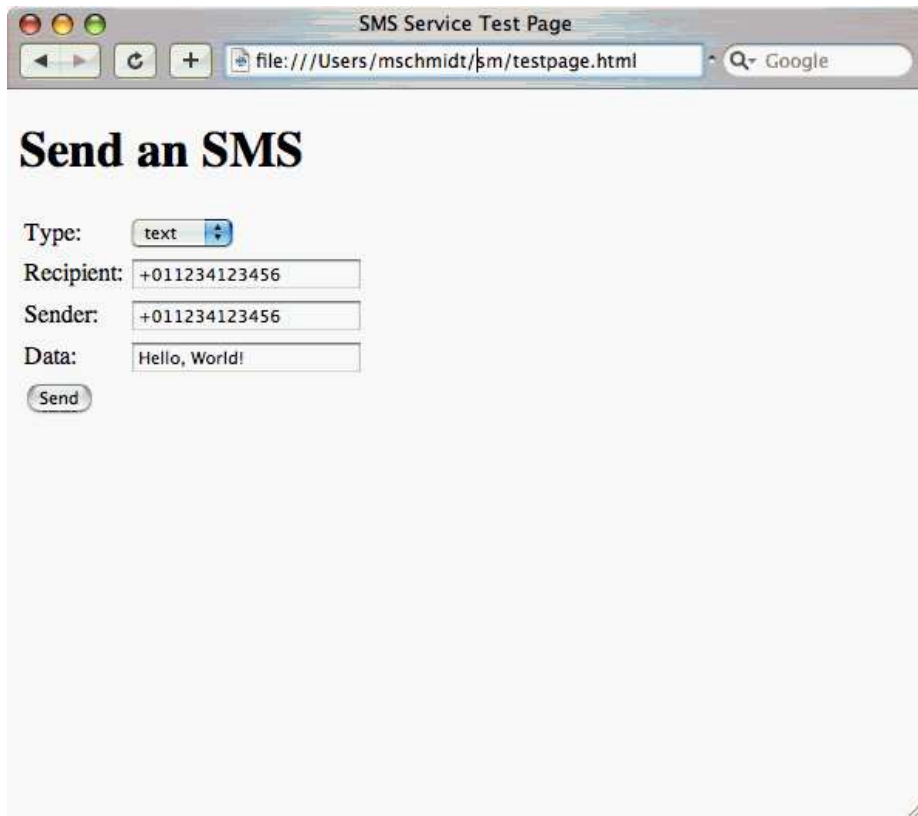
mschmidt:/tmp>
```

Hm, there’s no error message from the session, but our cell phone didn’t yell “YOU HAVE A NEW MESSAGE!” at full volume as it usually does when it receives an SMS. So, what’s wrong? Provided that the SMS server is working properly, the problem must be hidden in our input. Obviously, the type and data parameters are correct, but the recipient’s phone number contains a subtle syntax error. All international phone numbers start with one of the following prefixes<sup>4</sup>:

- “00”<international area code><national area code>
- “+”<international area code><national area code>

---

<sup>4</sup>Usually, a leading zero of the national area code is omitted, but you should not blindly follow this rule. For Italy, as an example, you have to explicitly transmit the leading zero of the national area code in international phone numbers.



SMS Service Test Page

file:///Users/mschmidt/§m/testpage.html

## Send an SMS

Type: text

Recipient: +011234123456

Sender: +011234123456

Data: Hello, World!

Send

---

Figure 4.1: Testing The Status Monitor

---

At first sight, everything seems to be alright with the recipient's phone number, but we forgot that we are using an HTTP service that expects all its parameters in URL-encoded format. Hence, the leading + sign is interpreted as a blank which renders the phone number completely wrong. Instead of transmitting "+011234123456" (an American phone number) " 011234123456" (a national phone number with a leading blank) is transmitted.

In the best case, this phone number does not exist and the SMSC fails to deliver your message. In the worst case, someone could become really angry while you're desperately running your unit test suite over and over again at 2am on a Saturday night.

When testing complex HTTP services the telnet command can be a bit tedious. For services expecting GET requests you're much better off using an ordinary web browser that transparently handles all encoding issues for you (command line die-hards will use `wget`<sup>5</sup> or `curl`<sup>6</sup> anyway). A HTML page such as the following is sufficient for testing purposes:

File

```

Line 1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
-       "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
-
-   <html>
-   <head>
-       <title>SMS Service Test Page</title>
-   </head>
-   <body>
-       <h1>Send an SMS</h1>
10      <form action="http://localhost:4242/send">
-          <table>
-              <tr>
-                  <td>Type:</td>
-                  <td>
15                      <select name="type">
-                          <option>text</option>
-                          <option>binary</option>
-                      </select>
-                  </td>
20          </tr>
-          <tr>
-              <td>Recipient:</td>
-              <td><input type="text" name="recipient"/></td>
-          </tr>
25          <tr>
-              <td>Sender:</td>
-              <td><input type="text" name="sender"/></td>
-          </tr>
-          <tr>
30              <td>Data:</td>
-              <td><input type="text" name="data"/></td>
-          </tr>
-          <tr>
-              <td colspan="2">
35                  <input type="submit" value="Send"/>
-              </td>
-          </tr>
-          </table>
-      </form>
40  </body>
-  </html>

```

<sup>5</sup><http://www.gnu.org/software/wget/wget.html>

<sup>6</sup><http://curl.haxx.se>

## URL Encoding

In the beginning of computer networking everything had to be represented in ASCII characters using only 7 bits. Because of this nearly all protocols in today's Internet turn non-ASCII data into 7-bit ASCII characters somehow. RFC 1738\* painstakingly explains which characters are allowed in URLs:

"Only alphanumerics [0-9a-zA-Z], the special characters [\$\_.+!\*'(),], and reserved characters used for their reserved purposes may be used unencoded within a URL."

The rules are simple: all ASCII control characters (0x00-0x1f, 0x7f), all non-ASCII characters (0x80-0xff), and all reserved characters ([\$&+./:;=?@]) have to be encoded under all circumstances. In addition, some characters considered unsafe should be encoded, too: ["?<>#%{}|\ ^~`] and the square brackets ([, ]) themselves (0x5b, 0x5d).

Encoding a single value is easy: you prepend its case-insensitive, two-digit hexadecimal ISO-Latin code by a percent symbol (%). For example, the blank character is encoded as %20 (for convenience, a blank can also be encoded as a single '+' symbol) and an upper-case 'A' (whose ISO-Latin code is 65) is turned into %41.

\*<http://www.rfc-editor.org/rfc/rfc1738.txt>

Another way to circumvent problems like these is to use a mature HTTP client library, such as the one that comes with the Ruby standard distribution. Let's see how we can perform HTTP requests and encapsulate the SMS server:

File

```
Line 1  require 'net/http'
-       require 'cgi'
-
-       class SmsService
5         def initialize(host = '127.0.0.1', port = 4242)
-           @host, @port = host, port
-         end
-
-         def send_text(params)
10          send_sms(params, 'text')
-         end
-
-         def send_binary(params)
```

```

-         send_sms(params, 'binary')
15     end
-
-     private
-
-     def send_sms(params, type)
20         result = false
-         Net::HTTP.start(@host, @port) do |http|
-             query = "type=#{type}"
-             params.each do |k,v|
-                 query << "&#{k}=#{CGI.escape(v)}"
25             end
-             response = http.get("/send?#{query}")
-             result = response.class == Net::HTTPOK
-
-             end
-             result
30         end
-     end

```

A small sample program demonstrates the usage of the `SmsService` class. Finally you (and all your enervated colleagues) can hear it again: “YOU HAVE A NEW MESSAGE!”.

File

```

Line 1  sms = SmsService.new
-       sms.send_text(
-           :recipient => '+0112341234567',
-           :data => 'Hello, world!'
5       )

```

In `send_sms(params,type)` of the `SmsService` class we open a new HTTP connection using the `start(host,port)` method of class `Net::HTTP`. It expects a code block that gets passed the current connection. Within the code block the query string for the GET request is prepared (note that we have to use the `encode(string)` method of the `CGI` class to URL-encode our query parameters.).

Eventually, on line 26 we initiate a GET request that returns an object derived from `Net::HTTPResponse`. This object encapsulates everything that makes up an HTTP response. Its most important methods are:

- `code()` returns the HTTP status code.
- `message()` returns the HTTP status message.
- `body()` returns the response body.
- The headers Hash contains all HTTP headers.

There is a separate class for every HTTP status code (yes, this little library defines more than 50 classes), so instead of checking if `code()`

returns 200 you can alternatively check if your response object's class is `Net::HTTPOK` (as we did in line 27).

Integrating the SMS service into the `StatusMonitor` is left as an exercise for the reader. Keep in mind that Short Messages do have to be short: they shouldn't be more than 160 characters. Sending Java stack traces is definitely not an option.

If you don't like encoding query parameters manually, you can use HTTP's POST command. In the current example it doesn't make a big difference, though. Here's how `send_sms(params,type)` would look:

```
File
Line 1  def send_sms(params, type)
-       result = false
-       Net::HTTP.start(@host, @port) do |http|
-         query = "type=#{type}"
5         params.each { |k,v| query << "&#{k}=#{v}" }
-         response = http.post(
-           '/send',
-           query,
-           'content-type' => 'application/x-www-form-urlencoded'
10        )
-         result = response.class == Net::HTTPOK
-       end
-       result
-     end
```

In lines 4 and 5 we build up the query string by concatenating key/value pairs using the `=` symbol. The pairs themselves are delimited by `'&'` symbols. That's how input values from an HTML form get transferred to a server and therefore, we have set the content type of our POST request to `"application/x-www-form-urlencoded"`. As the server now knows how to interpret the data, and because we're not sending it in a URL, we don't have to encode it.

## The Other Side Of HTTP

We have seen that accessing HTTP services with Ruby is a breeze. This made us think a bit: shouldn't we make the status monitor available as a HTTP service, too? It certainly would have some advantages:

- We could get rid of the proprietary CSV parameter list, which would make it much easier to add new parameters.
- Adding additional functions would be easier, because we could publish them using different URLs—new functions would never interfere with existing ones.

### More Privacy!

Even in the early days of the World Wide Web security was an important issue. In 1994 Netscape Communications invented a protocol called Secure Sockets Layer (SSL)\* that enabled web clients and servers to exchange sensitive data without worries.

The first protocol version and its implementation was full of flaws, but since then SSL has become a de facto standard and has been improved several times. Currently, its most popular implementation is OpenSSL†. The OpenSSL project has created a C library that has been ported to countless platforms and is the basis for Ruby's SSL support, too.

In the simplest cases it makes nearly no difference whether you use HTTP or HTTPS. For example, printing the index page of the OpenSSL web site can be achieved as follows:

File

```
Line 1  require 'net/https'
-
-      h = Net::HTTP.new('www.openssl.org', 443)
-      h.use_ssl = true
5      h.get2('/') { |response| print response.body }
```

You have to set the `use_ssl` attribute of your `Net::HTTP` object to `true` and you have to use `get2(path,initheader=nil,&block)` instead of `get(path,initheader=nil,dest=nil,&block)`. Oh, and it's important that you explicitly set the SSL port (443) when opening the connection. Otherwise, the HTTP default port (80) will be used and your request will probably fail.

Because it's based on the OpenSSL reference implementation, Ruby's support of the SSL protocol is as secure and complete as it can be. You can encrypt and decrypt communication transparently (as we did in the example above). There are methods for signing and verifying both client and server certificates. For example, `peer_cert()` returns the server's X.509 certificate and using `cert=(certificate)` you can set a client's X.509 certificate. The library allows you to maintain a certificate store in your local file system, too.

---

\*[http://en.wikipedia.org/wiki/Secure\\_Sockets\\_Layer](http://en.wikipedia.org/wiki/Secure_Sockets_Layer)

†<http://www.openssl.org>



## HTTP Proxy Servers

Our first HTTP example works fine, because the SMS service is running on a server in the local network. In big companies HTTP access to servers on the Internet is often routed through a *proxy server*. Proxies cache frequently requested web sites to decrease network load, and they restrict access to unapproved content.

HTTP clients have to connect to the proxy server instead of the actual server. The proxy server then forwards the client's request and sends back the result. Using a proxy server with Net::HTTP works as follows:

```
Net::HTTP::Proxy(
  proxy_host,
  proxy_port).start('www.example.com') do |http|
  # http is connected to proxy_host:proxy_port
end
```

If the proxy host is set to nil, no proxy will be used, so it will do no harm if you always use the proxy call.

Sometimes a proxy server expects an authentication via a user name and a password. Such information is passed as follows:

```
Net::HTTP::Proxy(
  proxy_host, proxy_port,
  proxy_user, proxy_pass).start('www.example.com') do |http|
  # http is connected to proxy_host:proxy_port
end
```

- We could access the status monitor with an ordinary web browser. This certainly has some advantages when we add statistics or query features.

Now that we've convinced ourselves, let's see what the Ruby platform has to offer for creating HTTP services. Unsurprisingly, it comes with one of the most advanced and most convenient frameworks currently available: WEBrick. WEBrick really has it all: you can create generic HTTP and HTTPS servers. All of them are multi-threaded, they support servlets, and you will find a bunch of useful utility classes and functions, too.

So, let's see how we can turn our status monitor into a HTTP service:

File

```
Line 1  require 'webrick'
-      include WEBrick
```

```

-   class StatusMonitorServlet < HTTPServlet::AbstractServlet
-
-   # ...
-
-   def do_GET(req, res)
-       app = req.query['app']
-       msg = req.query['msg']
10      level = req.query['level'] || ''
-       level.downcase!
-       res['content-type'] = 'text/plain'
-       if %w(debug info warn error fatal).include?(level)
-           entry = LogEntry.new
15          entry.application = app
-           entry.level = level
-           entry.message = msg
-           entry.save
-           inform_helpdesk(app, level, msg) if level == 'fatal'
20          res.status = 200
-           res.body = "Message was logged successfully."
-       else
-           res.status = 500
-           res.body = "An error occurred while logging message!"
25      end
-   end
-
-   alias do_POST :do_GET
-
-   # ...
-
-   end
-
-   server = HTTPServer.new(:Port => 4242)
35  server.mount('/log', StatusMonitorServlet)
-
-   %w(INT TERM).each do |signal|
-       trap(signal) { server.shutdown }
-   end
40
-   server.start

```

Even if it doesn't seem obvious at first sight, we only had to modify our last version a small amount. The StatusMonitor class has been renamed to StatusMonitorServlet and is derived from WEBrick::HTTPServlet::AbstractServlet. initialize(host,port) has been removed and serve(client) has been replaced by do\_GET(request,response). Everything else was left untouched.

Whenever the server receives a GET request, do\_GET(request,response) is called automatically. The query parameters app, level, and msg are read from the client request. This renders the former CSV solution obsolete and adding new parameters is trivial now (for better readability we have

dropped the `level_map` hash).

What follows is business as usual. We initialize a new `LogEntry` object and store it in the database. Instead of sending back '0' or '1' to indicate success or failure we now set the HTTP status code accordingly in lines 20 and 23. Another way of setting the response status is to raise an exception, so `raise HTTPStatus::OK` is the same as `res.status = 200`. In this case it does not seem to make a lot of sense, but raising an exception for an error condition such as `HTTPStatus::InternalServerError` automatically sets the response body to an HTML error page containing a complete stack trace. Such a page will be returned for uncaught exceptions, too.

Because there is no reason for the status monitor to care about the request method, we have aliased `do_GET(request,response)` with `do_POST(request,response)` on line 28. This allows clients to freely choose which request method they want to use.

Naked servlets are nonviable; they depend heavily on a friendly server environment to live in. Hence, we create a servlet environment beginning on line 34. In the next line the `StatusMonitorServlet` is mounted under the path `"/log"` and lines 37 to 39 install a callback for `INT` and `TERM` signals that shuts down the server gracefully when it receives an corresponding signal. After starting the server in the last line, you can point your web browser to `"http://localhost:4242/log?app=billing&level=info&msg=hello"` to create a new log message.

## Some WEBrick Details

WEBrick's rules for dispatching HTTP requests are simple. When the server receives an command, for example `PUT`, it calls the corresponding `do_XXX` method, in this case `do_PUT(request,response)`. WEBrick does not restrict this dispatching mechanism to the official HTTP commands and you're free to define your own HTTP commands, such as the one in our friendly greeting service below.

File

```
Line 1  class FriendlyServlet < HTTPServlet::AbstractServlet
-      def do_GREET(req, res)
-          res.status = 200
-          res.body = "Hello!! Nice to meet you!\n"
5      end
-  end
-
-  server = HTTPServer.new(:Port => 4200)
-  server.mount('/', FriendlyServlet)
```

```

10  %w(INT TERM).each do |signal|
-    trap(signal) { server.shutdown }
-  end
-  server.start

```

You can use it like any other HTTP command and you will receive the same request parameters, too (so, if you need a WEBDAV<sup>7</sup> extension, for example, go ahead and build one).

```
mschmidt:/tmp> telnet localhost 4200
```

```

⇒  Trying ::1...
    Connected to localhost.
    Escape character is '^]'.
⇐  GREET / HTTP/1.0
⇒
    HTTP/1.1 200 OK
    Connection: close
    Date: Fri, 23 Sep 2005 07:18:19 GMT
    Server: WEBrick/1.3.1 (Ruby/1.8.2/2004-12-25)
    Content-Length: 26

    Hello!! Nice to meet you!
    Connection closed by foreign host.

```

The HEAD and OPTIONS commands have default implementations:

`do_HEAD(request,response)`

Returns everything `do_GET(request,response)` would return except the response body. This enables clients to check, whether a certain document has changed, so it is downloaded only if it is necessary.

`do_OPTIONS(request,response)`

Returns a list of all commands available

All `do_XXX` methods get passed request and response parameters of type `WEBrick::HTTPRequest` and `WEBrick::HTTPResponse`, respectively. Both of them provide all the things you'd normally expect. The most important methods and attributes of `WEBrick::HTTPRequest` are:

- `query` is a Hash object containing all query parameters that have been sent by the client. If a parameter is transmitted more than once, the corresponding entry in the query Hash is an Array object containing *all* values transmitted.

---

<sup>7</sup><http://www.webdav.org>

- `header` is a Hash object containing all HTTP headers sent by the client.
- `cookies` is an Array object containing all cookies sent by the client. Array elements are `WEBrick::Cookie` objects.
- Details of the resource requested by the client are available in the attributes `host`, `port`, `request_uri`, `query_string`, `script_name`, and `path_info`.

`WEBrick::HTTPResponse` is just as simple.

- Put the overall status of a request into the `status` attribute.
- Set `body` to the data you want to send back to the client. Make sure it corresponds to the response's content type.
- `WEBrick` automatically sends back standard HTTP headers such as `server` and `date`. The header Hash allows you to add your own. For example,

```
response.header['content-type'] = 'text/xml'
```

declares that the server output is an XML document.

- You can send cookies to the client by adding `WEBrick::Cookie` objects to the `cookies` Array.

There are many more methods and attributes. Most are simply for convenience, to provide access to low-level HTTP properties.

## It Can Always Be Easier

Although creating your own servlets is easy, `WEBrick` comes with some default servlet implementations for common tasks such as serving files (we talk about the `FileHandler` in Section 4.2, *WEBricklets*, on page 170) and executing CGI scripts.

Sometimes it seems to be overkill to define a complete servlet class. The `WEBrick::HTTPServer` class has a highly convenient shortcut:

File

```
Line 1  server = HTTPServer.new(:Port => 4100)
-      server.mount_proc('/') do |req, res|
-          res.status = 200
-          res.body = "Hello, #{req.query['name']}!"
5      end
-
-      %w(INT TERM).each do |signal|
-          trap(signal) { server.shutdown }
```

```
- end
10 server.start
```

In line 2 we add a code block using `mount_proc(dir,&block)` to a server object. These handlers respond to both GET and POST requests and are managed by an instance of class `ProcHandler`.

Another useful WEBrick handler is `CGIHandler`. It's invoked automatically by `FileHandler` when the request URL ends with the extension `.cgi`. The *Common Gateway Interface* (CGI) defines a simple protocol between web servers and programs that create dynamic content: the web server runs an executable program with the `exec()` system call and the CGI protocol specifies a list of environment variables that can be used by the CGI script to get request parameters and so on. After the program has terminated, the web server sends its output to the web client. Typical companies have tons of old CGI programs (often Perl or bash scripts) that “will definitely be replaced as soon as we have some time.” Here's a simple Perl script that creates an XML report of all flowers that are currently used in our bouquets:

*Common Gateway  
Interface*

File

```
Line 1  #!/usr/bin/perl -w
-
- use strict;
- use DBI;
5
- sub get_flowers {
-     my $result = [];
-     my $dbh = DBI->connect('dbi:mysql:webshop', '', '');
-
10     my $sql = qq{ SELECT name, price FROM flowers };
-     my $sth = $dbh->prepare($sql);
-     $sth->execute();
-     my ($name, $price);
-     $sth->bind_columns( undef, \$name, \$price );
15
-     while($sth->fetch()) {
-         push(@$result, [$name, $price]);
-     }
-
20     $sth->finish();
-     $dbh->disconnect();
-     return $result;
- }
-
25 print "status: 200\r\n";
- print "content-type: text/xml\r\n\r\n";
-
- print "<?xml version='1.0'?>\r\n";
```

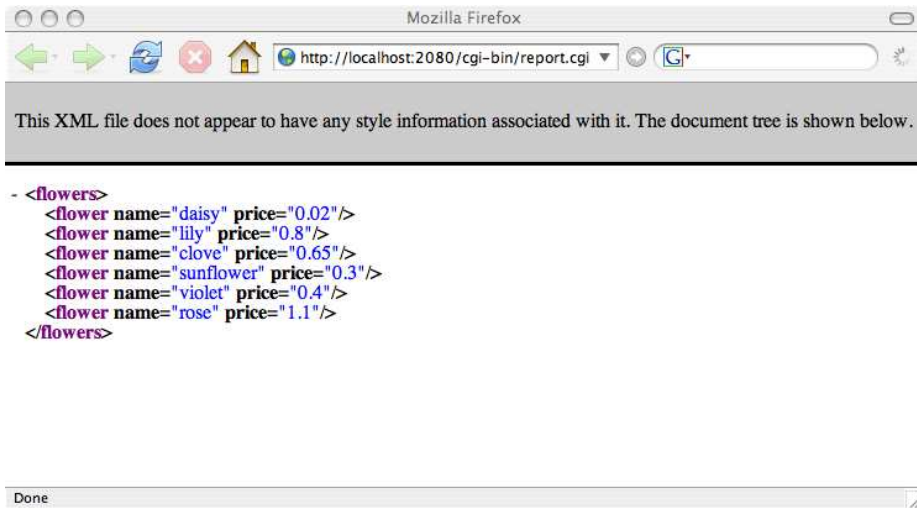


Figure 4.2: WEBrick's CGI Handler in Action

```

-   print "<flowers>\r\n";
30  my $flowers = get_flowers();
-   for (@$flowers) {
-       print "    <flower name='$_->[0]' price='$_->[1]'/>\r\n";
-   }
-   print "</flowers>\r\n";

```

To reuse such legacy scripts (do you remember why you switched to Ruby?) we set up an HTTP server as follows:

```

File
Line 1  cgi_dir = File.expand_path('./cgi-bin')
-       s = HTTPServer.new(:Port => 2080)
-       s.mount('/cgi-bin', HTTPServlet::FileHandler, cgi_dir,
-             :FancyIndexing => true
5       )

```

We installed a `FileHandler` that manages the content of the `cgi-bin` directory. Whenever you request a resource ending with `.cgi` WEBrick delegates the request to `CGIHandler`, executes the corresponding program, and returns its output. (You'll learn what the `FancyIndexing` option is in Section 4.2, *Hiding Little Secrets*, on page 171).

The CGI program can set the final HTTP status code by printing the status header (as we did in line 25 of the Perl program).

For the sake of completeness, let's look at the `ERBHandler`. This is

called by the `FileHandler` when a resource ending with `.html` is requested. Before the requested file is sent back, it gets processed by the templating engine ERB. To learn the scoop about `.html` files, have a look at *Agile Web Development With Rails* [?].

## WEBricklets

You do not have to develop huge applications to benefit from WEBrick. Often little scripts—let’s call them *WEBricklets*—can significantly improve *WEBricklets* your life.

For example, when developing Java software for PragBouquet, I often work on a remote host using a Secure Shell. I do this because the host is extremely powerful and it has the same Java environment as the final production system. My only development tools there are Vim<sup>8</sup> and Ant<sup>9</sup>. I write a lot of unit tests and I use Ant’s `junit` task to execute them. This task produces nicely formatted HTML pages. In the past I had to copy these pages to my PC using `scp` in order to view them. Thanks to WEBrick these times are long gone. I now run a mini WEBrick-based server on the remote host which allows me to view the pages using the web browser running on my PC.<sup>10</sup>

File

```

Line 1  require 'webrick'
-       include WEBrick
-
-       dir = Dir::pwd
5       port = 13000 + (dir.hash % 1000)
-
-       puts "URL: http://#{Socket.gethostname}:#{port}"
-
-       s = HTTPServer.new(
10      :Port => port,
-       :DocumentRoot => dir,
-       :ServerType => Daemon
-       )
-
15      trap('INT') { s.shutdown }
-       s.start

```

These 16 lines of Ruby code start a HTTP server on a vaguely random port and provide access to all files in the current directory. The script

---

<sup>8</sup><http://www.vim.org>

<sup>9</sup><http://ant.apache.org>

<sup>10</sup>The basis for this example was originally posted by Jim Weirich under <http://onestepback.org/index.cgi/Tech/Ruby/WEBrick.rdoc>



turns itself into a daemon process. After you run it in the directory the junit task's output goes to, it will serve your test results until you explicitly kill it.

In line 4 we detect the current directory (which is used as the server's document root) and in line 5 a new port is calculated. Then, the file server's URL is printed to the console. Usually, I paste it to my browser's bookmark manager and give it a reasonable name like "JUnit results of project X". It can happen that the program calculates a port that is already in use by another application. In this case set the port variable temporarily to a constant value and start the server again.

Beginning in line 9, we create and initialize a new HTTPServer object and pass it two new options. We set DocumentRoot to the current directory, mapping that directory to the URL /. Behind the scenes this option installs an object of class HTTPServlet::FileHandler whose do\_GET() method returns every file requested (after determining its MIME type and setting the content-type header correctly). Option ServerType is set to "Daemon" which—unsurprisingly—turns the server into a daemon (you can learn how to turn your own scripts into a daemon by intensively studying some of H.P. Lovecraft's books at full moon or by having a look at Section 6.3, *Creating Daemons and Services*, on page 269).

## Hiding Little Secrets

For files such as unit test results, the solution in the preceding section is appropriate, because the files' contents usually contain no secrets (your unit tests results are running constantly at a 100% success rate anyway, aren't they?). As we all know, being paranoid does not mean they aren't after us, so it sometimes makes sense to plug a little security layer to our software. In the case of our file server we could add HTTP basic authentication.

File

```
Line 1  require 'webrick'
-      include WEBrick
-
-      dir = Dir::pwd
5      port = 13000 + (dir.hash % 1000)
-      puts "URL: http://#{Socket.gethostname}:#{port}"
-
-      authenticate = Proc.new do |req, res|
-          HTTPAuth.basic_auth(req, res, "") do |usr, pwd|
10         usr == 'maik' && pwd == 'secret'
-      end
-      end
```

```

-
-   s = HTTPServer.new(:Port => port, :ServerType => Daemon)
15  s.mount('/', HTTPServlet::FileHandler, dir,
-     :FancyIndexing => true,
-     :HandlerCallback => authenticate
-   )
-
20  trap('INT') { s.shutdown }
-   s.start

```

Lines 8 to 12 define our authentication logic by creating a Proc object that uses method `basic_auth(request,response,realm)`<sup>11</sup> of class `HTTPAuth`. This method expects a code block and passes it the user name and password that have been transmitted with the current request. If the authentication fails—`basic_auth()` returns false—WEBrick automatically returns the HTTP status code 401 and a corresponding HTML error page. If authentication succeeds, WEBrick continues as usual.

Now we have to install the authentication handler. We can't use `mount_proc()` any longer, as it doesn't allow us to install any hooks. Instead, we have to use a servlet. Fortunately, for our purposes it's not necessary to define a new one—we can use the `HTTPServlet::FileHandler` class. We add it to the server using `mount()` and pass several options:

- `FancyIndexing` defines how directory listings will be handled. If the requested URI refers to a directory and this option is false (the default), the HTTP status code 403 (Forbidden) is returned. Otherwise, a directory listing will be displayed.
- `HandlerCallback` points to a code block that will be called before the `do_XXX` method belonging to the current request is called.

This solution is far from being perfect. It would only increase security if you put the script into your home directory and grant read permission to no one but yourself. WEBrick offers more advanced security mechanisms, including support for the same password files the Apache web server<sup>12</sup> uses. These files are usually created with the `htpasswd` command:

```

mschmidt:/tmp> htpasswd -cdb /tmp/test.pwd scott tiger
Adding password for user scott
mschmidt:/tmp> cat /tmp/test.pwd
scott:Erw4v9nQMwHQ
mschmidt:/tmp>

```

<sup>11</sup>The realm parameter is currently ignored.

<sup>12</sup><http://www.apache.org>

Here we created a file called `test.pwd` that contains the encrypted password of the user named `scott`. To use this password database with WEBrick, only the `authenticate()` method is redefined:

File

```
Line 1  authenticator = HTTPAuth::BasicAuth.new(
-       :UserDB => HTTPAuth::Htpasswd.new('/tmp/test.pwd'),
-       :Realm => ""
-     )
5
-   authenticate = Proc.new do |req, res|
-     authenticator.authenticate(req, res)
-   end
```

Instead of hard-wiring user names and passwords into your script, they will be read by `HTTPAuth::Htpasswd` and evaluated by `HTTPAuth::BasicAuth`.

You can increase the level of security even more if you use `htdigest` to create passwords. In `authenticate()` replace `HTTPAuth::Htpasswd` by `HTTPAuth::Htdigest` and `HTTPAuth::BasicAuth` by `HTTPAuth::DigestAuth`. Unfortunately, all these mechanisms implement only weak security. For something stronger, use HTTPS right from the beginning:

File

```
require 'webrick'
require 'webrick/https'
include WEBrick

dir = Dir::pwd
port = 13000 + (dir.hash % 1000)
puts "URL: http://#{Socket.gethostname}:#{port}"

authenticate = Proc.new do |req, res|
  HTTPAuth.basic_auth(req, res, "") do |usr, pwd|
    usr == 'maik' && pwd == 'secret'
  end
end

s = HTTPServer.new(
  :Port => port,
  :ServerType => Daemon,
  :SSLEnable => true,
  :SSLVerifyClient => :OpenSSL::SSL::VERIFY_NONE,
  :SSLCertName => [ %w(C US), %w(O PragBouquetSSL.com), %w(CN WWW) ]
)
s.mount(
  '/',
  HTTPServlet::FileHandler,
  dir,
  :FancyIndexing => true,
  :HandlerCallback => authenticate
)
```

```
trap('INT') { s.shutdown }
s.start
```

Here we have modified our first version and added support for SSL. We had to require `webrick/https` and we had to slightly modify the initialization of our server in lines 18 to 20. For a production system you'd need to get a valid certificate, but we only wanted to demonstrate the technical details.

## Conclusion

HTTP gets abused in countless ways. Because these ways weren't even considered by its inventors, implementations that bend HTTP stray outside of HTTP's design. (There's even a whole RFC called *On the use of HTTP as a Substrate*<sup>13</sup> dealing with this issue). Sometimes it's unnecessary to add a HTTP layer to your program, but often it is highly convenient. It may even open your application up to completely unexpected purposes.

In the end it doesn't really matter whether HTTP has been a good choice for a particular application or not. If you have to integrate with an existing HTTP service, you have to use HTTP, too. As you have seen, Ruby supports you in any imaginable way.

Alternatively, if you do want to add an HTTP interface to your own application, WEBrick will prove to be an invaluable companion.

In terms of HTTP support, Ruby is more than ready for prime time.

---

<sup>13</sup><http://www.faqs.org/rfcs/rfc3205.html>

# Distributed Applications with RPC

---

In the beginning, distributed applications seemed to be some kind of magic to many people, because you had to use rather obscure standards like Sun's Remote Procedure Call (RPC) <sup>1</sup> protocol to make two processes talk to each other.

Today countless protocols, technologies, products and standards exist for inter-process communication (IPC). Most of them—such as CORBA, XML-RPC, and SOAP—are language-neutral and bindings are available for many programming languages. Still there are some protocols that do only work with certain programming languages like Java's *Remote Method Invocation* (RMI) or Distributed Ruby (DRb).

*Remote Method  
Invocation*

Theoretically, though, when building an architecture from scratch, there are many options to choose from (XML-RPC, CORBA, RMI, REST, SOAP, etc.) and chances are good that the company you are working for uses at least two third of all protocol standards available world-wide. Ruby supports many of them and in the following sections we will demonstrate the usage of those that are most frequently used today.

### 5.1 Another Day, Another Protocol

After studying the market intensively for several years, PragBouquet's marketing department made an astonishing observation: people often add personalized greeting cards to bunches of flowers!

To satisfy this unexpected requirement on a technical level, the following solution has been implemented: in the web shop users can choose

---

<sup>1</sup><http://www.faqs.org/rfcs/rfc1057.html>

an image from a list, write a short text, and then the text is printed onto the back of the greeting card in a nice font.

In the past people often got some strange texts on their cards, because their love letters have been too long and the old solution did not provide any feedback. Hence, PragBouquet decided to separate the image production process into a server and to add some more logic to it. The web shop client requests the front and back page of the greeting card from the server and presents them to the user. If everything is fine, the client will ask the server to print the card. Otherwise, it will be deleted.

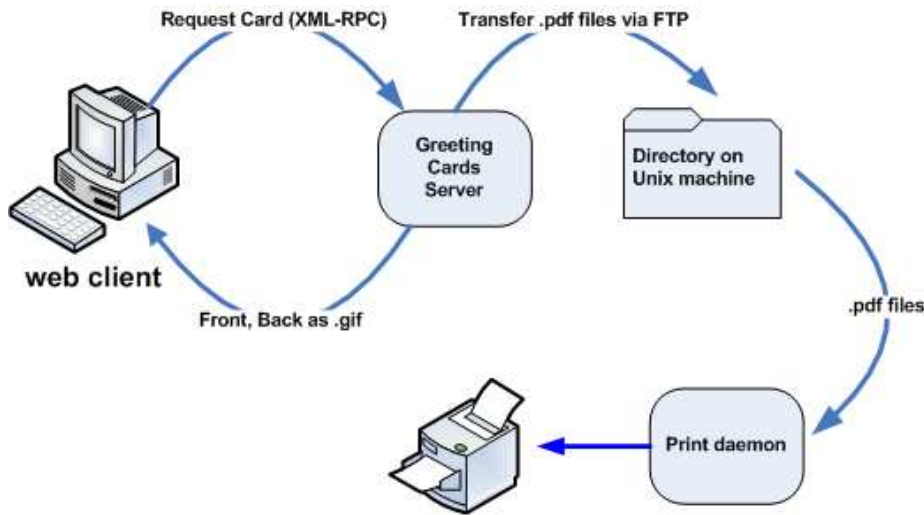
At the time the current greeting card system was built, the responsible chief software evangelist was convinced that XML-RPC will last forever. Hence, he insisted on plugging an XML-RPC interface to nearly everything during his short employment period. One of the last relicts is a C++ server that handles all things related to the creation of greeting cards. This piece of software is a pain in the neck. It only runs on Windows NT 4, and since the order volume increased significantly at PragBouquet it cries for Dr. Watson two times a week without an obvious reason. Not to forget that while trying to make it work with a newer, faster laser printer, two people got mad.

Parts of the source code got lost somehow and the company that has developed the current solution does not exist any longer. No one has been able to find the programmer who has originally created the server (you hope that he has to make a living by creating micro-code for toast-ers today) and consequentially it has been decided to re-implement it, at least partially.

The new system still has to support the old XML-RPC interface, because the web shop guys are not able to change their software soon enough. The server only has to care about the image manipulation, because for printing the cards, another system will be built. It only has to transfer the cards to be printed as .pdf files (comprising two pages) into a certain directory where they are picked up by the new printer process (see Figure 5.1, on the following page).

Luckily, we have found a README file that accurately describes the procedures supported by the server:

- `draw_card(order_no,image_id,text)` draws a greeting card, i.e. it creates images in GIF format for the front and the back pages. It returns the image data and a unique card reference that can be used to identify the card later. To identify a greeting card and to




---

Figure 5.1: Greeting Card Architecture

---

put it to the right bouquet, the client has to transmit the according order number as first parameter. All images used for greeting cards at PragBouquet have an id that has to be passed as the `image_id` parameter. `text` contains the text that should appear on the card.

- `print_card(cardref)` sends the greeting card identified by `cardref` to the printer.
- `delete_card(cardref)` deletes the greeting card identified by `cardref`.

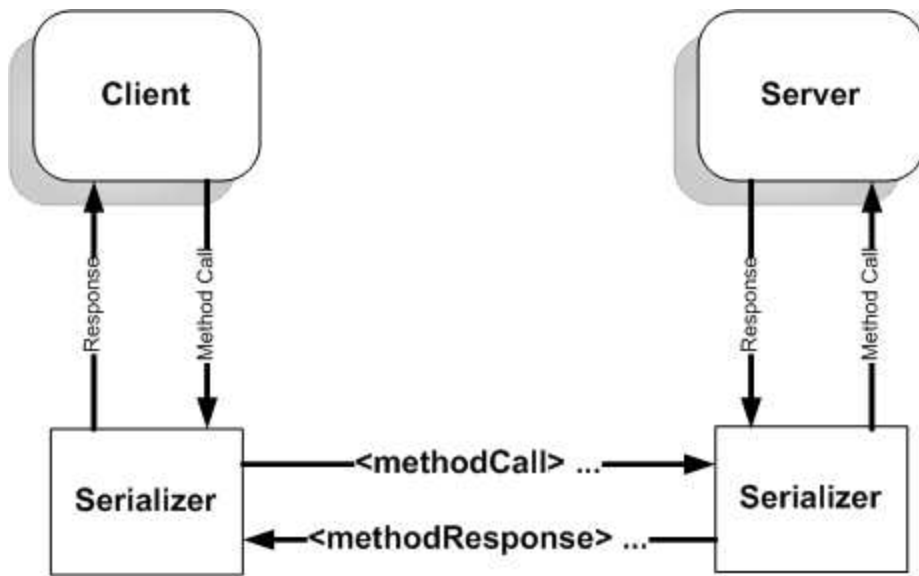
Before implementing the interface above, we will briefly describe XML-RPC. If you are already familiar with its bowels, you can safely skip the next section.

### XML-RPC In Less Than 8 Minutes

Shortly after XML was born, a lot of people obsessively tried to represent everything using `<` and `>` symbols. So it came as no surprise that in 1998/1999 a protocol for Remote Procedure Calls based on XML has been defined by Dave Winer. Running out of creativity after a long and exhausting specification process its inventor called it XML-RPC<sup>2</sup>.

---

<sup>2</sup><http://www.xmlrpc.com>




---

Figure 5.2: XML-RPC Architecture

---

Simply put, XML-RPC works like any other RPC protocol, but under the hood it is a lot simpler than its binary brothers and sisters. Implementing your own XML-RPC client will not take much more than a black and boring winter day, I guess, but smart people have already done it for you. So, better spend such a day reading trivial stuff while having a good cup of tea.

From an architectural point of view XML-RPC is a synchronous protocol where every request is followed by a response. Only two types of messages, method calls and method responses, are supported and both are encoded as XML documents that get transferred on a HTTP layer (see Figure 5.2).

When we, e.g., call a method to draw a greeting card having the Id 42 and the message text on the back of the card (`draw_greeting_card(4711, 42, 'Please, forgive me, honeybunch!')`) its XML-RPC representation will look like this:

File

```

Line 1  <methodCall>
-       <methodName>draw_card</methodName>
-       <params>
-         <value><int>4711</int></value>

```



```

5      <value><int>42</int></value>
-      <value>
-          <string>Please, forgive me, honeybunch!</string>
-      </value>
-  </params>
10 </methodCall>

```

The according response contains an image file encoded in base-64 and additionally indicates the type of the file, i.e. GIF, JPEG, PNG, and so on.

File

```

Line 1 <methodResponse>
-   <params>
-       <value><string>4711_006754</string></value>
-       <value><string>GIF</string></value>
5       <value>
-           <base64>
-               VGhpcyBpcyBhIHJhdGhlciBsb25nIHRleHQgSSBo
-               YXZlIHdyaXR0ZW4gdG8gc2ltdWxhdGUGYSBwaWN0
-               dXJlLiBPZiBjb3Vyc2UsIEkgZGlkIG5vdCB3YW50
10              IHRvIGZha2UgYSBiYXNlLTlTY0IHN0cmLuZyBhbmQg
-               c28gSSBoYXZlIHVzZWQgUnVieSB0byBidWlsdCBv
-               bmUgZnJvbSB0aGlzIHJlYWxseSBzdHVwaWQgYW5k
-               IHVzZWxlcnMgdGV4dC4=
-           </base64>
15      </value>
-      <value><nil/></value>
-   </params>
- </methodResponse>

```

XML-RPC supports a fixed set of simple and compound data types (see Figure 5.3, on the following page for a list of all simple data types). Due to the nature of XML all parameter values are tagged and their values are transmitted as strings. For example, a double value like -3.14 is encoded as `<double>-3.14</double>`.

In addition, two compound types are available: arrays and structs. Contrary to array implementations in most programming languages, XML-RPC arrays aggregate a list of elements that must not all be of the same type. Arrays may contain simple data types, structs, and other arrays. For example, `[1, 2, 3]` is represented like this:

```

<value>
  <array>
    <value><int>1</int></value>
    <value><int>2</int></value>
    <value><int>3</int></value>
  </array>
</value>

```

| Type     | Values                                                                               | Tag                |
|----------|--------------------------------------------------------------------------------------|--------------------|
| Integer  | 32-bit signed integer                                                                | <i4> or <int>      |
| Double   | 64-bit IEEE 75 floating-point number                                                 | <double>           |
| Boolean  | false (0) or true (1)                                                                | <boolean>          |
| String   | ASCII string (may contain NULL bytes). Current implementations support Unicode, too. | <string>           |
| Base64   | Binary string encoded in Base64 <sup>3</sup>                                         | <base64>           |
| DateTime | Absolute specification of date and time without time zone information                | <dateTime.iso8601> |
| nil      | Null value (this an unofficial extension to the standard)                            | <nil>              |

---

Figure 5.3: XML-RPC Simple Data Types

---

Structs contain a list of key/value pairs where values might be simple data types, arrays, or structs. The following represents the Hash { 'name' => 'maik', 'age' => 32 }:

```
<value>
  <struct>
    <member>
      <name>name</name>
      <value><string>Maik</string></value>
    </member>
    <member>
      <name>age</name>
      <value><int>32</int></value>
    </member>
  </struct>
</value>
```

As a response to a method call, an exception (XML-RPC calls them *faults*) can be raised. Because they can only occur instead of a method

response, they are encoded in the same way:

File

```

Line 1  <methodResponse>
-       <fault>
-         <value>
-           <struct>
5         <member>
-           <name>faultCode</name>
-           <value><int>7777</int></value>
-         </member>
-         <member>
10        <name>faultString</name>
-        <value>
-          <string>Text too short for an apology!</string>
-        </value>
-      </member>
15      </struct>
-    </value>
-  </fault>
- </methodResponse>

```

Fault codes have not been standardized, but several XML-RPC implementations use small numbers to indicate errors on the lower transport layers, so better use numbers bigger than 1000 for your own codes.

All these details should only deepen your understanding of the underlying technology, because for an application developer most of these things happen completely transparently. Usually, you won't even notice that you're calling a procedure on a remote host. An XML-RPC library handles all the low-level stuff, turns method calls into XML documents, transfers these documents via HTTP to a server, reads its XML response, turns the response into a data structure or an exception, and finally returns the result to the caller.

## xmlrpc4r

Kudos to Michael Neumann for contributing *xmlrpc4r*—one of the most xmlrpc4r complete and most advanced XML-RPC implementations currently available—to the Ruby standard library.

We will use it in the following to develop the new greeting cards server and its according test client. Before diving into the details of *xmlrpc4r* let's create a skeleton class for representing two-sided greeting cards:

File

```

Line 1  class GreetingCard
-      attr_reader :card_ref, :front, :back
-
-      class << self
5        def create(order_no, image_id, text)

```

```

-         front = read_image(image_id)
-         back = create_text_page(text)
-         card_ref = order_no.to_s + "_" + rand(1000).to_s
-         store_card(card_ref, front, back)
10      GreetingCard.new(card_ref, front, back)
-
-      end
-
-      private
-
15      def read_image(image_id)
-        IO.read("img/#{image_id}.gif")
-      end
-
-      def create_text_page(text)
20        return nil unless text
-        # Here we have to create the nice-looking
-        # text page somehow.
-        ""
-      end
-
25      def store_card(card_ref, front, back)
-        # Generate PDF and store cardref.pdf
-        # containing front and back images.
-      end
30    end
-
-    private
-
-    def initialize(card_ref, front, back)
35      @card_ref, @front, @back = card_ref, front, back
-    end
-  end
end

```

A GreetingCard has only three attributes: an unique reference (cardref), a front page (front) and a back page (back). A factory method called create(image\_id,text) creates new GreetingCard objects.

Current versions of xmlrpc4r are based on WEBrick and the following server should look familiar:

```

File
Line 1  require 'webrick'
-      require 'xmlrpc/server'
-      require 'net/ftp'
-      require 'greeting_card'
5
-      s = XMLRPC::WEBrickServlet.new
-      s.add_handler('pragbouquet.draw_card') do |order_no, id, text|
-        card = GreetingCard.create(order_no, id, text)
-        {
10        'cardref' => card.card_ref,
-        'front' => XMLRPC::Base64.encode(card.front),

```

```

-         'back' => XMLRPC::Base64.encode(card.back)
-     }
- end
15
- s.add_handler('pragbouquet.print_card') do |cardref|
-     # Transfer cardref.pdf to special directory, where
-     # it gets picked up and is printed eventually
-     ftp = Net::FTP.new('ftp.imagehost')
20     ftp.login('imageserver', 'secret')
-     files = ftp.chdir('pub/cards/img')
-     ftp.putbinaryfile("#{cardref}.pdf")
-     ftp.close
- end
25
- s.add_handler('pragbouquet.delete_card') do |cardref|
-     File.delete("#{cardref}.pdf")
- end
-
30 s.set_default_handler do |name, *args|
-     raise XMLRPC::FaultException.new(
-         -100,
-         "Method #{name} missing or wrong number of parameters!"
-     )
35 end
-
- server = WEBrick::HTTPServer.new(:Port => 8080)
- server.mount('/RPC2', s)
- trap('INT') { server.shutdown }
40 server.start

```

We have created a servlet of class `XMLRPC::WEBrickServlet` and added handlers for the `draw_card()`, `print_card()`, and `delete_card()` functions. It's not demanded by the specification, but it's good style to give them all a prefix ("pragbouquet" in our case) and it will pay off later.

`draw_card()`'s implementation is straight-forward using the `GreetingCard` class. Note that in lines 11 and 12 we had to explicitly convert our images into base-64 format. The handlers for `print_card()` and `delete_card()` do not produce anything useful at the moment, but at least you can test if they are available.

In line 30 we have installed a so called default handler that is comparable to Ruby's `method_missing()` method. It gets called whenever the server receives a request it cannot dispatch for any reason, i.e. because the method name is unknown or the amount of method parameters does not match. In these cases an exception is raised, i.e. a fault response will be returned.

The rest of the server does not differ from initializing and starting

any other WEBrick server. By loose convention, XML-RPC services are located under /RPC2, so we mount our servlet there.

Alright, the XML-RPC service is listening on port 8080 under the URL/RPC2. The following client will test if it works as expected:

File

```

Line 1  require "xmlrpc/client"
-
-  server = XMLRPC::Client.new('localhost', '/RPC2', 8080)
-  begin
5      response = server.call(
-      'pragbouquet.draw_card',
-      4711,
-      42,
-      'Forgive me!'
10  )
-  rescue XMLRPC::FaultException => e
-      puts "An error occurred:"
-      puts e.faultCode
-      puts e.faultString
15  end

```

xmlrpc4r's usage is intuitive. To obtain a reference to the server you have to create an instance of XMLRPC::Client specifying the host, the path, and the port the service is listening on. Calling a particular procedure is performed by call(procedure,\*args), then. Its result is the decoded server response.

As we all know there is no Ruby code that could not be made shorter and more expressive:

File

```

Line 1  require "xmlrpc/client"
-
-  server = XMLRPC::Client.new('localhost', '/RPC2', 8080)
-  gc = server.proxy2('pragbouquet')
5  success, cards = gc.draw_card(4711, 42, 'Forgive me!')
-  if success
-      puts cards
-  else
-      puts "An error occurred:"
10      puts cards.faultCode
-      puts cards.faultString
-  end

```

In line 4 we have created a proxy for the greeting card service that allows us to use the remote procedures exactly as they were local ones (here we needed the prefix “pragbouquet” again). Furthermore, we have suppressed the exception feature, i.e. calling proxy2() makes all remote procedures return a boolean value indicating success or failure. If we

had used `proxy()`, exceptions would be raised. The same applies to `call()` and `call2()`.

In addition to the WEBrick servlet you can choose from a variety of different server types that can be configured by several options:

- Server is a stand-alone server that does not depend on WEBrick or anything else.
- CGIServer can be used in a CGI environment, i.e. you can put your XML-RPC implementation into a CGI script that is executed by a web server, whenever it receives an XML-RPC request. To prevent the web server from executing the script over and over again, this server type works with FastCGI<sup>4</sup>, too.
- ModRubyServer allows you to embed your XML-RPC procedures into a web server using `mod_ruby`<sup>5</sup>.

## Conclusion

XML-RPC is fairly popular, e.g., it is supported by many operating systems like Apple's Mac OS X, and it builds the basis for services like the Blogger API<sup>6</sup>. Its biggest strength compared to competitors like SOAP or CORBA is its simplicity. When designing a distributed application, XML-RPC is always worth a look. Especially, when you're using Ruby, because `xmlrpc4r` is extremely powerful and gives you a variety of options for making your remote procedures available to the public quickly.

## 5.2 We Will Take No REST, Will We?

Remember the LDAP address book we implemented in Section 2.4, *Lightweight Directory Access Protocol (LDAP)*, on page 51? We did it because the web shop guys wanted to give *Ruby On Rails* a try. Because of some unexpected change requests (what else?), they didn't have enough time to work their way through Ruby and the Rails framework, but the address book is still needed. They know that you've already set up an LDAP server, and that your Ruby classes are working, too. So they asked you to turn the implementation into a HTTP service that they can access from their Java classes. Although the web shop team said that the

---

<sup>4</sup><http://www.fastcgi.com>

<sup>5</sup><http://www.modruby.net>

<sup>6</sup>[http://www.blogger.com/developers/api/1\\_docs](http://www.blogger.com/developers/api/1_docs)

details of the interface do not matter much, you know that they'd prefer XML.

You decide that it would be a good idea to map the address book operations (create a new address book, show an address book, add an entry to an existing address book, modify an entry, delete an entry, and delete an address book) to different HTTP methods. To show an address book, for example, we could use the HTTP GET method. To delete an entry or a complete address book, we can use an HTTP DELETE. We'll pass parameters as XML documents.

Let's create the main infrastructure first: a WEBrick server that will be the basis for our service:

File

```

Line 1  require 'rexml/document'
-       require 'addressbook'
-       require 'ldap'
-       require 'thread'
5       require 'cgi'
-       require 'webrick'
-       include WEBrick
-       class AddressBookServlet < HTTPServlet::AbstractServlet
-         @@instance = nil
10      @@instance_creation_mutex = Mutex.new
-
-       def self.get_instance(config, *options)
-         @@instance_creation_mutex.synchronize do
-           @@instance ||= self.new(config, *options)
15      end
-       end
-
-       def initialize(config, ldap_conn)
-         super
20      @ldap_conn = ldap_conn
-       @ldap_mutex = Mutex.new
-       end
-       end
-       connection = Conn.new
25      connection.set_option(LDAP_OPT_PROTOCOL_VERSION, 3)
-       connection.bind('cn=root,dc=pragbouquet,dc=com', 'secret')
-
-       server = HTTPServer.new(:Port => 4242)
-       server.mount('/ab', AddressBookServlet, connection)
30
-       %w(INT TERM).each do |signal|
-         trap(signal) { server.shutdown }
-       end
-
35      server.start

```



This server differs a bit from the servers we have used so far. The servlet is a bit more complicated because we're working with an expensive resource, the connection to the LDAP server. By default, a new servlet instance will be created for every request. This is convenient, because you do not have to care about multithreading issues, but it can waste a lot of resources, too.

The WEBrick inventors were aware of this problem, so they provided the `get_instance(config,*options)` method. This gives you complete control the servlet creation process. We used it to implement a singleton pattern that creates only a single instance of the `AddressBookServlet` class, but you can come up with any resource management strategy you prefer or need. For example, you could instead implement a servlet pool.

Our solution is much simpler. In line 13 we synchronize the creation of the one and only servlet instance (to prevent a race condition when two requests arrive simultaneously before the instance is created) and in the following line we actually create a new instance if it does not exist already.

For the first time we've defined an `initialize()` method for our servlet. We had to do this because we want to configure it before we actually mount it. In line 29 we pass it the LDAP connection. In addition, we create a `Mutex` object in line 21 that we will use for synchronizing access to the LDAP server.

Initializing the server is business as usual. We create a connection to the LDAP server, initialize a servlet, and mount it at `/ab` (ab is an abbreviation for address book).

Before adding handlers for the different HTTP methods, we add `to_xml()` methods to the `Recipient` and `AddressBook` classes that return XML representations of their according instances.

File

```

Line 1  class Recipient
        def to_xml
        recipient = REXML::Element.new('recipient')
        recipient.add_attribute('forename', forename)
        recipient.add_attribute('surname', surname)
5       address = recipient.add_element('address')
        address.add_element('street').add_text(street)
        address.add_element('postal-code').add_text(postal_code)
        address.add_element('city').add_text(city)
10      address.add_element('state').add_text(state)
        recipient.add_element('description').add_text(description)
        recipient
        end

```

```

-   end
15
-   class AddressBook
-       def to_xml
-           addressbook = REXML::Element.new('address-book')
-           addressbook.add_attribute('uid', user.uid)
20           each_recipient { |r| addressbook.add_element(r.to_xml) }
-           addressbook
-       end
-   end

```

We use REXML to turn our objects into XML. This is convenient, as the `to_xml()` methods return `REXML::Element` objects, not strings, so we can easily add elements and attributes if we need to (and, of course, we will).

Now we have everything prepared, so let's implement the first HTTP method handler. This returns the complete address book belonging to a particular uid. Our server expects a request of the form [http://host:port/ab/\(uid\)](http://host:port/ab/(uid)), so to get the address book belonging to uid 4711, you send a GET request to <http://host:port/ab/4711>.

File

```

Line 1  class AddressBookServlet
-       # Return all entries belonging to a particular address book.
-       def do_GET(req, res)
-           res['content-type'] = 'text/xml'
5           uid = get_uid(req)
-           @ldap_mutex.synchronize do
-               begin
-                   ab = AddressBook.new(@ldap_conn, User.new(uid))
-                   entries = ab.to_xml
10                  entries.each_element('recipient') do |r|
-                      cn = to_cn(
-                          r.attributes['forename'],
-                          r.attributes['surname']
-                      )
15                  link = r.add_element('link')
-                  link.add_text(
-                      "http://#{req.host}:#{req.port}/ab/#{uid}/#{cn}"
-                  )
-               end
20                  entries.write(res.body, 0)
-                  res.body << "\n"
-                  res.status = 200
-               rescue RuntimeError
-                   res.status = 404
25               end
-           end
-       end
-   end

```

```

-   def get_uid(req)
30     req.path_info =~ /\(/(\d+)\)/ ? $1 : nil
-   end
-
-   def to_cn(forename, surname)
-     CGI.escape(forename + " " + surname)
35   end
-   end

```

Admittedly, this is a lot of code, but it's not difficult to understand. Let's dissect it, line by line. First, we extract the uid from the URL. This is done in the `get_uid()` method.

Because we need it two lines later to read the address book, in line 6 we synchronize the access to the LDAP server. Then we convert the address book into an XML element with our new `to_xml()` method.

The iterator starting on line 10 adds a `<link>` element to each address book entry (that's the reason why `to_xml()` does not return a string, but a `REXML::Element` object). This element tells the client how to refer to the entry on the server. To build the URL we append the common name (cn) of the entry to the address book's URL and therefore we have to url-encode it in the `to_cn()` method.

Eventually, we create a nicely formatted version of our XML document in line 20 and send it back to the client. If we could not find the address book requested, we leave the body empty and set the status to *File Not Found* in line 24.

Let's test the method with the command line tool `curl` (the `-i` option instructs `curl` to print the HTTP headers, too).

```

mschmidt:/tmp> curl -i http://localhost:4242/ab/4711
HTTP/1.1 200 OK
Connection: Keep-Alive
Date: Sun, 13 Nov 2005 09:02:52 GMT
Content-Type: text/xml
Server: WEBrick/1.3.1 (Ruby/1.8.3/2005-09-21)
Content-Length: 739

<address-book uid='4711'>
  <recipient forename='P.H.' surname='Beans'>
    <address>
      <street>Nuclear Powerplant Road 1</street>
      <postal-code>65801</postal-code>
      <city>Springfield</city>
      <state>MO</state>
    </address>
    <description>My boss.</description>
  </recipient>
</address-book>

```

```

    <link>http://localhost:4242/ab/4711/P.H.+Beans</link>
  </recipient>
  <recipient forename='Marge' surname='Jackson'>
    <address>
      <street>Evergreen Terrace 42</street>
      <postal-code>62701</postal-code>
      <city>Springfield</city>
      <state>IL</state>
    </address>
    <description>
      Don&apos;t forget our wedding anniversary!
    </description>
    <link>http://localhost:4242/ab/4711/Marge+Jackson</link>
  </recipient>
</address-book>

```

Everything looks good, but you may ask yourself, what all this has to do with REST—*Representational State Transfer*?<sup>7</sup> Isn't it an ordinary HTTP service? Yes, it is! The main thing that differs from the services we have built so far is the use of URLs. They do not contain actions such as *send\_sms* or *store\_address*, but they refer to resources such as address books on the server.

*Representational State  
Transfer*

Another difference with REST is that we no longer use just GET and POST requests. To delete an address book entry or a complete address book, we use the DELETE method, for example:

File

```

Line 1  class AddressBookServlet
-      def do_DELETE(req, res)
-        if req.path_info =~ /\((\d+)\)(\.(.+))$/
-          delete_recipient($1, $2)
5      elsif req.path_info =~ /\((\d+)\)$/
-        delete_address_book($1)
-      end
-      res.status = 204 # No Content
-    end
10
-    def delete_recipient(uid, common_name)
-      @ldap_mutex.synchronize do
-        ab = AddressBook.new(@ldap_conn, User.new(uid))
-        cn = CGI.unescape(common_name)
15      surname = cn.split(/.* (\w+)\$/)[1]
-        forename = cn[0 .. -(surname.size + 1)]
-        ab.remove(Recipient.new(forename, surname))
-      end
-    end
20

```

<sup>7</sup>[http://en.wikipedia.org/wiki/Representational\\_State\\_Transfer](http://en.wikipedia.org/wiki/Representational_State_Transfer)

```

-   def delete_address_book(uid)
-       @ldap_mutex.synchronize do
-           ab = AddressBook.new(@ldap_conn, User.new(uid))
-           ab.each_recipient { |r| ab.remove(r) }
25          ab.delete
-       end
-   end
- end

```

At the beginning of the `do_DELETE(req,res)` method we check if we have to delete a single address book entry or a complete address book. Depending on the URL format we delegate work to the `delete_address_book(uid)` or to the `delete_recipient(uid,common_name)` method. In lines 14 to 16 we extract the forename and surname of the recipient to be deleted from the common name passed in the URL.

The following session first demonstrates how to delete Marge Jackson from the address book belonging to user 4711 and then how to delete the entire address book (curl's `-X` option lets us specify which HTTP request to use).

```

mschmidt:/tmp> curl -i http://localhost:4242/ab/4711/Marge+Jackson \
> -X DELETE
HTTP/1.1 204 No Content
Date: Sun, 13 Nov 2005 13:20:42 GMT
Server: WEBrick/1.3.1 (Ruby/1.8.3/2005-09-21)

```

```

mschmidt:/tmp> curl -i http://localhost:4242/ab/4711
HTTP/1.1 200 OK
Connection: Keep-Alive
Date: Sun, 13 Nov 2005 13:24:02 GMT
Content-Type: text/xml
Server: WEBrick/1.3.1 (Ruby/1.8.3/2005-09-21)
Content-Length: 373

```

```

<address-book uid='4711'>
  <recipient forename='P.H.' surname='Beans'>
    <address>
      <street>Nuclear Powerplant Road 1</street>
      <postal-code>65801</postal-code>
      <city>Springfield</city>
      <state>MO</state>
    </address>
    <description>My boss.</description>
    <link>http://localhost:4242/ab/4711/P.H.+Beans</link>
  </recipient>
</address-book>

```

```

mschmidt:/tmp> curl -i http://localhost:4242/ab/4711 -X DELETE
HTTP/1.1 204 No Content

```

```
Date: Sun, 13 Nov 2005 13:20:55 GMT
Server: WEBrick/1.3.1 (Ruby/1.8.3/2005-09-21)
```

```
mschmidt:/tmp> curl -i http://localhost:4242/ab/4711
HTTP/1.1 404 Not Found
Connection: Keep-Alive
Date: Sun, 13 Nov 2005 13:24:40 GMT
Content-Type: text/xml
Server: WEBrick/1.3.1 (Ruby/1.8.3/2005-09-21)
Content-Length: 0
```

It looks like our delete methods work. Now let's balance the books by creating new entries.

File

```
Line 1  class AddressBookServlet
-      def do_POST(req, res)
-          doc = REXML::Document.new(req.body)
-          if doc.root.name == 'recipient'
5              create_recipient(doc, req, res)
-          else
-              create_address_book(doc, req, res)
-          end
-      end
10
-      def create_address_book(doc, req, res)
-          user = parse_user(doc)
-          @ldap_mutex.synchronize do
-              AddressBook.create(@ldap_conn, user)
15          end
-          res.status = 201
-          res['Location'] = "/ab/#{user.uid}"
-      end
-
20      def create_recipient(doc, req, res, update = false)
-          uid = get_uid(req)
-          recipient = parse_recipient(doc)
-          @ldap_mutex.synchronize do
-              begin
25                  ab = AddressBook.new(@ldap_conn, User.new(uid))
-                  if update
-                      ab.modify(recipient)
-                  else
-                      ab.add(recipient)
30                  end
-                  cn = to_cn(recipient.forename, recipient.surname)
-                  res.status = 201 # Created
-                  res['Location'] = "/ab/#{uid}/#{cn}"
-              rescue RuntimeError => ex
35                  @logger.info ex
-                  res.status = 404
-          end
-      end
```

```

-     end
- end
40
- def parse_user(doc)
-   attributes = doc.root.attributes
-   User.new(
-     attributes['uid'],
-     attributes['forename'],
45     attributes['surname']
-   )
- end
-
50 def parse_recipient(doc)
-   attributes = doc.root.attributes
-   Recipient.new(
-     attributes['forename'],
-     attributes['surname'],
55     attributes['street'],
-     attributes['postal-code'],
-     attributes['city'],
-     attributes['state'],
-     attributes['description']
60   )
- end
- end

```

New objects are created using the POST request. First, we check if a new address book or a new address book entry should be created. In both cases we read the attributes of the object to be created from an XML document that is passed with the request. Attributes for a new address book will be read by `parse_user(doc)` and attributes for a new address book entry will be read by `parse_recipient(doc)`.

The `create_recipient(doc, req, res, update=false)` is a bit special. It's not only able to create new recipient objects, but it can also modify existing ones. You can change its behaviour using the update flag.

Both methods, `create_address_book()` and `create_recipient()`, set the Location header in the HTTP response. This contains a link to the object that we just created. This is a REST convention.

The following session creates a new address book and a new address book entry:

```

mschmidt:/tmp> curl -i http://localhost:4242/ab/23 -X POST -d \
> "<address-book uid='23' forename='Homer' surname='Simpson'/>"
HTTP/1.1 201 Created
Connection: Keep-Alive
Date: Sun, 13 Nov 2005 18:46:54 GMT
Server: WEBrick/1.3.1 (Ruby/1.8.3/2005-09-21)

```

```

Content-Length: 0
Location: http://localhost:4242/ab/23

mschmidt:/tmp> curl -i http://localhost:4242/ab/23 -X POST -d \
> "<recipient forename='Barney' surname='Gumble' street='Musterstr. 42'
> postal-code='11011' city='Berlin' state='n/a'
> description='My best friend.' />"
HTTP/1.1 201 Created
Connection: Keep-Alive
Date: Sun, 13 Nov 2005 18:48:46 GMT
Server: WEBrick/1.3.1 (Ruby/1.8.3/2005-09-21)
Content-Length: 0
Location: http://localhost:4242/ab/23/Barney+Gumble

mschmidt:/tmp> curl -i http://localhost:4242/ab/23
HTTP/1.1 200 OK
Connection: Keep-Alive
Date: Sun, 13 Nov 2005 18:49:41 GMT
Content-Type: text/xml
Server: WEBrick/1.3.1 (Ruby/1.8.3/2005-09-21)
Content-Length: 366

<address-book uid='23'>
  <recipient forename='Barney' surname='Gumble'>
    <address>
      <street>Musterstr. 42</street>
      <postal-code>11011</postal-code>
      <city>Berlin</city>
      <state>n/a</state>
    </address>
    <description>My best friend.</description>
    <link>http://localhost:4242/ab/23/Barney+Gumble</link>
  </recipient>
</address-book>

```

Now we can create new objects. It would be nice to be able to modify existing entries, too.

File

```

Line 1  class AddressBookServlet
-      def do_PUT(req, res)
-        doc = REXML::Document.new(req.body)
-        create_recipient(doc, req, res, true)
5      end
-    end

```

This code is short, because we could reuse the `create_recipient()` method that we used to create new entries. Note that to update existing entries we use the PUT request. If, for example, Barney becomes Homer's very best friend, we can change the description attribute as follows:

```
mschmidt:/tmp> curl -i http://localhost:4242/ab/23 -X PUT -d \
```



| CRUD Action | HTTP Method |
|-------------|-------------|
| Create      | POST        |
| Retrieve    | GET         |
| Update      | PUT         |
| Delete      | DELETE      |

---

Figure 5.4: Mapping CRUD to REST

---

```
> "<recipient forename='Barney' surname='Gumble'
> street='Musterstr. 42' postal-code='11011'
> city='Berlin' state='n/a'
> description='My very best friend.' />"
HTTP/1.1 201 Created
Connection: Keep-Alive
Date: Sun, 13 Nov 2005 19:02:07 GMT
Server: WEBrick/1.3.1 (Ruby/1.8.3/2005-09-21)
Content-Length: 0
Location: http://localhost:4242/ab/23/Barney+Gumble

mschmidt:/tmp> curl -i http://localhost:4242/ab/23
HTTP/1.1 200 OK
Connection: Keep-Alive
Date: Sun, 13 Nov 2005 19:03:46 GMT
Content-Type: text/xml
Server: WEBrick/1.3.1 (Ruby/1.8.3/2005-09-21)
Content-Length: 371

<address-book uid='23'>
  <recipient forename='Barney' surname='Gumble'>
    <address>
      <street>Musterstr. 42</street>
      <postal-code>11011</postal-code>
      <city>Berlin</city>
      <state>n/a</state>
    </address>
    <description>My very best friend.</description>
    <link>http://localhost:4242/ab/23/Barney+Gumble</link>
  </recipient>
</address-book>
```

Perfect! We now have methods for all the CRUD actions. Figure 5.4 shows how we mapped them to HTTP request types.

At the risk of destroying your faith in technology, we have a confession to make. There is no REST standard. REST is an architectural style that depends on a lot of other standards such as HTTP and XML. It

recommends representing entities on a server as XML documents that can be referred to by URLs. But that's about it. There is no such thing as a REST service framework. But, because Ruby has excellent support for HTTP and XML, it also has excellent support for REST services. In fact, maybe you've already implemented a REST service without even noticing it.

## 5.3 SOAP

Today you gave a little presentation of the `Stock` class we developed in Section 2.3, *What Do We Have In Stock?*, on page 41 to the clerks and the team that normally develops PragBouquet's financial applications. They were all very impressed, and so you've landed the job to work out the details of the final system with Jeff, the leader of the financial applications team.

Jeff told you that Microsoft's .NET environment is their preferred development platform. Processes and components on this platform often communicate via SOAP, so Jeff's life would be easier if the `Stock` class was available as a SOAP service, too. You don't have much experience with SOAP and absolutely no experience with Ruby and SOAP, but you are sure that there will be an easy way to turn the `Stock` class into a SOAP service.

### Refactoring the Stock Class API

First, we have to refactor the API of the `Stock` class a bit, because Jeff made some useful suggestions. For example, he wants to remove the `Bouquet` class, because in his opinion, a flower stock should not have to know anything about bouquets. At least it should be possible to remove flowers without putting them into a bouquet first. He also thinks that the `print_report()` method should be replaced with a `get_report()` method that actually returns a stock report instead of printing it.

After 90 minutes of discussion you both agree upon the following API:

```
create_flower(name,price,quantity)
```

creates an entry for a new flower called `name` in the database and sets an initial price and an initial quantity.

```
add_flowers(name,quantity)
```

adds quantity flowers called `name` to the stock. It returns the quantity of flowers called `name` in stock.

`remove_flowers(name,quantity)`

removes quantity flowers called name from the stock. It returns the quantity of flowers called name in stock.

`set_price(name,price)`

sets the price of the flower called name to price.

`get_report()`

returns a Hash containing a two-dimensional array for each flower in stock. The first array entry is the quantity of the flower and the second is the current price. A typical example looks like this:

```
{
  'rose' => [1000, 2.99],
  'sunflower' => [500, 1.79]
}
```

All price information is stored in US dollars. Every method can raise an exception if it gets invalid arguments or runs into a database error.

“That’s great!” you think. “I don’t have to make any changes to the database schema.”<sup>8</sup> Before we deal with any soapy things, we will drop the Bouquet class and completely refactor our Stock. If SOAP is worthy of all the hype surrounding it, we hopefully won’t have to modify our class again later. Our database access layer looks like this, then:

File

```
Line 1  require 'rubygems'
-       require 'active_record'
-
-       ActiveRecord::Base.establish_connection(
5         :adapter => 'mysql',
-         :host => '127.0.0.1',
-         :database => 'webshop'
-       )
-
10      class Flower < ActiveRecord::Base
-        belongs_to :stock_item
-      end
-
-      class StockItem < ActiveRecord::Base
15        has_one :flower
-      end
```

No big surprises here: the Bouquet class and all its dependencies are gone. Let’s implement our API specification:

File

```
Line 1  class Stock
```

---

<sup>8</sup>Still thinking the old way, eh? With ActiveRecord these changes would have been a piece of cake anyway :-)

```

-   def create_flower(name, price, quantity)
-       flower = Flower.find_by_name(name)
-       raise "#{name} already exists!" if !flower.nil?
5       si = StockItem.new(:quantity => quantity)
-       si.save
-       si.create_flower(:name => name, :price => price)
-       quantity
-   end
10
-   def add_flowers(name, quantity)
-       adjust_quantity(name, quantity)
-   end
-
15  def remove_flowers(name, quantity)
-       adjust_quantity(name, -quantity)
-   end
-
-   def set_price(name, price)
20       flower = Flower.find_by_name(name)
-       raise "#{name} is unknown!" if flower.nil?
-       flower.price = price
-       flower.save
-       price
25  end
-
-   def get_report
-       StockItem.find(:all).inject({}) do |r, si|
-           r[si.flower.name] = [si.quantity, si.flower.price]; r
30       end
-   end
-
-   private
-
35  def adjust_quantity(name, difference)
-       flower = Flower.find_by_name(name)
-       raise "#{name} is unknown!" if flower.nil?
-       si = StockItem.find(flower.stock_item_id)
-       si.quantity += difference
40       raise 'Not enough flowers!' if si.quantity < 0
-       si.save
-       si.quantity
-   end
-   end

```

As expected, the Ruby implementation almost reads like the API description itself, and ActiveRecord makes the code pure, short, and simple.

## A Look Under the Hood of SOAP

In doing some research on SOAP, you might have come across the draft W3C specification:<sup>9</sup>

“SOAP is a lightweight protocol for exchange of information in a decentralized, distributed environment. It is an XML based protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined datatypes, and a convention for representing remote procedure calls and responses.”

Simply put, SOAP is XML-RPC after it has been ground by the mills of a standards committee.<sup>10</sup> Like XML-RPC, SOAP specifies a way to implement a remote procedure call architecture where all requests and responses are encoded as XML documents. Compared to XML-RPC, it has a more flexible data type system, deals with encoding and authentication issues, and is meant to be used on different transport layers, not only HTTP.

Let’s see how it works in the real world. A method call such as `create_flower('rose', 1.99, 1000)` gets converted into the following document:

```
<?xml version="1.0" encoding="utf-8"?>
<env:Envelope xmlns:xsd="http://www.w3.org/2001/XMLSchema"
              xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <env:Body>
    <n1:create_flower xmlns:n1="urn:Stock"
                     env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <name xsi:type="xsd:string">rose</name>
      <price xsi:type="xsd:double">+1.99</price>
      <quantity xsi:type="xsd:int">1000</quantity>
    </n1:create_flower>
  </env:Body>
</env:Envelope>
```

If we leave out all the namespace stuff, it becomes easier to see how it works:<sup>11</sup>

```
<?xml version="1.0"?>
<Envelope>
```

<sup>9</sup><http://www.w3.org/TR/2000/NOTE-SOAP-20000508>

<sup>10</sup>In the beginning SOAP stood for “Simple Object Access Protocol”, but today it only means SOAP. Have a look at the current specification and you can easily see why the committee decided to drop the “Simple.”

<sup>11</sup>We only remove the namespaces for better readability. When you are actually working with SOAP, you have to use them!

```

<Body>
  <create_flower>
    <name type="string">rose</name>
    <price type="double">+1.99</price>
    <quantity type="int">1000</quantity>
  </create_flower>
</Body>
</Envelope>

```

Obviously, every method call gets translated into its own element (in our case, it's `<create_flower>`). All the method's parameters are turned into elements, too, and they become child elements of the method call element. They all have a `type=` attribute that specifies the data type of the parameter using XmlSchema.<sup>12</sup>

The return value of our method call looks like this:

```

<?xml version="1.0" encoding="utf-8"?>
<env:Envelope xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <env:Body>
    <n1:create_flowerResponse xmlns:n1="urn:Stock"
      env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <return xsi:type="xsd:int">1000</return>
    </n1:create_flowerResponse>
  </env:Body>
</env:Envelope>

```

Removing all the namespace clutter will help again:

```

<?xml version="1.0"?>
<Envelope>
  <Body>
    <create_flowerResponse>
      <return type="int">1000</return>
    </create_flowerResponse>
  </Body>
</Envelope>

```

To encode a method's return value, a new element is created and its name is build by appending "Response" to the name of the method that was called. The return value itself is encoded as a child called `<return>` of the newly created element. Its data type is specified the same way as the data type of method parameters.

SOAP is much more than this (see *Programming Web Services with SOAP* [?], for example), but for our purposes it's sufficient to know how

---

<sup>12</sup><http://www.w3.org/XML/Schema>

method calls and responses are encoded. To call the `create_flower()` method we could create an XML document such as the one above and send it using Ruby's HTTP library. Then we could parse the response using REXML. Actually, there are lots of clients out there that work exactly this way. We will follow a more flexible approach and use Ruby's standard SOAP library, called `soap4r`.

## SOAP the Hard Way

To turn the `Stock` class into a SOAP service we use Hiroshi Nakamura's `soap4r` library. It is part of the Ruby standard library, implements SOAP version 1.1<sup>13</sup>, and makes the creation of web services really easy:

```
File
Line 1  require 'soap/rpc/standaloneServer'
-       require 'stock'
-
-       class StockServer < SOAP::RPC::StandaloneServer
5         def on_init
-           @stock = Stock.new
-           @log.level = Logger::Severity::DEBUG
-           add_method(self, 'create_flower', 'name', 'price', 'quantity')
-           add_method(self, 'add_flowers', 'name', 'quantity')
10          add_method(self, 'remove_flowers', 'name', 'quantity')
-           add_method(self, 'set_price', 'name', 'price')
-           add_method(self, 'get_report')
-         end
-
15        def create_flower(name, price, quantity)
-           @stock.create_flower(name, price, quantity)
-         end
-
-         def add_flowers(name, quantity)
20          @stock.add_flowers(name, quantity)
-         end
-
-         def remove_flowers(name, quantity)
-           @stock.remove_flowers(name, quantity)
25        end
-
-         def set_price(name, price)
-           @stock.set_price(name, price)
-         end
30
-         def get_report
-           @stock.get_report
-         end
```

---

<sup>13</sup>There is a W3C recommendation for version 1.2, but it's currently not in widespread use.

```

-   end
35
-   server = StockServer.new('stock', 'urn:Stock', '0.0.0.0', 2000)
-   trap(:INT) { server.shutdown }
-   server.start

```

We actually didn't have to touch the `Stock` class to make its functionality available in our first SOAP server. In fact, we didn't have to do much at all.

First, we derived our `StockServer` class from a standard `soap4r` class called `SOAP::RPC::StandaloneServer`. This is one of several ways to create SOAP services with `soap4r`.

The `on_init()` method gets called when the server is initialized, so we use it to create our `Stock` instance, configure the logging system (you'll need a lot of debug output, especially in the beginning), and declare all the remote methods we're going to support. All the remaining methods simply delegate their work to their counterparts in the `Stock` class.

In line 36 we create a server instance. The constructor gets four parameters:

- The application name.
- URN of the application.
- The name or IP address of the host the service is running on. `0.0.0.0` means that the service is listening on all interfaces.
- The port the service is running on.

In line 37 we make sure that the server gets terminated when it receives a `SIGINT` signal. Finally, on the last line, we start the server.

At this point you could ask a member of Jeff's team if they'd try to use and test the new service, but you're afraid that it if doesn't run properly and could ruin all your newly gained reputation. Better write your own test client:

```

File
Line 1  require 'soap/rpc/driver'
-
-   stock = SOAP::RPC::Driver.new('http://localhost:2000', 'urn:Stock')
-   [
5       %w(create_flower name price quantity),
-       %w(add_flowers name quantity),
-       %w(remove_flowers name quantity),
-       %w(set_price name price),
-       %w(get_report)
10  ].each do |signature|

```



```

-     stock.add_method(*signature)
-   end
-
-   begin
15     stock.create_flower('rose', 1.99, 1000)
-     stock.create_flower('orchid', 3.14, 200)
-     puts "Created 'rose' and 'orchid'."
-     p stock.get_report
-
20     stock.remove_flowers('rose', 5)
-     puts 'Removed 5 roses.'
-     p stock.get_report
-
-     stock.add_flowers('orchid', 100)
25     puts 'Added 100 orchids.'
-     p stock.get_report
-
-     stock.set_price('orchid', 3.01)
-     puts 'Changed orchid price to $3.01.'
30     p stock.get_report
-   rescue Exception => ex
-     puts ex
-   end

```

This produces the following output (after we have deleted all records from the flowers table):

```

Created 'rose' and 'orchid'.
{"orchid"=>[200, 3.14], "rose"=>[1000, 1.99]}
Removed 5 roses.
{"orchid"=>[200, 3.14], "rose"=>[995, 1.99]}
Added 100 orchids.
{"orchid"=>[300, 3.14], "rose"=>[995, 1.99]}
Changed orchid price to $3.01.
{"orchid"=>[300, 3.01], "rose"=>[995, 1.99]}

```

That wasn't too difficult, either. We obtained a proxy for the stock service by creating a `SOAP::RPC::Driver` instance, passing it the address and the URN of the service we want to access. In lines 4 to 12 we tell the proxy which methods we'd like to use. From then on we could treat the proxy as if it were an instance of the `Stock` class.

So far, so good, but our current implementation of the server does not meet our high coding standards. Why do we have to repeat the `Stock` class' API in the SOAP server? Ruby is supposed to be a dynamic language. Isn't it possible to create the server automatically from the `Stock` class? Fortunately, it is:

```

Line 1  class StockServer < SOAP::RPC::StandaloneServer
-       def initialize(*args)

```

File

```

-      super
-      @log.level = Logger::Severity::DEBUG
5      add_servant(Stock.new)
-    end
-  end

```

Sweet, isn't it? Instead of adding every single remote method with `add_method()`, you can make all the methods of a particular class available by passing an instance of the class to `add_servant()`.

## Web Services Description Language

Did you notice that we are still violating the DRY principle? In the client we call `add_method()` for every method of the `Stock` class to define the interface we want to use. Wouldn't it be nice if we could generate the boring and tedious interface code from a more abstract definition?

In CORBA environments you can use the Interface Definition Language (IDL) to describe interfaces and to generate stubs and skeletons for servers and clients. SOAP uses the *Web Services Description Language* (WSDL) for this purpose. It's an XML dialect made for describing the interface of remote services.

*Web Services Description  
Language*

With WSDL you can describe the service itself, its interface, and its bindings. It's not required, but it's good practice, to separate the service description from the interface description. Our service description looks as follows:

File

```

Line 1  <?xml version="1.0"?>
-      <definitions name="StockServiceImplementationDescription"
-          targetNamespace="http://www.pragbouquet.com/wsdl/StockService.wsdl"
-          xmlns="http://schemas.xmlsoap.org/wsdl/"
5          xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
-          xmlns:tns="http://www.pragbouquet.com/wsdl/StockService.wsdl"
-          xmlns:xsd="http://www.w3.org/2001/XMLSchema">
-
-      <import namespace="urn:Stock" location="stock.wsdl"/>
10
-      <service name="StockService">
-          <documentation>
-              The stock service allows you to manage PragBouquet's flower
-              stock.
15          </documentation>
-          <port binding="tns:StockBinding" name="StockPort">
-              <soap:address location="http://localhost:2000"/>
-          </port>
-      </service>
20  </definitions>

```

The biggest problem with WSDL files is that they are often cluttered with XML namespaces. If you ignore these, the rest is easy to read. In line 9 we import the interface definition of the stock service that we are going to describe next. We then use the `<service>` element to briefly describe it. With the `<documentation>` element we explain a bit about the purpose of our service and the `<port>` element tells us where it can be found.

Describing the service's interface is a bit more complex, but still not difficult:

File

```

Line 1  <?xml version="1.0"?>
-      <definitions name="StockServiceInterfaceDescription"
-          targetNamespace="http://www.pragbouquet.com/wsd1/StockService.wsd1"
-          xmlns="http://schemas.xmlsoap.org/wsd1/"
5          xmlns:soap="http://schemas.xmlsoap.org/wsd1/soap/"
-          xmlns:tns="http://www.pragbouquet.com/wsd1/StockService.wsd1"
-          xmlns:xsd="http://www.w3.org/2001/XMLSchema">
-
-          <message name="set_price_in">
10              <part name="name" type="xsd:string"/>
-              <part name="price" type="xsd:double"/>
-          </message>
-
-          <message name="set_price_out">
15              <part name="price" type="xsd:double"/>
-          </message>
-
-          <portType name="StockInterface">
-              <operation name="set_price">
20                  <input message="tns:set_price_in"/>
-                  <output message="tns:set_price_out"/>
-              </operation>
-          </portType>
-
25          <binding name="StockBinding" type="tns:StockInterface">
-              <soap:binding style="rpc"
-                  transport="http://schemas.xmlsoap.org/soap/http"/>
-              <operation name="set_price">
-                  <soap:operation soapAction="set_price"/>
30                  <input>
-                      <soap:body
-                          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
-                          namespace="urn:Stock"
-                          use="encoded"/>
35                  </input>
-                  <output>
-                      <soap:body
-                          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
-                          namespace="urn:Stock"

```

```

40         use="encoded"/>
-       </output>
-       </operation>
-       </binding>
-       </definitions>

```

WSDL does not force us to declare the whole interface at once, so we only declared the `set_price(name,price)` method. Let's dissect it element by element.

We use the `<message>` elements to declare the signatures and return values of all methods we are going to describe. In SOAP everything is a message—a method gets a message containing its parameters and it sends back its return value as a message. The `set_price(name,price)` method receives a message called `set_price_in` that consists of two parts: a String called `name` and a double called `price`. A `set_price_out` message containing a double object named `price` is sent back.

It may look a bit awkward to declare our Ruby methods this way, but SOAP was meant to be interoperable and so the standards committee had to agree upon a list of data types that are available in most (static) programming languages. Hence, data types in SOAP can be everything you are used to in object-oriented languages such as Java and C++: atomic types such as `int` and `double`, arrays, structs, and even full-blown objects. They are described using `XmlSchema`<sup>14</sup> and reside in the `xsd` namespace.

We describe the methods (or operations as SOAP calls them) offered by the service with the `<portType>` element beginning on line 18. In our case this is simple: we have an operation called `set_price()` that expects a message of type `set_price_in` and returns a `set_price_out` message.

The rest of the WSDL file deals with the so-called service binding. It's possible to declare several implementations of the same interface in a single WSDL file. For example, there could be different SOAP implementations of our Stock service. One that uses HTTP as its transport layer and another one that uses HTTPS. Both services would have the same interface, but they would have different bindings. (In fact, they don't even have to be SOAP services at all.) But our Stock service is a plain old SOAP service running on top of HTTP.

Now that we have this abstract, omnipotent XML description of our stock, only a single question remains: what the heck can we do with it?

---

<sup>14</sup><http://www.w3.org/XML/Schema>

Simple answer: we can use it to generate all the boring code we had to write manually before:

```
mschmidt:/tmp> ls
sd_stock.wsdl    stock.wsdl
mschmidt:/tmp> wsdl2ruby.rb --type client --wsdl sd_stock.wsdl
I, [2005-12-31T12:29:39.745796 #567] INFO -- app: Creating class \
definition.
I, [2005-12-31T12:29:39.786257 #567] INFO -- app: Creates file \
'StockServiceImplementationDescription.rb'.
I, [2005-12-31T12:29:39.789094 #567] INFO -- app: Creating driver.
I, [2005-12-31T12:29:39.790492 #567] INFO -- app: Creates file \
'StockServiceImplementationDescriptionDriver.rb'.
I, [2005-12-31T12:29:39.796227 #567] INFO -- app: Creating client \
skelton.
I, [2005-12-31T12:29:39.797680 #567] INFO -- app: Creates file \
'StockServiceClient.rb'.
I, [2005-12-31T12:29:39.800845 #567] INFO -- app: End of app.\
(status: 0)
mschmidt:/tmp> ls
StockServiceClient.rb
StockServiceImplementationDescription.rb
StockServiceImplementationDescriptionDriver.rb
sd_stock.wsdl
stock.wsdl
mschmidt:/tmp>
```

wsdl2ruby.rb turns .wsdl files into Ruby code. Although it is part of soap4r, it is not part of the Ruby standard distribution and has to be installed separately.<sup>15</sup>

You can specify the .wsdl file to be compiled with the --wsdl option. Using the --type option you can choose if you want client or server code. We were interested in client code this time and wsdl2ruby.rb has generated three files for us. The most interesting one is StockServiceClient.rb:

File

```
Line 1  #!/usr/bin/env ruby
-      require 'StockServiceImplementationDescriptionDriver.rb'
-
-      endpoint_url = ARGV.shift
5      obj = StockInterface.new(endpoint_url)
-
-      # run ruby with -d to see SOAP wire dumps.
-      obj.wiredump_dev = STDERR if $DEBUG
-
10     # SYNOPSIS
-     #   set_price(name, price)
-     #
```

---

<sup>15</sup>soap4r's home page is <http://dev.ctor.org/soap4r>

```

- # ARGS
- #   name          String - {http://www.w3.org/2001/XMLSchema}string
15 #   price         Double - {http://www.w3.org/2001/XMLSchema}double
- #
- # RETURNS
- #   price         Double - {http://www.w3.org/2001/XMLSchema}double
- #
20 name = price = nil
- puts obj.set_price(name, price)

```

How convenient: replace name and price in line 20 with actual values and you are done.

All the other files that have been created are not meant to be touched. The file ending with “Driver” contains the proxy (or driver as soap4r calls it) for the SOAP service. The remaining file contains classes needed by the driver. Whenever the .wsdl file changes, they have to be generated anew.

For the sake of completeness let’s generate the server side, too:

```

mschmidt:/tmp> ls
sd_stock.wsdl  stock.wsdl
mschmidt:/tmp> wsdl2ruby.rb --type server --wsdl sd_stock.wsdl
I, [2005-12-31T14:33:09.426067 #701] INFO -- app: Creating class \
definition.
I, [2005-12-31T14:33:09.428748 #701] INFO -- app: Creates file \
'StockServiceImplementationDescription.rb'.
I, [2005-12-31T14:33:09.431321 #701] INFO -- app: Creating servant \
skelton.
I, [2005-12-31T14:33:09.432716 #701] INFO -- app: Creates file \
'StockServiceImplementationDescriptionServant.rb'.
I, [2005-12-31T14:33:09.436486 #701] INFO -- app: Creating standalone \
stub.
I, [2005-12-31T14:33:09.438000 #701] INFO -- app: Creates file \
'StockService.rb'.
- Standalone stub can have only 1 port for now. So creating stub for \
the first port and rests are ignored.
- Standalone server stub ignores port location defined in WSDL. \
Location is http://localhost:10080/ by default. Generated client \
from WSDL must be configured to point this endpoint manually.
I, [2005-12-31T14:33:09.474488 #701] INFO -- app: End of app. \
(status: 0)
mschmidt:/tmp> ls
StockService.rb
StockServiceImplementationDescription.rb
StockServiceImplementationDescriptionServant.rb
sd_stock.wsdl
stock.wsdl
mschmidt:/tmp>

```

File

```

Line 1  require 'StockServiceImplementationDescription.rb'
-
-  class StockInterface
-    # SYNOPSIS
5    #   set_price(name, price)
-    #
-    # ARGS
-    #   name          String - {http://www.w3.org/2001/XMLSchema}string
-    #   price         Double - {http://www.w3.org/2001/XMLSchema}double
10   #
-    # RETURNS
-    #   price         Double - {http://www.w3.org/2001/XMLSchema}double
-    #
-    def set_price(name, price)
15      p [name, price]
-      raise NotImplementedError.new
-    end
-  end
end

```

Generating the server skeleton is normally only interesting if you're developing a SOAP service from scratch. If you already have an implementation (like our Stock class), it's easier to "soapify" it with the method `add_servant()`. Despite all this, it's always a good idea to design your software without having a certain technology for distributed applications in mind. It may change sooner than you think...

There's an even easier way to automatically create client code from a .wsdl file:

File

```

Line 1  require 'soap/wsdlDriver'
-
-  wsdl = 'sd_stock.wsdl'
-  stock = SOAP::WSDLDriverFactory.new(wsdl).create_rpc_driver
-  stock.set_price('orchid', 2.42)

```

The constructor of the `SOAP::WSDLDriverFactory` accepts the file name of a WSDL file or an URL that points to one. Its `create_rpc_driver()` method returns a proxy for the service described in the WSDL file.

We cannot simplify the client anymore, so it's time to start the server and to send the .wsdl files to Jeff and his guys. They'll know what to do with 'em....

## Conclusion

After the inevitable hype dissipated, SOAP quickly became a standard technology in the industry. If you haven't used it already, the chances are good that you'll have to use or build a SOAP service some day. For most applications, SOAP is complete overkill. Sometimes, though, it

makes sense (and there will always be pointy-haired bosses that force you to use a certain technology even when it's absolutely inappropriate).

With Ruby it almost doesn't matter whether SOAP is appropriate or not, because `soap4r` completely hides all the nasty details, allowing you to fully concentrate on your application domain.

## 5.4 CORBA, RMI, and Friends

With the advent of the Internet, textual protocols like SMTP and HTTP became the norm and so it comes as no surprise that the most popular web services protocols like SOAP and XML-RPC are textual protocols, too.

Some years ago the situation looked a bit different and many organizations and companies tried to establish their own “standard” for building distributed architectures. Sun, e.g., has defined the *Remote Method Invocation* protocol (RMI) for Java and the *Object Management Group* (OMG)<sup>16</sup> has specified CORBA.

*Remote Method  
Invocation*

*Object Management  
Group*

All these approaches suffered from the same problems: they were way too complex and they all relied upon binary protocols. Only big companies like Sun, Borland, or IBM had the manpower to implement such specifications and even for them it was sometimes too difficult to do it right. Consequentially, the situations today is a mess: there are implementations for only a few programming languages, many systems do not interact as they should because of proprietary vendor extensions, and all in all the former “standards” have been superseded by their young and fresh fellows like XML-RPC, anyway.

Despite this, many companies have created CORBA components and RMI services during that short period of time and chances are good that you still have to use some of these relicts. Unfortunately, there is no CORBA or RMI implementation for Ruby (and probably there never will be one), but instead of implementing a CORBA or RMI protocol stack in Ruby, it's much more comfortable to reuse an existing implementation like the one for Java, e.g.

Hence, this section mainly deals with integrating Ruby and Java and we will show you how you can still use the gigabytes of .jar files that

---

<sup>16</sup>Read more about the OMG and its work at <http://www.omg.org>.



you have collected and created during the last years. Although our main example deals with CORBA, it explains all techniques necessary for accessing your good old RMI services, too.

## CORBA, Java, and Ruby

CORBA stands for *Common Object Request Broker Architecture* and is a language-neutral standard for object-oriented inter-process communication. Simply put, it allows you to instantiate remote objects and invoke methods on them transparently over a network. It has been popular in the industry at the end of the nineties, so chances are good that you can find some CORBA services in any company that has survived the dot-com bubble burst <sup>17</sup>.

Core of every CORBA system is the so called *Object Request Broker* (ORB). All service objects register themselves at the ORB and clients ask the ORB for particular services. To make network transparency possible all the marshaling and demarshaling code for all objects is generated from an abstract interface definition. Interfaces of CORBA objects are defined using the *Interface Definition Language* (IDL). Such interfaces are compiled using an IDL compiler that generates so called *stubs* for the clients and *skeletons* for the server objects. Stubs and skeletons act as proxies and convert method calls into network traffic and vice versa.

*Object Request Broker*

*Interface Definition Language*

*stubs*

*skeletons*

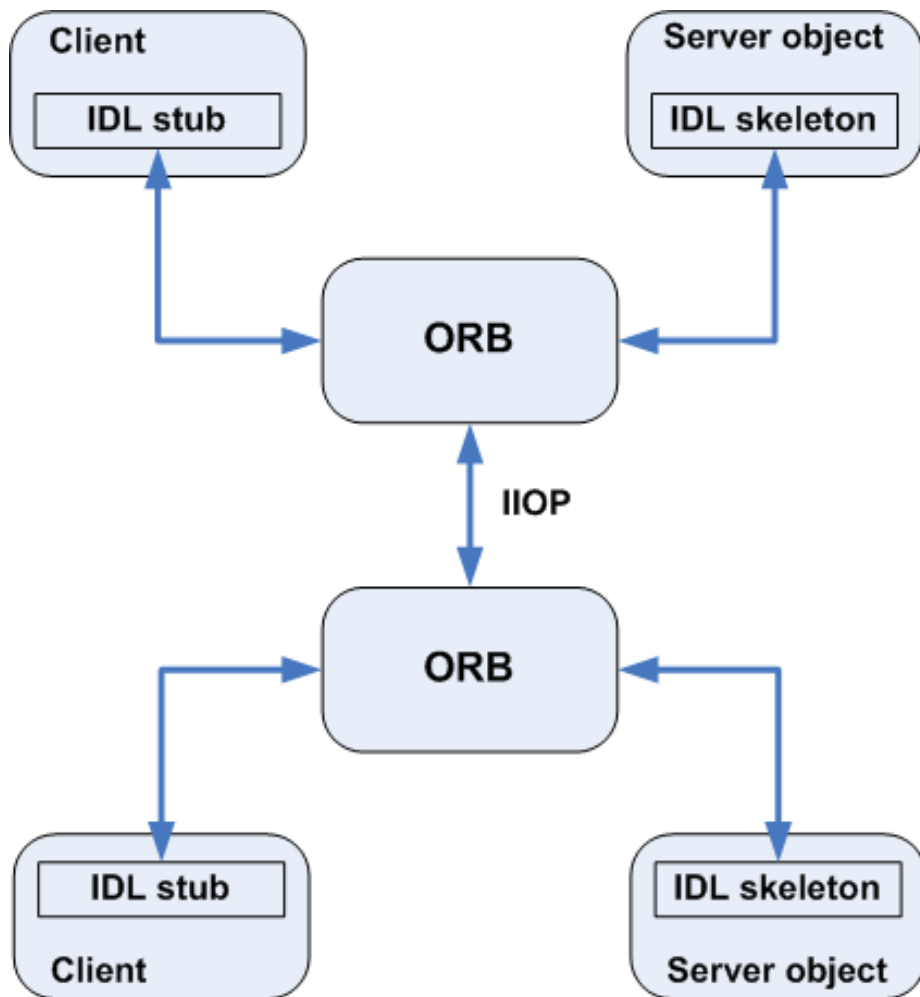
CORBA implementations usually come with a lot of standard services like, e.g., a naming service. Hence, clients do not have to know exact network addresses to obtain a reference to a particular service, they only have to know its name and have to ask the ORB to find the service (they have to know the address of the ORB, of course). It is even possible that different ORBs communicate to find a particular service. Therefore the *Internet Inter ORB Protocol* (IIOP) was invented. See Figure 5.5, on the following page for an overview.

*Internet Inter ORB Protocol*

## The SMS Server again

Before drowning in theory we will show you how to access CORBA services with Ruby. Do you remember the HTTP server that allowed us to send short messages in cellular networks (if not you should read now)? I have to admit that I've tricked you a bit: there is no such thing like a

<sup>17</sup><http://en.wikipedia.org/wiki/Dot-com>



---

Figure 5.5: CORBA Overview

---

HTTP interface for this server. It's a CORBA service and I've wrapped it with a thin HTTP layer.

The CORBA interface definition of the SMS service we have used looks like this:

```
File
Line 1  #ifndef SMS_IDL__
-       #define SMS_IDL__
-
-       module sms {
5
-       exception IOError {};
-
-       interface SmsService {
-           /* Sends a textual short message and returns the amount
10          * of characters sent.
-           */
-           short send_text(
-               in string recipient,
-               in string sender,
15          in string data) raises (IOError);
-
-           /* Sends a binary short message (the data has to be
-           * encoded as two-digit, case-insensitive hexadecimal
-           * values). Returns the amount of bytes sent.
20          */
-           short send_binary(
-               in string recipient,
-               in string sender,
-               in string data) raises (IOError);
25      };
-      };
-
-      #endif
```

Even if you've never seen an .idl file before you should have no problems understanding it. We define an interface called `SmsService` that declares an exception called `IOError` and exports two methods for sending short messages. All these elements belong to the `sms` module.

An IDL compiler turns such an interface definition into stub and skeleton code for a particular programming language and generates everything that is needed to transport data types, method calls, and exceptions in a transparent manner.

Unfortunately, there is no native CORBA binding for Ruby (to be concise: there is not even a standardized IDL mapping. Standardized mappings exist for C, C++, Java, COBOL, Smalltalk, Ada, Lisp, Python, and IDLscript) and consequentially there is no IDL compiler for Ruby. That's

not a big problem, because there are many for Java and it's easy for us to use one of them <sup>18</sup>.

Our approach is to build the client code with Java and embed it into Ruby. One of the nice things about CORBA is that it doesn't matter if the server has been implemented in C++ or Java. You can always write or use a Java client, which is usually easier than fiddling around with C++ compiler options and the like.

The `idlj` command shipping with the Java SDK turns our `.idl` file into a bunch of `.java` files (the `-fall` option generates both client and server code):

```
mschmidt:~/sms> idlj -fall sms.idl
mschmidt:~/sms> ls sms
IOError.java                SmsServiceHolder.java
IOErrorHandler.java         SmsServiceOperations.java
IOErrorHolder.java          SmsServicePOA.java
SmsService.java             _SmsServiceStub.java
SmsServiceHelper.java
```

First of all, we create a dummy implementation of the SMS service that makes testing easier and prevents us from sending short messages frequently. Our implementation has to be derived from class `SmsServicePOA` and looks as follows:

File

```
Line 1  import sms.*;
-
-      class SmsServiceImpl extends SmsServicePOA {
-          public short send_text(
5              final String recipient,
-              final String sender,
-              final String data) throws IOError
-          {
-              // Send SMS somehow...
10              final short text_length = (short)data.length();
-              final short bit_amount = (short)(text_length * 7);
-              short byte_amount = (short)(bit_amount / 8);
-              if (bit_amount % 8 != 0)
-                  byte_amount++;
15              return byte_amount;
-          }
-
-          public short send_binary(
-              final String recipient,
20              final String sender,
```

<sup>18</sup>At <http://java.sun.com/j2se/1.4.2/docs/guide/idl/GShome.html> you can find a nice Java/CORBA tutorial.

```

-         final String data) throws IOError
-     {
-         if (data.length() % 2 != 0)
-             throw new IOError("Odd length of binary data!");
25
-         // Send SMS somehow...
-         return (short)(data.length() / 2);
-     }
- }

```

This implementation of the `sendTextSms()` and `sendBinarySms()` does not do a lot, but for testing purposes it's sufficient. Both methods calculate the amount of bytes that would have been transmitted, if we actually had sent a short message. For binary messages it's easy, because we encode the binary data as a string of two-digit, case-insensitive hexadecimal values. Hence we have to divide the string length by 2 to get the byte amount. The calculation for text messages is a bit more complicated, because the character set encodings for short messages usually only use 7 bits per character.

File

```

Line 1  import org.omg.CosNaming.*;
-       import org.omg.CORBA.*;
-       import org.omg.PortableServer.*;
-       import sms.*;
5
-       public class SmsServer {
-           public static void main(final String args[]) {
-               try {
-                   final ORB orb = ORB.init(args, null);
10                  final POA rootpoa = POAHelper.narrow(
-                      orb.resolve_initial_references("RootPOA")
-                  );
-                  rootpoa.the_POAManager().activate();
-
15                  final org.omg.CORBA.Object ref =
-                      rootpoa.servant_to_reference(new SmsServiceImpl());
-                  final org.omg.CORBA.Object objRef =
-                      orb.resolve_initial_references("NameService");
-
20                  final NamingContextExt ncRef =
-                      NamingContextExtHelper.narrow(objRef);
-                  final NameComponent path[] =
-                      ncRef.to_name("SmsService");
-                  ncRef.rebind(path, SmsServiceHelper.narrow(ref));
25
-                  System.out.println("SMS server is starting ...");
-                  orb.run();
-              }
-              catch(Exception e) {
30                  System.err.println("An exception occurred: " + e);

```

```

-         e.printStackTrace(System.err);
-     }
- }
- }

```

If you're a CORBA pro, the lines above will be totally clear to you. All the others have to believe me: these lines register the services of the `SmsServer` class at the ORB under the name "SmsService". So, every client that wants to use our service has to ask the ORB for a reference to the service called "SmsService".

Before actually starting and registering the `SmsServer` object we have to start the ORB itself. For our purposes Sun's reference implementation called `orbd` is everything we need:

```
mschmidt:~/sms> orbd -ORBInitialPort 1050 -ORBInitialHost localhost &
```

The standard port for ORBs is 900, but we did not want to start the service as the root user and so we chose port 1050.

Let's start the server and register our service object at the ORB:

```

mschmidt:~/sms> javac SmsServer.java sms/*.java
mschmidt:~/sms> java SmsServer -ORBInitialHost localhost \
> -ORBInitialPort 1050 &
SMS server is starting ...

```

## The CORBA Client

Now that the whole infrastructure is up and running and we can implement our client.

File

```

Line 1  import sms.*;
-       import org.omg.CosNaming.*;
-       import org.omg.CORBA.*;
-
5       public class SmsServiceClient {
-           public SmsServiceClient() throws Exception {
-               this("localhost", 1050);
-           }
-
10      public SmsServiceClient(
-           final String host,
-           final int    port) throws Exception
-       {
-           _smsService = getSmsService(host, port);
15      }
-
-       public short sendTextSms(
-           final String recipient,

```

```

-         final String sender,
20         final String data) throws IOError
-     {
-         return _smsService.send_text(recipient, sender, data);
-     }
-
25     public short sendBinarySms(
-         final String recipient,
-         final String sender,
-         final String data) throws IOError
-     {
30         return _smsService.send_binary(recipient, sender, data);
-     }
-
-     private SmsService getSmsService(
-         final String host,
35         final int port) throws Exception
-     {
-         final String[] args = new String[] {
-             "-ORBInitialHost", host,
-             "-ORBInitialPort", "" + port
40        };
-         final ORB orb = ORB.init(args, null);
-
-         final org.omg.CORBA.Object objRef =
-             orb.resolve_initial_references("NameService");
45         final NamingContextExt ncRef =
-             NamingContextExtHelper.narrow(objRef);
-
-         final String name = "SmsService";
-         return SmsServiceHelper.narrow(ncRef.resolve_str(name));
50    }
-
-     private final SmsService _smsService;
- }

```

To demonstrate the usage of class `SmsServiceClient` in Java, we have written a small command line tool that allows us to send textual short messages from the shell:

File

```

Line 1  public class SendSms {
-         public static void main(final String args[]) {
-             try {
-                 final String recipient = args[0];
5                 final String sender = args[1];
-                 final String data = args[2];
-                 final SmsServiceClient smsServiceClient =
-                     new SmsServiceClient();
-                 short bytes = smsServiceClient.sendTextSms(
10                 recipient,
-                 sender,

```

```

-         data
-     );
-     System.out.println(bytes + " bytes have been sent.");
15 }
-     catch(Exception e) {
-         System.err.println("An exception occurred: " + e) ;
-         e.printStackTrace(System.err);
-     }
20 }
- }

```

This little program expects three command line arguments: the phone number of the recipient, the phone number of the sender, and the text to be sent (do not forget to put the text in quotes, if it contains blanks. Otherwise, the shell will interpret parts of your text as separate arguments). It is compiled and run as follows:

```

mschmidt:~/sms> javac SmsServiceClient.java SendSms.java sms/*.java
mschmidt:~/sms> java SendSms +0112345 +0198765 Hello
5 bytes have been sent.

```

## Bridging the Gap

We have everything available now: a running CORBA service and the according client code. The only thing left to do is integrating the client code with Ruby. Thanks to a project called “Ruby Java Bridge”<sup>19</sup> this is a piece of cake:

```

File
Line 1  require 'rjb'
-
-     sms_service_client_class = Rjb::import('SmsServiceClient')
-     sms_service = sms_service_client_class.new('localhost', 1050)
5  begin
-     puts sms_service.sendTextSms('+0112345', '+0198765', 'hello')
-     puts sms_service.sendBinarySms('+0112345', '+0198765', 'caffe')
-  rescue IOError => ex
-     puts "An exception occurred: #{ex}"
10 end

```

It produces:

```

mschmidt:~/sms> ruby sms_client.rb
5
An exception occurred: IDL:sms/IOError:1.0

```

The whole integration happens in two lines of code. `Rjb::import(classname)` returns a reference to the `SmsServiceClient` class and then we create an

<sup>19</sup><http://raa.ruby-lang.org/project/rjb>



## Ruby and Java

One of the things that made Java as popular as it is, is its virtual machine (JVM). Instead of generating code for a particular CPU, Java source code is compiled into byte code for a virtual processor. Hence, to run Java programs on a new hardware platform or a new operating system, you only have to write a new interpreter for the Java byte code.

There are many projects that have successfully built interpreters for languages targeting the Java Virtual Machine. For example, Jython\* is a Java implementation of the Python interpreter and Groovy† is a new dynamic language—similar to Ruby in many respects—that has been built for the JVM from the beginning.

Simply put, for integrating an arbitrary programming language with the JVM you have two options: you can choose the Jython or Groovy approach and translate the language directly into Java byte code or you can embed the JVM using the so called *Java Native Interface* (JNI)‡. Sun has created a Java Specification Request (JSR)§ that deals with the integration of the JVM and scripting languages.

It's only natural that Ruby developers also tried to reuse all the fine stuff that is available in the Java world and so there are many projects dealing with this topic:

- JRuby¶ is an implementation of a Ruby interpreter in Java. It's useful for embedding Ruby in Java and vice versa.
- rjb (Ruby Java Bridge) || uses JNI to access Java objects in Ruby programs. We will use it in this book, because it's relatively lightweight and is a perfect fit for integration purposes.
- yajb (Yet Another Java Bridge) \*\* uses a network approach to access Java objects in Ruby programs, i.e. it implements a Ruby client and a Java server that communicate using some kind of XML-RPC.
- rjni (Java binding for Ruby through JNI) †† uses JNI to access Java objects in Ruby programs. It's no longer maintained.

\*<http://www.jython.org>

†<http://groovy.codehaus.org>

‡<http://java.sun.com/docs/books/tutorial/native1.1>

§<http://www.jcp.org/en/jsr/detail?id=223>

¶<http://jruby.sourceforge.net>

||<http://raa.ruby-lang.org/project/rjb>

\*\*<http://raa.ruby-lang.org/project/yajb>

††<http://thekode.net/ruby/rjni>

instance as usual. Afterwards, we can forget that it actually is a Java class.

In the following lines we send a text message and a binary message. To provoke an exception, we send an invalid binary message and catch the exception in line 8.

It's nearly unbelievable, but that's all you have to do to use arbitrary Java classes in your Ruby programs and the API of the Rjb library is really lightweight:

- `load(classpath='.', jvmargs=())` explicitly loads the Java Virtual Machine. `classpath` contains a list of directories separated by the host's path separator. This list will be prepended to `ENV['CLASSPATH']`. `jvmargs` is an array of strings containing all arguments to be passed to the JVM. For example, the following statement

```
Rjb::load(
  'lib/servlet.jar',
  [
    '-Dhttp.proxyHost=example.com',
    '-Dhttp.proxyPort=8080'
  ]
)
```

loads the JVM, prepends the current directory and the file `lib/servlet.jar` to the class path and sets the system properties for a HTTP proxy.

If the JVM is not loaded explicitly by calling `load()`, it will be loaded automatically, before the first call to `import()`. Hence, you can write hybrid Ruby/Java one-liners:

```
ruby -rrjb -e "Rjb::import('java.lang.System').out.println('Strange, eh?')"
```

prints:

```
Strange, eh?
```

- `unload()` removes the Java Virtual Machine from memory.
- `import(classname)` turns the Java class called `classname` into a Ruby class. If we have the following Java class

File

```
package com.pragbouquet;

public class Flower {
    public Flower(final String name, final double price) {
        _name = name;
        _price = price;
    }
}
```

```

    public String getName() { return _name; }

    public double getPrice() { return _price; }
    public void setPrice(final double price) {
        _price = price;
    }

    private double _price;
    private String _name;
}

```

we can use it like this

File

```

flower = Rjb::import('com.pragbouquet.Flower')
f = flower.new('rose', 2.49)
puts "A #{f.getName()} costs ${f.getPrice()}."

```

and the code produces:

```
A rose costs $2.49.
```

- `bind()` allows you to associate a Ruby class with a Java interface. For example, the following program defines a `FileFilter` in Ruby that gets passed to the `listFiles(filter)` method of Java's `File` class.

File

```

Line 1  class FileFilter
-       def accept(file)
-           !(file.toString =~ /\.java$/).nil?
-       end
5       end

-
-       filter = FileFilter.new
-       filter = Rjb::bind(filter, 'java.io.FileFilter')
-       java_file = Rjb::import('java.io.File')
10      jf = java_file.new('.')
-       jf.listFiles(filter).each { |f| puts f.toString }

```

produces:

```

./BindDemo.java
./Flower.java

```

- `throw()` throws a Java exception. When you extend a Java class that has been converted into a Ruby class, it can be necessary to “simulate” a Java exception:

```

def foo(argument)
    if argument.nil?
        Rjb::throw('java.lang.NullPointerException', 'argument is null.')
    end
end

```

All classes returned by `Rjb::import(classname)` automatically have the following methods:

- `new_with_sig(signature,(arg)+)` allows you to call typed Java constructors from Ruby. Java is a statically typed language and therefore you sometimes have to explicitly say, which constructor you want to use. `Rjb` tries to automatically determine the signature you want according to the following rules:
  1. `Rjb` first of all checks if the number of arguments matches.
  2. All arguments that are instances of the same class match.
  3. Ruby `Fixnum` arguments match Java `byte`, `char`, `double`, `float`, `int`, `long`, and `short` parameters.
  4. Ruby `String` arguments match `java.lang.String` parameters.
  5. `true` and `false` match Java boolean parameters.
  6. Ruby arrays match Java arrays.
  7. Every Ruby object matches the Java `Object` class.

If there still is an ambiguity, you can call the constructor you want by passing the type information explicitly. Therefore you have to encode the signature using Java's type encoding<sup>20</sup> (see Figure 5.6, on the next page):

File

```
Line 1  flower = Rjb::import('com.pragbouquet.Flower')
-      f = flower.new_with_sig('Ljava.lang.String;D', 'rose', 2.49)
-      puts "A #{f.getName()} costs ${f.getPrice()}."
```

To encode the signature of an array you have to prepend a '[' character to the type encoding. The following `irb` session demonstrates how to call the Java's `String(byte[] bytes,String charsetName)` constructor:

```
irb(main):001:0> require 'rjb'
=> true
irb(main):002:0> Str = Rjb::import('java.lang.String')
=> #<Rjb::Java_lang_String:0x337b04>
irb(main):003:0> ruby = Str.new_with_sig(
irb(main):004:1*   '[Ljava.lang.String;',
irb(main):005:1*   [82, 117, 98, 121],
irb(main):006:1*   'iso-8859-1'
irb(main):007:1> )
=> #<#<Class:0x34739c>:0x328b90>
```

<sup>20</sup>[http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Class.html#getName\(\)](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Class.html#getName())

| Element Type | Encoding           | Element Type | Encoding      |
|--------------|--------------------|--------------|---------------|
| boolean      | Z                  | byte         | B             |
| char         | C                  | class        | L<classname>; |
| interface    | L<interface name>; | double       | D             |
| float        | F                  | int          | I             |
| long         | J                  | short        | S             |

Figure 5.6: Encoding of Java Types

```
irb(main):008:0> p ruby.toString
"Ruby"
=> nil
```

- `_invoke(method_name,signature,(arg)+)` invokes the method called `method_name` having the signature `signature` on the current object and passes it the according arguments. For details about the signature mechanism see the description of the `new_with_sig(signature,(arg)+)` method.
- `_classname()` returns the class name as string:

File

```
Line 1  str = Rjb::import('java.lang.String')
-      instance = str.new('Hello, world!')
-      puts instance._classname
```

produces:

```
java.lang.String
```

### CORBA's coming home...

Finally, we implement a HTTP server using WEBrick that forwards incoming requests to the CORBA service:

```
File
Line 1  require 'rjb'
-      require 'webrick'
-      include WEBrick
-
5      Rjb::load
-      sms_service_client_class = Rjb::import('SmsServiceClient')
-      sms_service = sms_service_client_class.new('localhost', 1050)
-
-      sms_server = HTTPServer.new(:Port => 4242)
10     sms_server.mount_proc('/send') do |req, res|
-         type = req.query['type'] || 'text'
-         recipient = req.query['recipient']
-         sender = req.query['sender'] || recipient
-         data = req.query['data']
```

```

15
-   res['Content-Type'] = 'text/plain'
-   res.status = 200
-   res.body = 'Message was sent successfully.'
-   method = type == 'binary' ? 'sendBinarySms' : 'sendTextSms'
20   begin
-       sms_service.send(method, recipient, sender, data)
-   rescue IOError => ex
-       res.status = 500
-       res.body = 'Message could not be sent.'
25   end
-   end
-
-   trap("INT") { sms_server.shutdown }
-   sms_server.start

```

If you have read carefully, the program above should be perfectly clear to you and with less than 30 lines of Ruby code we have turned our CORBA service into a HTTP service.

## C++ CORBA Services

For CORBA clients in theory it doesn't matter if the service they want to use has been written in C++, Java, or COBOL. If you absolutely want to (or have to) use a C++ client, you can use an approach similar to our Java example above: generate the C++ client stub from the .idl file using the `idl2cpp` command and compile the resulting code into a shared object. Then, integrate the shared object using the Ruby native interface, SWIG <sup>21</sup>, or Ruby/DL <sup>22</sup>.

We won't cover this topic here, because you can find tons of tutorials about integrating Ruby with C/C++ code on the internet and in *Programming Ruby* [?] there is a whole chapter about it, too.

## Distributed Ruby (dRuby)

As soon as a programming language offers dynamic features (such as reflection or code that's interpreted at runtime), it was pretty much guaranteed that someone would invent a system for distributing objects over a network. It's so easy: create methods that marshal and unmarshal objects, put the resulting byte streams into an envelope, and transfer them between processes using TCP (or any other protocol, for that matter).

---

<sup>21</sup><http://www.swig.org>

<sup>22</sup><http://raa.ruby-lang.org/project/ruby-dl>

The Ruby community fell to the temptation, and so Masatoshi Seki developed the Distributed Ruby (dRuby or DRb) module.<sup>23</sup> dRuby is similar to Java's RMI in many respects, including its use of a proprietary binary format to encode messages. It's simple to use, and usually does not require any modifications to the classes you want to distribute.

To see how it works we'll implement a distributed sequence object. Back in the sidebar, on page 29, we complained about the weak support for artificial primary keys in modern databases. Now it's time to roll up our sleeves and fix the problem once and for all—at least for our Ruby applications.

File

```

Line 1  require 'drb'
-
-  class Sequence
-    def initialize(start_value = 0, step = 1)
5      @value, @step = start_value - step, step
-    end
-
-    def next_value
-      @value += @step
10    end
-  end
-
-  DRb.start_service('druby://localhost:9000', Sequence.new)
-  DRb.thread.join

```

As promised, we did not have to use any special tricks to turn the Sequence class into a network service. To make an object available on a network with dRuby, you have to pass it to `DRb.start_service(url,object)` (see line 13). This method expects a dRuby URL and the object to be distributed.

It's easy to use our distributed sequence.

File

```

Line 1  require 'drb'
-
-  DRb.start_service
-  puts "Sequence #1:"
5  sequence = DRbObject.new(nil, 'druby://localhost:9000')
-  2.times { puts sequence.next_value }
-
-  puts "Sequence #2:"
-  sequence2 = DRbObject.new(nil, 'druby://localhost:9000')
10  2.times { puts sequence.next_value }

```

produces:

<sup>23</sup><http://www2a.biglobe.ne.jp/~protect~seki/ruby/druby.en.html>

```
Sequence #1:
0
1
Sequence #2:
2
3
```

That was too easy, wasn't it? Usually, creating network services is a difficult thing, so where's the catch in our example? Right: it isn't thread-safe. If two clients try to get the next sequence value simultaneously, it's possible that both of them get the same value. We'll fix this by adding some synchronization code:

File

```
Line 1  require 'thread'
-
-  class Sequence
-    def initialize(start_value = 0, step = 1)
5      @value, @step = start_value - step, step
-      @mutex = Mutex.new
-    end
-
-    def next_value
10     @mutex.synchronize do
-       @value += @step
-     end
-     @value
-   end
15   end
```

This version can be safely used in a multi-threaded environment, so are we done? Will a single sequence be sufficient for all our processes and database tables? Probably not. Let's implement a class that manages a set of named sequences:

File

```
Line 1  require 'drb'
-  require 'sequence'
-
-  class SequenceManager
5    def initialize
-      @sequences = {}
-      @mutex = Mutex.new
-    end
-
10   def create(name, start_value, step)
-     @mutex.synchronize do
-       if !@sequences.has_key?(name)
-         @sequences[name] = Sequence.new(start_value, step)
-       end
15   end
-     @sequences[name]
```



```

-     end
-
-     def get(name)
20       @sequences[name]
-     end
-   end
-
-   DRb.start_service('druby://localhost:9000', SequenceManager.new)
25   DRb.thread.join

```

Let's see if it works as expected:

File

```

Line 1  require 'drb'
-       require 'sequence'
-
-       DRb.start_service
5       factory = DRbObject.new(nil, 'druby://localhost:9000')
-
-       puts "Sequence #1:"
-       sequence = factory.create('order_table', 5, 2)
-       2.times { puts sequence.next_value }
10
-       puts "Sequence #2:"
-       sequence2 = factory.get('order_table')
-       2.times { puts sequence2.next_value }

```

produces:

```

Sequence #1:
5
7
Sequence #2:
5
7

```

Hm, we have a problem here. It's because dRuby normally passes objects by value. In our case the Sequence objects created by the SequenceManager are copied before being transmitted, so changes to the local instances aren't seen on the server. In line 8, we create a sequence called `order_table` and get two values from it in the following line. These changes are made only to the local object. When we get the `order_table` sequence again in line 12, it's another copy of the original on the server.

What we actually need is a remote reference to a Sequence object on the server. Fortunately, dRuby makes it possible: just mix `DRbUndumped` into classes you want to transfer by reference:

File

```

Line 1  class Sequence
-       include DRbUndumped
-     end

```

### Distributed Ruby and Security

dRuby passes (references to) real Ruby objects. In terms of security this is a big problem. For example, running the following code

File

```
Line 1  require 'drb'
-       DRb.start_service
-       object = DRbObject.new(nil, 'druby://localhost:9000')
-       puts object.instance_eval( "'date'" )
```

will print something like this:

```
Mon Dec 19 18:49:09 CET 2005
```

Would you be comfortable with code executing the shell command `rm -rf /*`? If not, you'd better set `$SAFE = 1` and undefine potentially dangerous methods such as `instance_eval()` and `instance_variable_set()`. In addition, you can install Access Control Lists (ACL) that prevent or allow access from certain hosts:

File

```
Line 1  require 'drb'
-       require 'drb/acl'
-
-       acl = ACL.new(%w(
5         deny all
-         allow localhost
-         allow 192.168.1.*
-       ))
-       DRb.install_acl(acl)
```

Class `DRbUndumped` creates a proxy that communicates with the real object. Instead of marshaling the whole object and passing it by value, only the proxy will be transferred. This explains why you have to execute `DRb.start_service` in your clients, too: every client has to be prepared to act as a server for incoming proxy calls.

After restarting the modified server our client works as expected:

```
Sequence #1:
```

```
5
7
```

```
Sequence #2:
```

```
9
11
```

For the final solution, you'd probably add a thin persistence layer that stores the current sequence values in a database or file system, but from a networking point of view we are done.

The implementation of dRuby is a perfect example of Ruby's strengths, demonstrating how easy it is to create a complete distributed object system with some nice bells and whistles in less than 3000 lines of code. If necessary, you can run it over a secure SSL connection and across firewalls.<sup>24</sup> In addition, it is surprisingly fast, as it uses Ruby's internal marshaling mechanism that's written in C.

dRuby finds uses all over the place. Folks often use it for prototyping new architectures. You'll also find it in Ruby on Rails,<sup>25</sup> where it's used to implement the remote breakpoint facility.

Despite all this, dRuby has the same disadvantages as all other language-specific systems. In particular, to access a dRuby service you have to use a Ruby client. Although we think that Ruby is currently the best programming language available, we're sure that it'll be improved or replaced by a better one someday. To prepare your distributed objects for the unpredictable future, you might want to consider using a standard technology such HTTP.

---

<sup>24</sup>See <http://www.rubygarden.org/ruby?DrbTutorial> for more details.

<sup>25</sup><http://api.rubyonrails.com/classes/Breakpoint.html>

# Tools and Techniques

---

A programming language alone will not make you more productive in an enterprise environment. Ruby will certainly increase your development speed, because it supports a lot of advanced concepts (pure object-oriented programming, iterators, and meta-programming, to name a few), but that's less than half the battle.

Especially, in enterprise environments, you face a lot of challenges that are not directly related to your programming language of choice, but that are typical for every piece of enterprise software. For example, enterprise software often comes in the shape of servers and daemon processes without any user interface. They have to be deployed somehow into the production system. To indicate that they are still alive and working they usually write into external log files. In addition, enterprise software often has to fulfill the needs of clients coming from different countries, speaking different languages, and paying in strange currencies.

Hence, in this chapter you will learn how to overcome internationalization problems, how to create sophisticated logging strategies, how to create your own daemon processes, and how to automatically deploy your software.

### 6.1 Internationalization and Localization

The world is getting smaller. For enterprise applications internationalization (i18n) and localization (l10n)<sup>1</sup> have become important topics.

---

<sup>1</sup>i18n is an abbreviation for internationalization. The word starts with an i, ends with an n, and in-between are 18 letters. Guess what l10n means....

You now have to be prepared to have people use your application in foreign countries—you'll have to deal with different character sets, date formats, address formats, number formats, etc.

There is no official definition of the terms internationalization and localization. We'll define them as follows:

- Your software is *internationalized* if it runs correctly in all locales it has to. If, for example, you have developed and tested a program in the United States and it runs correctly in your target markets Germany and Japan without modifying it, it is internationalized.
- Your software is *localized* if it is internationalized and reflects the peculiarities of the locale it is running in—it outputs messages in the correct language, prints numbers, dates, and currencies in the right format, and so on.

Perhaps you have worked with Java or C# before and now you are impatiently waiting for easy answers to all your i18n questions. You've skimmed the documentation of the Ruby standard library and did not instantly find the counterparts of `Locale`, `DateFormat`, `InputStream` and so on. In this chapter we will reveal an ugly truth: i18n is not supported very well in Ruby. Enterprise platforms such as J2EE and .NET come with a huge number of classes that deal with this topic, and even have standardized formats for resource bundles and so on. Ruby does not.

This is not as bad as it may seem in the beginning, but you will have to do a lot of things manually that you took for granted in other programming languages. But you can always find a solution.

## Character Sets and Encodings

Character sets and encodings are different things. The Unicode code character set, for example, contains nearly all known characters. It can be encoded in more than one way: UTF-8, UTF-16, and UTF-32.<sup>2</sup> Because of its Japanese roots, Ruby in principle supports different encodings such as EUC, SJIS, and UTF-8. Unfortunately, many of its classes do not.

---

<sup>2</sup>For more information about character sets and their encodings see:

- <http://www.w3.org/International/resource-index.html>
- <http://www.joelonsoftware.com/articles/Unicode.html>

### What's \$KCODE?

You can specify Ruby's internal character set by assigning 'EUC', 'SJIS', 'UTF8', or 'NONE' (which means use the default character set) to the global variable \$KCODE. Alternatively, pass 'e', 's', or 'u' to the -K option of the Ruby interpreter.

\$KCODE determines, which characters are allowed in identifiers (variable names, method names, etc.) and literals. To see what implications it has, run the following UTF-8 program without telling Ruby that it is encoded in UTF-8:

File

```
Line 1  class Circle < Struct.new(:x, :y, :r)
-       def area
-          $\pi$  = Math::PI
-          $\pi$  * r * r
5       end
-     end
-
-     c = Circle.new(0, 0, 1)
-     puts c.area
```

You will get the following result (in a UTF-8 terminal):

```
mschmidt:/tmp> ruby circle.rb
circle.rb:3: Invalid char '\317' in expression
circle.rb:3: Invalid char '\200' in expression
circle.rb:3: parse error, unexpected '='
   $\pi$  = Math::PI
      ^
circle.rb:4: Invalid char '\317' in expression
circle.rb:4: Invalid char '\200' in expression
circle.rb:4: parse error, unexpected '*', expecting '='
   $\pi$  * r * r
  ^
```

Obviously, ' $\pi$ ' has not been recognized as a valid word character. Tell Ruby that it is an UTF-8 program and everything is fine:

```
mschmidt:/tmp> ruby -Ku circle.rb
3.14159265358979
```

Note, though, that the \$KCODE does not affect the behavior of the String class, so the length() method, for example, still returns the number of bytes not characters:

```
mschmidt:/tmp> ruby -Ku -e 'puts "Motörhead".length'
10
```

One of the biggest problems with i18n is that developers do not know — and often do not care—what encoding their textual data has. If you get text from an external source such as a database, an LDAP repository, a socket, or a plain file, you always have to ask: “What encoding does this data use?”

In fact, the situation is even worse: you also have to ask yourself which encoding your text editor uses to store your source code files. The Ruby lexer treats everything between single or double quotes as a String—it copies all the *bytes* between quotes into a String object. Hence, it makes a big difference whether these bytes represent characters of a single-byte encoding such as ASCII or a multi-byte encoding such as UTF-8.

To see the difference, store the following statement as an UTF-8 text<sup>3</sup> and run it:

File `puts 'Über-Programmer'.length`

It will print “16”, although our string has only 15 characters. That’s because the German umlaut ‘Ü’ needs two bytes in UTF-8. Store the file using the ISO-8859-1 encoding and it will print “15”.

We have to realize that Ruby’s String class is nothing more than a byte array with a convenient API. It doesn’t know anything about the character concept. That becomes even more obvious when we try methods such as `upcase()`. When we store the following source code as an UTF-8 file and run it

File `puts 'Müller'.upcase`

it produces:

MÜLLER

When we store the same source code in ISO-8859-1 it prints:<sup>4</sup>

M?LLER

*Both* versions produce an incorrect result: the correct uppercase version of “Müller” is “MÜLLER”. Fortunately, the German alphabet only has four additional characters: ä, ö, ü, and ß. Their uppercase ver-

<sup>3</sup>If your text editor does not support UTF-8, go and get a new one. No excuses!

<sup>4</sup>I’m working with an UTF-8 terminal that prints ‘?’ for every character it doesn’t recognize. In this case ‘?’ stands for 0xfc which is a lowercase ‘ü’ in ISO-8859-1 and is invalid in UTF-8.

sions are Ä, Ö, Ü, and SS, so we can write a correct `upcase()` method ourselves:

File

```
Line 1  def german_upcase(text)
-       uc = text.upcase
-       uc.tr('äöü', 'ÄÖÜ')
-       uc.gsub('ß', 'SS')
5  end
-
-   str = 'Märchenstraße'
-   puts str
-   puts german_upcase(str)
```

The program above outputs the following:<sup>5</sup>

```
Märchenstraße
MÄRCHENSTRASSE
```

Alright, we have corrected one method, but what about `downcase()`, `index()`, `reverse()`, and so on.? All of them fail with multi-byte characters:

File

```
Line 1  str = 'Märchen'
-       puts 'Ouch!' if str.length != 7
-       puts 'Ouch again!' if str.reverse != 'nehcräM'
```

prints:

```
Ouch!
Ouch again!
```

Hmmm, things are getting complicated. Ruby supports different character sets, but you have to do a lot yourself. To determine the correct length of an UTF-8 string, for example, you could use the `unpack()` method with the format specifier `'U*'`, which stands for *unsigned integer values representing UTF-8 characters*:

```
puts 'Motörhead'.unpack('U*').length # -> 9
```

To manipulate each character (not byte!) of a UTF-8 string, you can use a regular expression that knows about UTF-8:

File

```
str = 'Motörhead'
p str.scan(/./)
p str.scan(/./u)
```

produces:

```
["M", "o", "t", "\303", "\266", "r", "h", "e", "a", "d"]
["M", "o", "t", "\303\266", "r", "h", "e", "a", "d"]
```

---

<sup>5</sup>'Märchen' is the German word for fairy-tale and 'Straße' is German for street.



**It really is that complicated...**

If you've never had to deal with characters outside the ASCII set then you might be surprised how complicated things get when you want to perform apparently simple operations such as converting a string to uppercase or sorting words.

Although the German alphabet has only four additional letters, they cause a lot of trouble and debate. We have three umlauts 'ä', 'ö', 'ü', and we have the letter 'ß' ("sharp s"). The uppercase equivalents of the umlauts are 'Ä', 'Ö', and 'Ü', but 'ß' becomes 'SS' when it's turned into uppercase, i.e. the size of an uppercase string can be bigger than the size of the original string!

A few years ago, the rules were even more complicated. When turning a string to uppercase, it was acceptable to convert ß to SZ, if SS had otherwise led to a misunderstanding (for example in the uppercase versions of Busse (plural form of the German word for bus) and Buße (German for expiation)).

Another big problem is sort order (collation). Usually, you ignore the umlaut—if you have two words that only differ in the dots (such as the German verbs "fallen" and "fällen"), the not-dotted version comes first. But there's also an industry standard (DIN 5007) that turns all German umlauts into their "non-dotted" equivalents (æ for ä, oe for ö, and ue for ü) and sorts them lexically. In this case "faellen" would come before "fallen".

When you add the 'u'-modifier to a regular expression, you get each UTF-8 character as a string (these strings can contain more than one byte!). Without the 'u'-modifier, you get every byte separately. This works for EUC (modifier: 'e') and SJIS (modifier: 's'), too.

Now let's turn our newly gained knowledge into a class that represents UTF-8 strings in Ruby:<sup>6</sup>

File

```
Line 1  class UString < String
-       def length; self.unpack('U*').length end
-
-       def reverse; self.scan(/./u).reverse.join end
5
```

<sup>6</sup>This example was originally published by *why the lucky stiff*:  
<http://redhanded.hobix.com/inspect/closingInOnUnicodeWithJcode.html>

```

-   def inspect; "u#{super}" end
-   end
-
-   module Kernel
10    def u(str)
-      UString.new str.gsub(/U\+([0-9a-fA-F]{4})/u) {
-        [ "$1".hex ].pack('U*')
-      }
-    end
15  end

```

Our UString class demonstrates some of Ruby's biggest strengths. First, we can override the methods of one of the most important standard classes (String) by subclassing it. Then, we override the `length()` and `reverse()` methods with our UTF-8 implementations. In addition, we changed the `inspect()` method so we can distinguish String and UString objects.

Finally, we added a new method called `u(str)` to Kernel. It allows us to create UTF-8 strings that can contain hexadecimal character literals beginning with 'U+'. Here's a small usage example:

File

```

Line 1  str = u'Märchen'
-      puts str.length == 7
-      puts str.reverse == 'nehcräM'
-
5      str = u'MU+00fcncchen'
-      puts str
-      puts str.length

```

It prints:

```

true
true
München
7

```

UString could be the basis for a full-blown UTF-8 string class, but you'd have to add a lot of code. Rather than writing your own implementation like this, you're better off using an existing one. We'll introduce some options in the following sections. None of them is perfect—they all have their strengths and weaknesses—so if you need stronger encoding support choose carefully.

## jcode

jcode is a standard library that modifies Ruby's String class. Depending on the value of the global variable `$KCODE` (see the sidebar, on page 232). jcode updates these methods: `chop!()`, `chop()`, `delete!()`,

### Don't Panic!

We have to admit it: Ruby is still in its infancy when it comes to internationalization, and it will take some time until the situation gets better.\*

But don't panic! Keep in mind that all this is only relevant if you have to manipulate, compare, or sort strings. If all you do is to read them from an external source and write them to another (such as displaying database data in a web browser) without touching a single byte, you are safe. And for special domains such as the Ruby on Rails framework, i18n solutions will probably be available soon.

Even if you absolutely have to manipulate Unicode strings, there is a good chance that you might find what you need in an Open Source library such as ICU4R. It might not be convenient, but at least it will work.

\*<http://redhanded.hobix.com/inspect/futurismUnicodeInRuby.html> shows what Matz wants to do.

`delete()`, `squeeze!()`, `squeeze()`, `succ!()`, `succ()`, `tr!()`, `tr()`, `tr_s!()`, and `tr_s()`. Additionally, it adds methods such as `jlength()`, `jcount()`, and `each_char()`, but it does not provide correct implementations for `upcase()`, `downcase()`, and so on.

File

```
Line 1  $KCODE = 'UTF8'
-       require 'jcode'
-       str = 'Köln' # Cologne, a big city in Germany.
-       puts "Ruby length: #{str.length}."
5       puts "jcode length: #{str.jlength}."
-       str.each_char { |c| print c.inspect, ' ' }
-       puts
```

prints:

```
Ruby length: 5.
jcode length: 4.
"K" "ö" "l" "n"
```

`jcode`'s biggest advantage is that it is part of every Ruby installation. In other respects it is severely limited and will only be sufficient for the simplest transformations.

## Unicode

For the sake of completeness we mention Yoshida Masato's unicode project.<sup>7</sup> It complements jcode somewhat, as it offers methods that allow you to compare UTF-8 strings and change them to upper or lower case:

File

```
Line 1  require 'rubygems'
-       require 'unicode'
-
-       puts Unicode::downcase('HOFBRÄUHAUS')
5       puts Unicode::upcase('straße')
```

prints:

```
hofbräuhaus
STRAßE
```

Unfortunately, even our trivial example does not work correctly, because the 'ß' letter has not been converted properly. In addition, the library seems not to be maintained any longer, so I recommend not using it.

## ICU4R

ICU4R<sup>8</sup> is (as I write this) a brand new project created by Nikolai Lugovoi that looks very promising. It is a C extension for IBM's "International Component for Unicode" (ICU)<sup>9</sup> It provides two classes: UString and URegexp. UString is a feature-rich implementation of strings encoded in UTF-16:

File

```
Line 1  require 'ustring'
-
-       str = u'Märchenstraße'
-       puts "Length is #{str.length} characters."
5       puts str.downcase
-       puts str.upcase
-       p str.to_s('ISO-8859-1').scan(/./)
```

produces:

```
Length is 13 characters.
märchenstraße
MÄRCHENSTRASSE
["M", "\344", "r", "c", "h", "e", "n", "s", "t", "r", "a", "\337", "e"]
```

<sup>7</sup><http://raa.ruby-lang.org/project/unicode/>

<sup>8</sup><http://rubyforge.org/projects/icu4r>

<sup>9</sup><http://ibm.com/software/globalization/icu>

In line 3 we create a new `UString` object with the `u(string)` method (you can turn a `String` object into a `UString` object by calling its `to_u()` or `u()` method.). We then print its length, change its case, and finally we turn back our `UString` object into a `String` by invoking `to_s(encoding)`, which accepts an encoding for the result now.

After you have created a `UString`, you can do a lot of useful things with it. For example, you can iterate over its characters and words:

File

```
Line 1  str = u'Märchenstraße'
        - str.each_char('de_DE') { |c| puts c.inspect_names }
        - "Hello, world!".u.each_word('en_US') { |w| puts "'#{w}'" }
```

Here's the output of the program above:

```
<U00004D>LATIN CAPITAL LETTER M
<U0000E4>LATIN SMALL LETTER A WITH DIAERESIS
<U000072>LATIN SMALL LETTER R
<U000063>LATIN SMALL LETTER C
<U000068>LATIN SMALL LETTER H
<U000065>LATIN SMALL LETTER E
<U00006E>LATIN SMALL LETTER N
<U000073>LATIN SMALL LETTER S
<U000074>LATIN SMALL LETTER T
<U000072>LATIN SMALL LETTER R
<U000061>LATIN SMALL LETTER A
<U0000DF>LATIN SMALL LETTER SHARP S
<U000065>LATIN SMALL LETTER E
```

```
'Hello'
','
' '
'world'
','
```

In the first line we iterate over each character using `each_char(locale)` and print its code and its official name. A lot of ICU methods accept a locale that consists of a two-character language code and a two-character country code separated by an underscore ('de\_DE', 'en\_GB', 'en\_US', etc.). If you set this locale to an empty string, it uses your machine's default locale. When iterating over characters the locale is important because of so-called combining marks that are sometimes used to store letters and their accents separately.<sup>10</sup>

The `each_word(locale)` method lets you iterate over the words in a string. Because the concept of a word greatly differs from language to lan-

<sup>10</sup>See <http://www.unicode.org/notes/tn2> for the gory details.

guage, you can pass a locale, too.

Another great feature of the `UString` class is its `fmt(locale,*args)` method, which is `sprintf()` on steroids:

```
File
Line 1  gb = u"In{0, date, MMMM} it will cost {1, number, currency}."
        puts gb.fmt('en_GB', Time.now, 12345.34)
        us = u"In{0, date, MMMM} it will cost {1, number, currency}."
        puts us.fmt('en_US', Time.now, 12345.34)
5      de = u"Im{0, date, MMMM} kostet es {1, number, currency}."
        puts de.fmt('de_DE', Time.now, 12345.34)
```

produces:

```
In January it will cost £12,345.34.
In January it will cost $12,345.34.
Im Januar kostet es 12.345,34 €.
```

`fmt()` handles everything nicely: month names are translated, decimal points are chosen correctly and thousands are separated by the right character. It picks the right currency symbol and it knows where to put it (did you know where Euro symbols go?). Awesome, isn't it? And it gets even better:

```
File
Line 1  s = u'{0, choice, 0#no chance/1#a chance/1<many chances}.'
        0.upto(2) { |i| puts "We have " + s.fmt('de_DE', i) }
```

produces:

```
We have no chance.
We have a chance.
We have many chances.
```

The 'choice' pattern consists of a number of range specifiers separated by '|' -characters. Every range specifier is compared to the actual argument and depending on its value the right one is chosen.<sup>11</sup>

Objects of class `URegex` handle Unicode regular expressions and can be created with the `ure(str)` method:

```
File
Line 1  puts 'Last character: ' + ure('(.)(.)(.)').match('süß'.u)[3]
        strange_csv = 'thisäisästrange'
        puts ure('ä').split(strange_csv.to_u, nil)
```

produces:

```
Last character: ß
this
```

---

<sup>11</sup>You should have a look at the ICU4R documentation. There are many more nice features.

```
is
strange
```

ICU4R's biggest disadvantage is that your strings need twice as much memory as before. In addition, a lot of important methods such as `tr()`, `chop()`, `delete()`, etc. are still missing.<sup>12</sup> Despite this, ICU4R is the most advanced Unicode library currently available for Ruby and might well become an important milestone for the internationalization of Ruby programs.

## International I/O

We know how strings in Ruby work in general. Now we'd like to see how the I/O classes support different character set encodings. For our first experiment, take your favorite text editor, set its encoding to UTF-8, and create a file called `utf8.txt` containing only the German surname 'Müller'. Six characters, no newline.

Then store the same file using the ISO-8859-1 character set or use the `iconv` utility to convert it:<sup>13</sup>

```
mschmidt:/tmp/data> iconv -f UTF-8 -t iso-8859-1 utf8.txt > \
> iso-8859-1.txt
mschmidt:/tmp/data> ls -l
total 16
-rw-r--r--  1 mschmidt  mschmidt  6 Jan  3 19:26 iso-8859-1.txt
-rw-r--r--  1 mschmidt  mschmidt  7 Jan  3 19:25 utf8.txt
```

Notice that the ISO-8859-1 file is shorter than its UTF-8 counterpart. A hex viewer such as `xxd` shows us why:

```
mschmidt:/tmp/data> xxd iso-8859-1.txt
00000000: 4dfc 6c6c 6572                               M.ller
mschmidt:/tmp/data> xxd utf8.txt
00000000: 4dc3 bc6c 6c65 72                             M..ller
mschmidt:/tmp/data>
```

In ISO-8859-1 the German umlaut 'ü' is encoded with a single byte (0xfc), but in UTF-8 it needs two (0xc3, 0xbc). All the other characters of the string 'Müller' are encoded the same way in ISO-8859-1 and UTF-8.

---

<sup>12</sup>It is a fairly young project, so that might already have changed by the time you're reading this.

<sup>13</sup>The ISO-8859-1 character set represents all western European languages.

Let's see what happens, when we feed our files to Ruby's standard IO classes:

File

```

Line 1  def examine_file(file_name)
-       content = IO.read(file_name)
-       puts "Its content is '#{content}'."
-       puts "Content length is #{content.length} bytes."
5       a = []; content.each_byte { |b| a << b }; p a
-       end
-
-       puts 'Here we have the ISO-8859-1 file...'
-       examine_file('data/iso-8859-1.txt')
10      puts
-       puts '... and here we have the UTF-8 file:'
-       examine_file('data/utf8.txt')

```

This produces:

```

Here we have the ISO-8859-1 file...
Its content is 'M?ller'.
Content length is 6 bytes.
[77, 252, 108, 108, 101, 114]

```

```

... and here we have the UTF-8 file:
Its content is 'Müller'.
Content length is 7 bytes.
[77, 195, 188, 108, 108, 101, 114]

```

That doesn't look very promising. Reading data in our standard character set works, but the IO class and the String class obviously don't know anything about character set encodings. All standard Ruby IO methods are strictly byte-oriented—you cannot tell them which character set encoding they should use.

This is different in Java, for example, where all strings are encoded with a Unicode encoding internally and all IO classes support different character set encodings. They distinguish between byte streams and character streams. To read an input stream that uses a particular encoding, you can use a `java.io.InputStreamReader` and initialize it with an `InputStream` and a character set encoding:

```
InputStreamReader in = new InputStreamReader(System.in, 'UTF-8');
```

The reader called in in the example above reads bytes from the console and interprets them as characters encoded in UTF-8.

If we need such a feature in Ruby, we have to do the conversion manually. Thanks to `Iconv` this is a piece of cake:

File

```

Line 1  require 'iconv'
-

```



```

-   class IO
-       def IO.i18n_read(name, from_cs = 'utf-8', to_cs = 'iso-8859-1')
5         converter = Iconv.new(to_cs, from_cs)
-         converter.iconv(read(name))
-       end
-     end
-
10  content = IO.i18n_read('data/utf8.txt')
-  puts "#{content} (#{content.length} bytes)"

```

In an UTF-8 terminal the program above prints:

```
M?ller (6 bytes)
```

We added a method called `i18n_read()` to the `IO` class that allows us to specify an input and an output character set. Now we can specify which character set encoding the creator of the file used and which encoding we'd like to work with.

`Iconv` is sufficient as long as you don't have to manipulate data. If you only want to display it on a web site, for example, set the character set encoding of the HTML pages and the HTTP header `content-type` appropriately, or convert the data using `Iconv`.

## Date and Time Formats

Back in the good old days of C programming, you had nothing but the `printf()` family of functions to format textual data. Sometimes you needed esoteric stuff such as `strftime()`, but even then you didn't want to know that there were different calendars such as the Gregorian and the Julian, and you didn't want to know that there were people who did not use 'am' and 'pm' in timestamps.

This kind of thinking no longer cuts it. The creators of platforms such as J2EE and .NET tried to solve typical i18n problems once and for all. Despite this, many programs that run on these platforms are often not capable of dealing with i18n issues. So what's the problem? Are the libraries buggy? Are the developers too stupid? They aren't, but often the APIs of the standard classes are much too complicated. Today you have beasts such as Java's `Calendar` and `DateFormat` classes that enable you to calculate your grandma's next 20 birthdays and print them in Klingon standard time, but seemingly simple tasks such as printing the current date need several method calls.

Perhaps surprisingly, in Ruby you have the best of breed classes when it comes to time and dates: the `Time`, `Date`, and `DateTime` classes provide

everything you're likely to need for date and time management. They even come with excellent documentation.<sup>14</sup> In addition, you can still hack away with `sprintf()`, `strftime()`, and friends.<sup>15</sup>

You won't run into serious problems as long as you output dates only in a numerical format such as 2006-01-09 or 30/09/1972. As soon as you want to output the name of a month or the name of a day, you have to manage the names for every language yourself if you use the standard classes. Ruby's `Date` and `Time` classes only provide the English names.

File

```
Line 1  require 'date'
-
-  birthday = Date.new(1972, 9, 30)
-  puts birthday.strftime('It was a %A in %B.')
```

prints:

`It was a Saturday in September.`

If you want to print the German translation, do something like the following:

File

```
Line 1  MONTHS = {
-      :de => %w(
-          placeholder Januar Februar März April Mai Juni
-          Juli August September Oktober November Dezember
5      )
-  }
-
-  DAYS = {
-      :de => %w(
10     Sonntag Montag Dienstag Mittwoch Donnerstag Freitag Samstag
-  )
-  }
-
-  month_name = MONTHS[:de][birthday.month]
15  day_name = DAYS[:de][birthday.wday]
-  puts "Es war ein %s im Monat %s." % [day_name, month_name]
```

Then you'll get these result:

`Es war ein Samstag im Monat September.`

That means you have to manage the translation of month and day names for every language you'd like to support. In the following section

<sup>14</sup>See <http://www.ruby-doc.org/stdlib/libdoc/date/rdoc> and <http://www.ruby-doc.org/stdlib/libdoc/time/rdoc>

<sup>15</sup>Isn't it interesting that Sun added `sprintf()`-like behavior recently to the Java platform? (<http://java.sun.com/j2se/1.5.0/docs/api/java/text/MessageFormat.html>)

you'll learn how to solve this kind of problem (alternatively, you can have a look at Section 6.1, *ICU4R*, on page 238).

## Managing Message Text

One of the most important things when localizing software is the management of the text of message text. Every program whose source code reaches a critical mass outputs hundreds or even thousands of different messages. No matter if the program has a text interface, a batch interface, or a GUI, it usually has to ask for input and print results or error messages.

Translating every single message into another language is already a lot of work, so it would be nice if you didn't also have to worry about the technical details of managing the different translations in your programs. Fortunately, the GNU foundation has created the `gettext` command.<sup>16</sup> Even better: Masao Mutoh has created a Ruby version called `Ruby-GetText`.<sup>17</sup>

In principle, using the `gettext` family of commands is simple. To localize all messages of your program, you have to perform the following steps:

1. Pass all hardcoded strings in your program to one of the `gettext` methods (`_(str)`, `gettext()`, etc.).
2. Run `rgettext` to create a `.pot` (*Portable Object Template*) file that contains a list of all messages to be translated. *Portable Object Template*
3. For every language you'd like to support create a copy of the `.pot` file and give it the extension `.po` (*Portable Object*). *Portable Object*
4. Edit the `.po` files and translate every message.
5. Turn all *Portable Objects* into *Machine Objects* with the `rmsgfmt` *Machine Objects* command. The machine objects have the extension `.mo`.

When you run the following program:

File

```
Line 1  require 'gettext'
-      include GetText
-
-      bindtextdomain('sample')
5      puts _('Our first example!')
-      str = gettext('Translate me!')
-      puts str
```

<sup>16</sup><http://www.gnu.org/software/gettext>

<sup>17</sup><http://gettext.rubyforge.org>

it prints:

```
Our first example!
Translate me!
```

We tell `rgettext` where to look for a particular translation of our message texts with the `bindtextdomain(domain,path=nil,locale=nil,charset=nil)` method. This points it to a file that can be found in the following subdirectory: `path/locale/LC_MESSAGES/domain.mo`. The parameters are:

`domain`

is a symbolic name for the translation package.

`path`

points to the directory that contains the `.mo` files. If it's `nil` and the environment variable `GETTEXT_PATH` is not set, `gettext` searches in `/usr/share/locale` and `/usr/local/share/locale`.

`locale`

specifies the locale (`'de_DE'`, `'en_US'`, and so on) to be used in the current file. If it is `nil`, the following environment variables will be checked consecutively: `LC_ALL`, `LC_TYPE`, `LC_MESSAGES`, `LANG`. If they are all empty, the system's default language will be used. You should not normally set this value explicitly, because doing so defeats the purpose of `gettext`.

`charset`

sets the output character set of the translated messages. If it is `nil` and the environment variable `OUTPUT_CHARSET` is not set, it defaults to the system's character set. You shouldn't set this variable explicitly.

Let's localize our little example step by step, creating a German translation. First, we extract all the messages with `rgettext`:

```
mschmidt:/tmp> rgettext sample.rb -o sample.pot
```

The newly created file `sample.pot` looks as follows:

File

```
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"POT-Creation-Date: 2006-01-12 19:28+0100\n"
"PO-Revision-Date: 2006-01-12 19:28+0100\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"
```

```
"Plural-Forms: nplurals=INTEGER; plural=EXPRESSION;\n"
```

```
#: sample.rb:5
msgid "Our first example!"
msgstr ""
```

```
#: sample.rb:6
msgid "Translate me!"
msgstr ""
```

It starts with a list of meta information, followed by all messages that have to be translated. Messages aren't identified by an artificial identifier but by the message text itself.

For every language we'd like to support, we have to create a portable object (.po) file:

```
mschmidt:/tmp> mkdir de
mschmidt:/tmp> cp sample.pot de/sample.po
```

Then we have to translate every message in the .po file. In our case the result looks like this:

```
msgid ""
msgstr ""
"Project-Id-Version: Sample 0.0.1\n"
"POT-Creation-Date: 2006-01-12 19:28+0100\n"
"PO-Revision-Date: 2006-01-12 19:28+0100\n"
"Last-Translator: Maik Schmidt\n"
"Language-Team: DE <de@li.org>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"
"Plural-Forms: nplurals=INTEGER; plural=EXPRESSION;\n"
```

```
#: sample.rb:5
msgid "Our first example!"
msgstr "Unser erstes Beispiel!"
```

```
#: sample.rb:6
msgid "Translate me!"
msgstr "Übersetze mich!"
```

Now we turn the translation into a machine object, creating a .mo file that contains a more compact representation. Usually, you will put them into a directory called locale that itself contains a subdirectory for every language supported (actually, we put it under a directory called LC\_MESSAGES, which is a convention for affirmative and negative system responses).

```
mschmidt:/tmp> mkdir -p locale/de/LC_MESSAGES
```

File

```
mschmidt:/tmp> msgfmt de/sample.po \
> -o ./locale/de/LC_MESSAGES/sample.mo
```

Finally, we test if everything works as expected:

```
mschmidt:/tmp> export GETTEXT_PATH=./locale
mschmidt:/tmp> LANG=de_DE ruby sample.rb
Unser erstes Beispiel!
Übersetze mich!
```

We set the `GETTEXT_PATH` environment variable to tell `gettext` where to search for `.mo` files (if your program does not work as expected, start it in debug mode with `ruby -d`). For the runtime of our sample script, we set the `LANG` variable to a German locale.

For the most simple cases this is all you need, but sometimes messages are a bit more dynamic (for example, when the value of a variable dictates whether you'd choose a plural or singular form of a message).

File

```
Line 1  require 'gettext'
-       include GetText
-       bindtextdomain("plural")
-       0.upto(3) do |i|
5         printf(
-           n_("%d file was removed.\n", "%d files were removed.\n", i),
-           i
-         )
-       end
```

prints:

```
0 files were removed.
1 file was removed.
2 files were removed.
3 files were removed.
```

You can specify different text for singular and plural forms of a message with the `n_(singular,plural,quantity)` method. You can define a “plural rule” in the portable object that determines—depending on the value of quantity—which message should be used.

In our current example `gettext` didn't just help us with translating a message text: it helped choose the right one. Although we did not specify which message text to choose the output is correct, because by default `gettext` chooses the first message text only if the parameter value is 1 (usually a good guess for a singular form).

A nice-looking German translation is a bit more difficult, because we need three different texts. Let's create the portable object first:

```
mschmidt:/tmp> rgettext plural.rb -o plural.pot
```

```
mschmidt:/tmp> cp plural.pot de/plural.po
```

After editing plural.po it looks like this:

```
msgid ""
msgstr ""
"Project-Id-Version: Sample 0.0.1\n"
"POT-Creation-Date: 2006-01-14 10:07+0100\n"
"PO-Revision-Date: 2006-01-14 10:07+0100\n"
"Last-Translator: Maik Schmidt\n"
"Language-Team: DE <de@li.org>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"
"Plural-Forms: nplurals=3; plural=(n < 2 ? n : 2)\n"

#: plural.rb:5
msgid "%d file was removed.\n"
msgid_plural "%d files were removed.\n"
msgstr[0] "Keine Dateien wurden gelöscht.\n"
msgstr[1] "Eine Datei wurde gelöscht.\n"
msgstr[2] "%d Dateien wurden gelöscht.\n"
```

For the first time we had to change one of the portable object headers: Plural-Forms. This header specifies how to determine a plural form. It consists of two parts: `nplurals` defines how many plural forms a message may have and `plural` is set to a piece of Ruby code that returns the index of the message text to be used for a particular parameter value. In our case we have different plural forms for 0, 1, and for values that are bigger than 1. Finally, we have to define the translated message texts in Array syntax.

It's time for a final test run:

```
mschmidt:/tmp> msgfmt de/plural.po \
> -o ./locale/de/LC_MESSAGES/plural.mo
mschmidt:/tmp> export GETTEXT_PATH=./locale
mschmidt:/tmp> LANG=de_DE ruby plural.rb
Keine Dateien wurden gelöscht.
Eine Datei wurde gelöscht.
2 Dateien wurden gelöscht.
3 Dateien wurden gelöscht.
```

Perfect! `gettext` is a mature and powerful tool.<sup>18</sup> Its Ruby implementation is comprehensive and often it will be everything you need to localize your software.

---

<sup>18</sup>Read its manual (<http://www.gnu.org/software/gettext/manual/gettext.html>), if you want to get the best out of it. Although it covers the C version, you'll learn a lot of useful tricks for the Ruby version, too.

## Conclusion

It will take some time until Ruby fully supports all the techniques that are necessary to create fully internationalized applications. Fortunately, it will probably take even longer until all software developers really understand what these techniques are ;-)

In the meantime you should keep in mind that many important tips and tricks related to internationalization and localization are completely independent of a particular programming language:

- Always test your applications with non-ASCII content from the beginning. The sooner you notice that your program fails miserably, when Özgür Müller wants to get some stuff delivered to Düsseldorf, the better are the chances that you can quickly fix it.
- Structure data—especially address data—as finely-grained as possible, because it's easier to create different output formats when you need them.
- Do not hardwire message texts, icons, and other resources. Use tools like `gettext` instead.
- Do not hardwire output formats for dates, numbers, currencies, etc.
- If you can specify an encoding (e.g. in HTTP headers or HTML pages), do it. At least it will make you think about it.
- Encapsulate string manipulations carefully, so you can replace them easily as soon as Ruby gets better i18n support.
- Delegate i18n issues to external sources, if they have better support. If, for example, your database provides a `to_upper(text)` function that correctly converts a string into uppercase, use it.

## 6.2 Logging

Typical enterprise production systems are often implemented as distributed architectures where lots of server processes that do not have any user interface communicate with each other. As a result, enterprise systems are full of log files. You probably already deal with some of these, like the ones created by the Apache web server or the Unix `syslog` daemon.

Log files are used for different purposes:



- As a poor man's debugger
- For creating statistics
- For troubleshooting
- For monitoring purposes

If you choose your logging strategy carefully, a whole tool suite can be build around the log files of your program. For the Apache web server, for example, there are programs that take the logs and create access statistics, click streams, and so on. Because it logs messages in a standard format, this whole process is decoupled from the web server.

In this section we'll introduce the two logging tools that are available for Ruby, `Logger` and `Log4r`.

## Logging with Logger

The standard distribution of Ruby provides logging support in form of the `Logger` class. It's pretty straightforward to use.

File

```
require 'logger'
logger = Logger.new(STDOUT)
logger.debug("Look, Ma: I've created my first logger...")
logger.info('... and started a program.')
logger.warn("It's getting boring.")
```

Our first example prints something like this to the console:

File

```
D, [2005-03-05T08:11:17.305000 #3012] DEBUG - : \
  Look, Ma: I've created my first logger...
I, [2005-03-05T08:11:17.305000 #3012] INFO - : \
  ... and started a program.
W, [2005-03-05T08:11:17.305000 #3012] WARN - : \
  It's getting boring.
```

As expected, we got three output lines. Big deal—we could have achieved the same results using simple `puts()` statements. But `Logger` has a lot more to offer than this.

## Log Levels

Every message that is logged with `Logger` has an associated log level:

```
DEBUG < INFO < WARN < ERROR < FATAL < UNKNOWN
```

The log levels are ordered by priority, so a warning message is more important than a debug message. For every log level there is a corresponding method that logs a message with a certain priority. This is especially useful for filtering and suppressing messages. For example,

debug messages are good for debugging purposes only and in a production system they may be annoying or could even slow down your process. Hence, it is possible to set a threshold for a Logger so only messages with a priority bigger or equal to the current threshold value get logged.

File

```
Line 1  require 'logger'
-   logger = Logger.new(STDOUT)
-   logger.level = Logger::WARN
-   logger.debug('You will not see me ...')
5   logger.info('... or me.')
-   logger.warn("I've warned you!")
-   logger.error('Ouch!')
```

In line 3 we have set the threshold to WARN, so only the last two messages get logged:

```
W, [2005-09-26T08:59:02.567292 #408]  WARN -- : I've warned you!
E, [2005-09-26T08:59:02.570480 #408]  ERROR -- : Ouch!
```

The set of available log levels is fixed, so if you need something other than the six levels provided by Logger, you'll have to use Log4r (see Section 6.2, *Logging with Log4r*, on page 254) instead.

## Logging To Files

Printing messages to the console doesn't cut it in a production environment. At the very least we'd like to store the output of our program in a file. Therefore Logger's new() method accepts a file name or an object derived from IO. This is how the file name variant works.

File

```
logger = Logger.new('logger_with_file.log')
logger.info('Logging to a file now.')
```

This example creates a file called logger\_with\_file.log in the current directory. Your program will log to it until your file system becomes full. To prevent such accidents (and angry calls from your system administrator), Logger supports so-called *rotating loggers* that automatically create generations of log files.

File

```
logger = Logger.new('rotating_logger.log', 2, 512)
10.times { logger.info('Wasting space and time...') }
```

This will create a rotating Logger, which renames the current log file whenever its size gets bigger than 512 bytes after logging a message.<sup>19</sup>

---

<sup>19</sup>We set the limit low here to show rotation in action. In production, you're likely to set the size limit to something in the megabyte range.

The file name will be generated by appending a number, so after running the program we have two files in the current directory: `rotating_logger.log` and `rotating_logger.log.0`.

In addition it's possible to create new generations of log files depending on periods of time. `Logger.new('sys.log', 'daily')` will create a `Logger` that changes the log file daily. Other options are `'weekly'` and `'monthly'`.

Logging to ordinary files is what you will need 99% of the time, but sometimes it's useful to log to an arbitrary IO object, such as a `StringIO` in the following example.

File

```
require 'stringio'
buffer = StringIO.new
io_logger = Logger.new(buffer)
io_logger.debug('Logging with IO.')
puts buffer.string
```

produces:

```
D, [2005-09-26T09:52:34.393910 #510] DEBUG -- : Logging with IO.
```

By writing your own class derived from `IO`, it's possible to create sophisticated loggers with `Logger`, but if you really need such a beast, you might be better off using `Log4r`.

## The Log Line Format

`Logger`'s biggest weakness is its lack of configuration options, especially regarding the format of the log lines. They always look like this:

```
log level, [timestamp #pid] log level -- progname: message
```

The elements are separated by different characters: The abbreviated log level is separated by a comma from the timestamp, timestamp and process id are put in square brackets, and so on. Wouldn't it be much easier to automatically process a log file if its format were a bit more consistent? Unfortunately, you cannot change the format easily.<sup>20</sup>

There is one piece of customization: the timestamp's format can be changed by setting the instance variable `datetime_format` to a format string supported by `strftime()`. For example, you could write

```
logger.datetime_format = '%Y-%m-%d %H:%M:%S'
```

<sup>20</sup>At least not in the version that is shipped with Ruby 1.8.2. The next generation (in 1.8.3) will provide such an option.

### How And What Should I Log?

Choosing a log file format does not seem like a big decision, but you should consider it carefully. The biggest challenge is to make it suitable for both human readers and automatic processes. If you have to spend minutes (or even hours) watching a log file using `less` or `tail` to find an obscure bug, you will be grateful if you don't have to read the same useless debug messages over and over again. You will be even more grateful if you can quickly come up with a little shell script that will do the job for you.

Make sure that you always write all vital information into the log file. Nothing is worse than having your application behave unexpectedly, but seeing no evidence of a problem in the log file. In an ideal world, the log file should contain everything you need to diagnose the problem. It is a good idea to log every input value your program receives, so you can at least create a unit test (and you have lots of them, don't you?) to reproduce the error on your development box.

Especially in enterprise environments, standardizing of log file layout and timestamp format pays off quickly.

## Logging with Log4r

For many purposes `Logger` is sufficient, but there are times when you need more sophisticated features. One of the biggest disadvantages of `Logger` is the lack of customizable log formats (in the current version) and the inflexibility of log destinations. In a multithreaded environment, for example, it is usually helpful to log the id of the current thread. In a distributed environment, logging to a local file may not meet your needs. With the basic `Logger` class, though, you don't have many options.

In the Java, C++, Perl, and Python worlds, the “log4\*” family (`log4j`, `log4cpp`, `Log4perl`, and `log4p`) sets the standard for logging. Thanks to Leon Torres we have a `Log4r`, too. It can be used like this.

File

```
Line 1  require 'rubygems'
-       require 'log4r'
-       include Log4r
-
5       joker = Logger.new('joker')
```

```

-   joker.outputters = Outputter.stdout
-   joker.info('I am back!')
-   joker.warn('This is my last warning, Batman!')

```

produces:

```

INFO joker: I am back!
WARN joker: This is my last warning, Batman!

```

For convenience we installed Log4r as a gem (read more about gems in Section 6.4, *RubyGems*, on page 288), so we have to require ‘rubygems’. To save some keystrokes we used include to bring the Log4r module into our program’s namespace.

We first create a Logger instance on line 5, giving it the name “joker”. By default, the Logger objects in Log4r do nothing with the messages you give them. To see the messages appear on the console, in a file, and so on, you have to explicitly assign one out more so-called *outputters* to the Logger instance. On line 6 we make sure that Joker’s messages can be seen on the console and in the following lines an information message and a warning are sent to our first Logger. Admittedly, the output is a bit spartan, but we will beautify it later.

Because super heroes are usually a bit short of time, they are only interested in the really important statements and serious threats coming from their archenemies. Log4r allows our heroes to increase the log level by setting `joker.level = WARN`, so Joker’s annoying info and debug messages will be suppressed and only the really scary stuff will appear on the ticker in the Batcave.

Our little program already demonstrates two of the most important classes in the Log4r class hierarchy, Logger and Outputter (The latter is equivalent to log4j’s Appender class.) Together with the Formatter class (called Layout in log4j) they form the basis for Log4r. We’ll dissect them in the following sections.

## Loggers

Logger classes are the interface to the whole logging system. You can create as many Logger objects as you like. Each must have a unique name, as Log4r organizes them in a hierarchy. Usually, you will create a separate Logger for every class with the same name as the class. This is only a convention: you can name your loggers any way you like.

Logger objects have methods for every log level; `debug()`, `info()`, `warn()`, `error()`, and `fatal()`. All take the message to be logged. Every Logger holds a threshold value. When one of these methods is called the logger checks whether the message's level is bigger or equal to the current threshold. If yes, the message is sent to *all* outputters belonging to the logger. Otherwise the message is ignored. Log4r's default levels are:

```
DEBUG < INFO < WARN < ERROR < FATAL
```

Log4r's Level class allows you to completely redefine the log levels. You can add new levels and change the order of existing levels. You could, for example, implement Sun's log level hierarchy for Java:<sup>21</sup>

File

```
Line 1  require 'rubygems'
-       require 'log4r'
-       require 'log4r/configurator'
-       include Log4r
5
-       Configurator.custom_levels(
-         :FINEST, :FINER, :FINE, :CONFIG,
-         :INFO, :WARNING, :SEVERE
-       )
10
-       logger = Logger.new('java-style')
-       logger.outputters = Outputter.stdout
-       logger.finest('Who needs this?')
-       logger.config('Or this?')
15      logger.info('yo')
```

This produces:

```
FINEST java-style: Who needs this?
CONFIG java-style: Or this?
```

Don't worry if you don't exactly know what a Configurator is—we'll explain it shortly. To define your own hierarchy of log levels you have to call `custom_levels(*levels)` and pass it a list of all the levels you want your Logger objects to have. This list has to be in ascending order of the log level priority. You can use strings or symbols for the level names. Level names have to start with an uppercase letter, but the names of their corresponding log methods will be completely lowercase. You should define your custom levels before you do anything else.

All Logger objects are children of the RootLogger singleton. This can be obtained by calling `Logger.root` or `Logger.global`. To navigate through the hierarchy of your Logger objects you can use path-like structures.

<sup>21</sup><http://java.sun.com/j2se/1.5.0/docs/api/java/util/logging/Level.html>

File

```

Line 1  Logger.new('abe')
-       Logger.new('abe::homer')
-       Logger.new('abe::homer::bart')
-       Logger['abe'].outputters = Outputter.stdout
5
-       Logger['abe'].debug('Once upon a time...')
-       Logger['abe::homer'].debug("D'oh!")
-       Logger['abe::homer::bart'].debug('Eat my shorts!')

```

produces:

```

DEBUG abe: Once upon a time...
DEBUG homer: D'oh!
DEBUG bart: Eat my shorts!

```

Here we created a hierarchy of three different Loggers called 'abe', 'homer', and 'bart'. By default the logger path delimiter is the '::' sequence (it can be changed by setting `LoggerPathDelimiter`). In our case we made 'homer' a child of 'abe', and 'bart' a child of 'homer'. Although we have only added an `Outputter` to 'abe', the debug messages of 'homer' and 'bart' are logged, too. This happens because all children inherit their parent's outputters by default (you can change this behavior by setting the additive attribute of the parent to false).

Child Loggers also inherit the log level threshold of their parents. This means that it's possible to change this value for a whole sub-tree of the Logger hierarchy:

File

```

Line 1  Logger['abe::homer'].level = INFO
-       Logger['abe'].debug('*snore*')
-       Logger['abe::homer'].debug("Marge!")
-       Logger['abe::homer::bart'].debug('Ay, caramba!')

```

which produces:

```

DEBUG abe: *snore*

```

We have set the log level threshold to `INFO` for all Loggers that are descendants of 'homer' and therefore only the debug messages of the 'abe' Logger are still visible.

## Formatters

Until now our log messages have been pretty boring, because only the log level, the logger name, and the log message itself have been printed. So, where's all the fancy stuff that we're been promising? Where are the timestamps, thread ids, and so on? Don't panic! `Log4r` has it all. Let's start with a simple example:

File

```

Line 1  fancy = Logger.new('fancy')
-       p = PatternFormatter.new(:pattern => "[%5l] %d: %m")
-       fancy.add(STDOUTOutputter.new('stdout', :formatter => p))
-       fancy.info(%w(cat mouse dog))
5       fancy.error('I am so nicely formatted!')

```

produces:

```

[ INFO] 2005-09-25 16:33:41: catmousedog
[ERROR] 2005-09-25 16:33:41: I am so nicely formatted!

```

Did you notice that we have passed an array to the `info()` call? `Log4r` handles such things nicely, because it calls `to_s()` on every object that is passed to a log method.

All the formatting stuff `Log4r` has to offer is performed by the `Format` and `PatternFormat` classes. Usually, a pattern format is all you need, because it works the way `printf()` fans are used to. On line 2, we create a new `PatternFormat` object that prints the log level (in square brackets and with a fixed width of 5 characters), the current timestamp, and the actual log message. Format strings may contain arbitrary characters and special sequences prefixed by a percent sign. All attributes available are listed in Figure 6.1, on the next page.

Because it's one of the most important pieces of information in log files, `Log4r` handles the formatting of timestamps separately. If your web shop is new and you do not expect too many requests, the following date format may be sufficient:

File

```

Line 1  p = PatternFormatter.new(
-       :pattern => "[%5l] %d: %m",
-       :date_pattern => '%Y'
-       )
5       fancy.add(STDOUTOutputter.new('stdout', :formatter => p))
-       fancy.info('Oh, our second customer.')

```

produces:

```

[ INFO] 2005: Oh, our second customer.

```

Or maybe you have so many requests that even microseconds matter? No problem, just pass the `usec()` method of the `Time` class as `date_method` to the `PatternFormat` constructor (`date_method` expects one of class `Time`'s methods):

File

```

Line 1  p = PatternFormatter.new(
-       :pattern => "[%5l] %d: %m",
-       :date_method => :usec
-       )
5       fancy.add(STDOUTOutputter.new('stdout', :formatter => p))

```



| Format Specifier | Meaning                                                                                                                                                                                   |
|------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| c                | A Logger's name.                                                                                                                                                                          |
| C                | A Logger's full name up to the RootLogger.                                                                                                                                                |
| d                | Current timestamp (in iso-8601 format by default). Otherwise the format returned by <code>date_pattern()</code> or by <code>date_method()</code> will be used.                            |
| t                | Name of the file (and line number) that's the origin of the log message. Has the same format as <code>Kernel.caller[0]</code> .                                                           |
| m                | Whatever is returned when <code>to_s()</code> is called on the log message.                                                                                                               |
| M                | Whatever is returned when <code>format_object()</code> in class <code>BasicFormatter</code> is called on the log message. For example, exceptions will be nicely formatted automatically. |
| l                | The log level.                                                                                                                                                                            |
| %                | The percent sign itself.                                                                                                                                                                  |

---

Figure 6.1: Log4r Format Specifiers

---

```
- fancy.info('Money, money, money.')
```

produces:

```
[ INFO] 724998: Money, money, money.
```

Still unsatisfied? Ok, then let's write a completely new Formatter. The following example formats exceptions in an aggressive manner:

File

```
Line 1  class ExceptionFormatter < Log4r::Formatter
-      def format(e)
-          log = "[%5s] %s: %s\n" %
-              [LNAMES[e.level], Time.now, e.data.to_s]
5      if e.data.class == Exception
-          msg = e.data.message
-          size = msg.size
-          line = '!' * (size + 6) + "\n"
-          log << line
10         log << '!' + msg.center(size + 2) + " !\n"
-         log << line
-
-         end
-         log
-     end
15 end
-
- custom = Logger.new('my_first_format')
- outputter = StdoutOutputter.new(
-     'stdout',
20     :formatter => ExceptionFormatter
- )
- custom.add(outputter)
- custom.info(%w(cat mouse dog))
- custom.fatal(Exception.new('ALAAARRRRMMM!'))
25 custom.error('I am so nicely formatted!')
```

produces:

```
[ INFO] Sun Sep 25 18:17:12 CEST 2005: catmousedog
[FATAL] Sun Sep 25 18:17:12 CEST 2005: ALAAARRRRMMM!
!!!!!!!!!!!!!!!!!!!!!!
!  ALAAARRRRMMM!  !
!!!!!!!!!!!!!!!!!!!!!!
[ERROR] Sun Sep 25 18:17:12 CEST 2005: I am so nicely formatted!
```

Writing your own formatter is simple: Derive a class from `Formatter` and override the `format(event)` method that gets passed the current log event as an instance of `LogEvent`. The `LogEvent` class encapsulates the attributes belonging to a single log message: the log level, the logger name, the message itself, and so on (the `LNAMES` Hash we have used in line 4 maps the numerical log levels to their textual representations). At the end of `format()` return a string in your selected format.

## Outputters

Log messages are pretty useless if they don't appear somewhere. Log4r supports the concept of outputters that receive log messages and “visualizes” them somehow. You can print them to the console, write them to a file, send them via e-mail, or send them to a syslog daemon (see Figure 6.2, on the following page for a complete list).

Each logger can have many outputters. You can add them to the outputters array at any time. In addition to the log level threshold of the Logger, each Outputter has a threshold, too. It can be set with the `only_at(*levels)` method (for example, `outputter.only_at(ERROR, FATAL)`).

Log4r comes with a lot of configurable outputters. Usually these will do everything you need. However, you can also create your own outputters if you need something special.

The following class implements a buffered outputter that consumes a certain number of log events, then logs them all at once whenever a configurable threshold value has been reached or a fatal event has been logged:

File

```

Line 1  class BufferedOutputter < IOOutputter
-       def initialize(name, io, options = {})
-         super(name, io, options)
-         @out = io
5         @last_events = []
-         @max_events = options[:max_events] || 50
-       end
-
-       def canonical_log(event)
10        @last_events << event
-         if @last_events.size >= @max_events ||
-           LNAME$[event.level] == 'FATAL'
-           @last_events.each { |e| super(e) }
-           @last_events = []
15        end
-       end
-     end
-   end

-   logger = Logger.new('buffered-outputter')
20   outputter = BufferedOutputter.new(
-     'buffered',
-     $stdout,
-     :max_events => 2
-   )
25   logger.outputters = outputter
-   logger.debug('I cannot be seen immediately!')
-   logger.info('Now both are there!')
```

| Outputter            | Purpose                                                                                                                                                    |
|----------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| IOOutputter          | Logs to an arbitrary IO object.                                                                                                                            |
| StdoutOutputter      | Logs to STDOUT.                                                                                                                                            |
| StderrOutputter      | Logs to STDERR.                                                                                                                                            |
| FileOutputter        | Logs to a single file.                                                                                                                                     |
| RollingFileOutputter | Logs to a file and automatically maintains a generation of log files depending on their size, that is, files will be closed and renamed at a certain size. |
| DateFileOutputter    | Logs to a file and automatically maintains a generation of log files depending on their timestamp: files will be closed and renamed daily, monthly, etc.   |
| SyslogOutputter      | Logs to the syslog daemon.                                                                                                                                 |
| EmailOutputter       | Sends log messages via e-mail.                                                                                                                             |
| RemoteOutputter      | Sends log messages to a remote LogServer instance.                                                                                                         |

---

Figure 6.2: Log4r Outputters

---

produces:

```
DEBUG buffered-outputter: I cannot be seen immediately!
INFO buffered-outputter: Now both are there!
```

Because `BufferedOutputter` needs a number of IO facilities, we chose to subclass `IOOutputter`, rather than `Outputter`. The `initialize(name,out,options)` method expects the outputter's name, the IO Object to write to, and an optional options Hash.

Beginning on line 9 we override the `canonical_log(logevent)` method. This is responsible for handling a log event that has made its way to our outputter. By the time a log event arrives here it has passed all log level checks and is actually meant to be logged. We add every event to our current list of events. Whenever this list exceeds the maximum size or a fatal event is logged, we delegate the logging of every buffered event to Log4r's default implementation by calling `super()`.

In our main program we create new `Logger` and `BufferedOutputter` objects. We associate the outputter with `$stdout`, so all outputs will appear on the console. To make sure we will see anything, we have set `max_events` to 2.

## Configuration

Until now we have configured all our Loggers manually in the code. For demonstration purposes this is perfect, but in a production system you want to have configuration files that can be safely edited without touching a single line of your program. Log4r gives you the freedom to choose XML or YAML as the format of your configuration files.

Configurator is a central component that controls most of the configuration options (we have already used it in Section 6.2, *Loggers*, on page 255 to configure our own log levels). Its most important methods are `load_xml_file(file_name)` and `load_xml_string(xml_string)`. These are used to configure Log4r using its XML dialect. A typical example looks like this:

File

```
Line 1  <log4r_config>
-      <pre_config>
-        <custom_levels>
-          FINEST, FINER, FINE, CONFIG,
5          INFO, WARNING, SEVERE
-        </custom_levels>
-        <global_level="FINE"/>
-      </pre_config>
```

```
- </log4r_config>
```

Log4r's XML configuration has to be defined under a `<log4r_config>` element. This doesn't have to be the root element in the actual XML file. This allows you to embed the Log4r configuration into your application's global configuration file: you do not have to spread your configuration across several files.

Optionally, the XML file may contain a `<pre_config>` section, where custom log levels, configuration parameters, and a global log level can be defined.

If we store the XML configuration file above in a file called `log4r.xml`, we can use it like this:

File

```
Line 1  Configurator.load_xml_file('log4r.xml')
-      main = Logger.new('main')
-      main.outputters = Outputter.stdout
-      main.finest("That's how my coffee should be!")
5      main.fine("That's ok, too.")
```

produces:

```
FINE main: That's ok, too.
```

Because we have set the global log level to FINE, only the second message gets logged.

Using parameters defined in the `<pre_config>` section we can communicate between the Configurator class and the configuration file:

File

```
<log4r_config>
  <pre_config>
    <parameter name="pattern" value="%l [%d] %m"/>
  </pre_config>

  <outputter name="logfile" level="WARN">
    <type>FileOutputter</type>
    <filename>#{basepath}/main.log</filename>
  </outputter>

  <logger name="main" level="DEBUG" trace="true">
    <outputters>stdout, logfile</outputters>
  </logger>
</log4r_config>
```

We initialize the `basepath` variable used in the configuration file above in our Ruby program:

File

```
Line 1  Configurator['basepath'] = '/tmp'
-      Configurator.load_xml_file('log4r2.xml')
-      main = Logger['main']
```

## XML Parameters In Log4r

It's possible to use Ruby variables in Log4r configuration files and vice versa. To pass a value from Ruby to the configuration you can use the Configurator class like a Hash:

```
Configurator['basepath'] = '/tmp'
```

In the configuration file you can use it like this, then:

```
<log4r_config>
  <pre_config>
    <parameters>
      <basepath>#{basepath}</basepath>
    </parameters>
    <parameter name="pattern" value="%l] %d: %m"/>
  </pre_config>
</log4r_config>
```

On the other side you can access parameters defined in the `<pre_config>` section of the XML configuration file in your Ruby program. For example, you can access the pattern parameter defined previously like this:

```
puts Configurator['pattern']
```

Most Log4r configuration parameters can be expressed in two ways, namely as a value= attribute or as a child element. For example, the following definition:

```
<outputter name="main" filename="./log/main.log"/>
```

is equivalent to:

```
<outputter>
  <name>main</name>
  <filename>./log/main.log</filename>
</outputter>
```

```
- main.debug('On console only.')
5 main.warn('On console and in file.')
```

This is where the fun begins! We have configured a Logger with two outputters for the first time. One prints to STDOUT and the other one into a file (/tmp/main.log). On the console we have

```
DEBUG main(pre_config_parameters.rb:11): On console only.
WARN main(pre_config_parameters.rb:12): On console and in file.
```

and /tmp/main.log contains

```
WARN main(pre_config_parameters.rb:12): On console and in file.
```

There are configuration sections for every Log4r object: loggers, formatters, and outputters. As a rule of thumb all attributes we have used as we have configured our objects manually are also available in the configuration files. Let's examine a complete example:

```

File
Line 1  <log4r_config>
-       <pre_config>
-         <global level="WARN"/>
-         <parameter name="pattern" value="%l [%d] %m"/>
5       </pre_config>
-
-       <outputter name="console">
-         <type>StdoutOutputter</type>
-         <formatter type="PatternFormatter" pattern="#{pattern}">
10         <date_pattern>%Y-%m-%d %H:%M:%S</date_pattern>
-         </formatter>
-       </outputter>
-
-       <outputter name="logfile" level="ERROR">
15       <type>RollingFileOutputter</type>
-       <filename>#{basepath}/main.log</filename>
-       <maxsize>1048576</maxsize>
-       <count>5</count>
-       <trunc>false</trunc>
20       <formatter type="PatternFormatter" pattern="#{pattern}">
-       <date_method>usec</date_method>
-       </formatter>
-     </outputter>
-
25     <logger name="app">
-       <outputter>console</outputter>
-     </logger>
-
-     <logger name="db" outputters="logfile"/>
30 </log4r_config>

```

Here we define two outputters called console and logfile. The first one writes its output to the console using a StdoutOutputter, the other uses a RollingFileOutputter that keeps up to 5 log files each having a maximum size of 1MB. Both outputters use a PatternFormatter having the same format. Only the date format differs: for the console we use the date\_pattern modifier and for the log file we use date\_method.

Then we define two loggers called app and db. app only logs to the console and db logs into a log file.

Running the following snippet

```

File
Line 1  require 'log4r/outputter/rollingfileoutputter'
-       include Log4r

```



```

- Configurator['basepath'] = '/tmp'
- Configurator.load_xml_file('complete.xml')
5
- app = Logger['app']
- app.debug('Look at me!')
- app.warn('Look at Roy!')
-
10 db = Logger['db']
- db.error('DB logs go always into a file.')
- db.warn('But not this one.')

```

leaves these messages on our console:

```
WARN [471281] Look at Roy!
```

Because we have set the global log level to WARN in the `<pre_config>` section, only the second message sent to the `app` object gets logged.

Our log file contains something like this:

```
ERROR [163338] DB logs go always into a file.
```

Only the first message is written, because the log level of the logfile outputter is ERROR.

You have to require all Log4r classes you have used in your configuration to prevent problems during the configuration process. For example, we had to explicitly require 'log4r/outputter/rollingfileoutputter'. Annoyingly, for the rolling file outputter Log4r creates an empty `/tmp/main.log` file and begins the actual logging in a file called `/tmp/main000001.log`. Note that we have set `trunc` to `false`. If we hadn't, our new log file would be truncated whenever we restart our program. That's usually not what you want.

For those who prefer YAML over XML's verbosity Log4r has the `YamlConfigurator`. Let's translate our example above into YAML (see Section 3.5, *YAML Ain't Markup Language (YAML)*, on page 134 if you are not familiar with YAML):

```

log4r_config:
  pre_config:
    global:
      level: WARN
    my_pattern:
      - &pat "%l [%d] %m"

  loggers:
    - name      : app
      outputters:
        - stdout

```

File

```

- name      : db
  outputters:
    - logfile

outputters:
- type      : StdoutOutputter
  name      : stdout
  formatter:
    date_pattern: '%Y-%m-%d %H:%M:%S'
    pattern     : *pat
    type        : PatternFormatter

- type      : RollingFileOutputter
  level     : ERROR
  name      : logfile
  maxsize   : '1048576'
  count     : '5'
  date_pattern: '%Y%m%d'
  trunc     : 'false'
  filename  : "#{basepath}/main.log"
  formatter :
    date_method : 'usec'
    pattern     : *pat
    type        : PatternFormatter

```

By replacing only the first two lines of the XML example above we achieve the same results with our YAML configuration.

File

```

Line 1  require 'log4r/yamlconfigurator'
-       YAMLConfigurator['basepath'] = '/tmp'
-       YAMLConfigurator.load_yaml_file('complete.yaml')

```

## Performance Considerations

Logging is an invaluable tool in enterprise environments. But, like most things in life, logging has a dark side. Excessive logging will slow down your processes significantly, because logging usually causes a lot of I/O and I/O is expensive. To prevent this, it's a good idea to execute logging statements only when they are absolutely necessary. To help you do this, Log4r makes it easy for you to set the current log level:

File

```

Line 1  logger = Logger.new('perf')
-       logger.outputters = Outputter.stdout
-
-       long_list = %w(lots of elements)
5       if logger.debug?
-         long_list.each_with_index do |o, i|
-           logger.debug("Element #{i}: #{o}")
-         end

```

```
- end
```

Logging the contents of `long_list`—especially when it’s really long—is an expensive operation, because you have to iterate over all elements and `to_s()` is called each one. Therefore, it makes sense to execute the according method calls only when the debug log level is actually active. It should not be too surprising that all log methods are capable of processing code blocks, too.

## Conclusion

Logger’s API is as simple as it could be. Although it is often tempting, you should not use plain `puts()` statements to print out messages even from simple scripts. Use `Logger` instead, because it’s as easy to use as `puts()` and comes with a lot of benefits.

For bigger software systems that comprise several modules you will run into `Logger`’s limits soon. In these cases, you’re better off using `Log4r` right from the beginning. It is highly configurable and has everything (and even more) you’d expect from a full-blown logging tool.

## 6.3 Creating Daemons and Services

On Unix operating systems, daemons are long-living processes that run in the background and do not have a controlling terminal. Often they are started when a computer is booted and then run forever. Their names usually end with ‘d’, as in `syslogd` or `httpd`. Because daemons are dangerous creatures, you have to tame them with a control script. The Apache web server, for example, comes with the famous `apachectl` script that allows you to start, stop, or restart the server. Daemons exist in the Microsoft Windows world, too, but the superstitious folks in Redmond call them services instead. In the following sections we show you how to create them on both platforms with Ruby.

### UNIX Daemons

According to Stevens’ *Unix Network Programming* [?], you have to do a lot of complicated things to safely create a daemon under Unix. You have to detach from the console, change the current working directory to a specific location (usually to the root directory `/`), set the file creation mask to 0, and close all unneeded file descriptors. Even for Unix wizards, it’s not easy to remember these steps (their order is significant, too). To make it even harder, your work isn’t done after the daemon

is running, because you still need a control script such as `apachectl`. It's therefore very nice that Thomas Uehlinger has created the `Daemons` package<sup>22</sup> that transparently handles all this for us. It turns an arbitrary Ruby script into a daemon process and automatically generates the according control script for starting, restarting, and stopping it.

Do you remember the greeting card server from Section 5.1, *Another Day, Another Protocol*, on page 175? We didn't implement the complete architecture that could be seen in Figure 5.1, on page 177. One process is still missing: the one that actually prints the cards. In this section we will develop a daemon process that will do the job.

Our print process is a little script that reads a particular directory every 10 seconds, sends all `.pdf` files in the directory to a process called `print_card`, and deletes them afterwards.

File

```

Line 1  require 'logger'
-
-  path = ARGV[0] || '/tmp'
-  interval = ARGV[1] || 10
5  interval = interval.to_i
-
-  logfile = File.dirname(__FILE__) + '/watcher.log'
-  logger = Logger.new(logfile)
-  logger.info('Started the watcher...')
10
-  loop do
-    files = Dir["#{path}/*.pdf"]
-    files.each do |filename|
-      logger.info("Processing #{filename}.")
15    %x(print_card #{filename})
-    File.delete(filename)
-  end
-  sleep(interval)
- end

```

Our little observer accepts two command line parameters, the directory to be observed (`/tmp` by default) and the timer interval measured in seconds (10 by default). Then we create a logger to see what our process is doing. In line 7 we determine the full path name of the log file. This is very important, because we are going to turn our script into a daemon process and these processes normally change their working directory to `/` when they start. The `path` variable has to contain an absolute path if we're to put the log file in the correct place..

<sup>22</sup><http://daemons.rubyforge.org>

The rest is fairly easy. We read the directory's content every interval seconds, execute a system command called `print_card`, and delete the file. Now let's turn our script into a full-blown Unix daemon process:

File

```
Line 1  require 'rubygems'
        - require 'daemons'
        - Daemons.run('watcher.rb')
```

Really, that's all we have to do. Using the automatically generated control script, we can start, stop, or restart our process like this:

```
$ ruby watcher_control.rb start
$ ruby watcher_control.rb restart
$ ruby watcher_control.rb stop
```

To perform all this magic, `Daemons` writes the process id into a file called `watcher.rb.pid` and reads it whenever it's needed.

You can pass command line options after a double-hyphen:

```
$ ruby watcher_control.rb start -- /tmp/cards/img 5
```

This command line starts our daemon process and looks for new `.pdf` files in the `/tmp/cards/img` directory every five seconds. It logs its activities into a file called `watcher.log` that lives in the same directory as the `watcher` script itself.

The `Daemons` class allows you to daemonize your programs in several ways. For example, it's possible to turn your script into a daemon by calling `Daemons.daemonize` at any time, but by doing so you lose ability to use the `_control` script to administer your daemon.

For debugging purposes, there is a `run` option that starts the script in the foreground.

## Windows Services

None of the nice things we learned in the preceding section work on the Microsoft Windows platform, because Unix daemons depend on the `fork()` system call which isn't available on Windows. Daniel Berger opened up the Windows platform to Ruby programmers with his `win32utils` project<sup>23</sup>. Part of this project is `win32-service`. This allows us to create daemons (services) for Win32, too. It is not as convenient as the `Daemons` package, because it does not have automatic support for control scripts, but it certainly is helpful.

<sup>23</sup><http://rubyforge.org/projects/win32utils>

Do you remember our status monitor (see Section 4.1, *The PragBouquet Status Monitor*, on page 143) and the SOAP stock server (see Section 5.3, *SOAP*, on page 196)? To make sure that the stock server is working properly, let's create a small script that periodically checks to see if the server is still alive. If it detects any problem, it sends a message to the status monitor. Our checks will not be too sophisticated—we'll only test if the service is physically available, but for a first solution this approach is sufficient.

Because we install the observer script as a Windows service, we have to (partly) implement the interface of a Windows service:

```
File
Line 1  $:.unshift File.dirname(__FILE__)
-   require 'status_monitor_client'
-   require 'logger'
-   require 'soap/rpc/driver'
5   require 'win32/service'
-   include Win32
-
-   class StockServerObserver < Daemon
-     def initialize(opts = {})
10      super()
-      @opts = opts
-     end
-
-     def service_init
15      @interval = @opts[:interval] || 60
-      @logger = Logger.new(@opts[:logfile] || 'c:/observer.log')
-      @stock = SOAP::RPC::Driver.new(
-        @opts[:soap_url] || 'http://localhost:2000',
-        @opts[:soap_urn] || 'urn:Stock'
20      )
-      @stock.add_method('get_report')
-      @sm = StatusMonitorClient.new(
-        @opts[:sm_host] || '127.0.0.1',
-        @opts[:sm_port] || 3333
25      )
-      @logger.info('Observer has been initialized.')
-     end
-
-     def test_stock_service
30      @stock.get_report.class == Hash
-     end
-
-     def service_main
-      @logger.info('Observer has been started.')
35      sleep 1 while state != RUNNING
-      while state == RUNNING
-        if !test_stock_service
-          msg = 'Stock service is not running.'
```

```

-         @logger.warn(msg)
40         @sm.warn('stock', msg)
-     else
-         @logger.info('Stock service is running.')
-     end
-     sleep(@interval)
45 end
-     @logger.info('Observer has been stopped.')
- end
- end

```

All the stuff related to our original task—observing the stock server—is fairly trivial. In the `test_stock_service()` method we perform a simple test to see if the stock server is still alive. We call the `get_report()` method and verify that it returns a Hash object. If the remote procedure call works properly, we can be fairly confident that the physical connection to our stock server is working. If it does not, we send a warning message to the status monitor.

We derived our `StockServerObserver` from the `Win32::Daemon` class, and we overrode the `service_init()` and `service_main()` methods. `service_init()` gets called when a service is initialized—we use it to create clients for the status monitor and the stock server. In addition, we initialize a `Logger` for logging the observer’s status.

The `service_main()` method contains the main logic of the daemon. This is typically an “infinite” loop that waits for external events or performs a task periodically. In our case we call `test_stock_service()` every `@interval` seconds as long as the daemon’s state is `RUNNING`. If `test_stock_service()` returns `false` or an exception is raised, we send a message to the status monitor.

Two things are very important when developing Windows services:

1. Windows services (like UNIX daemons) change their working directory when they start, so you have to specify all files you need in absolute form. To find Ruby libraries outside the standard directories, we add the script’s directory to Ruby’s load path in the first line.
2. You should not rely upon being in a certain state. A lot of things regarding Windows services work asynchronously, because services are controlled from the outside with a control script or the services panel. Hence, in line 35 we wait until the service is actually running, because sometimes it needs a few seconds.

You cannot simply start our current script and expect that it turns itself into a service. Every service has to have a name and it has to be installed properly before starting it.

```
File
Line 1  require 'win32/service'
-       include Win32
-
-       def install_service(name, display_name, executable)
5         service = Service.new
-         service.create_service do |s|
-             s.service_name = name
-             s.display_name = display_name
-             s.binary_path_name = 'ruby ' + File.expand_path(executable)
10          s.dependencies = []
-        end
-        service.close
-        puts "#{name} has been installed."
-      end
```

Call `install_service()` to install any Ruby program you like as a service. Pass it a unique service name, a name that should be displayed in the service list, and the path to a Ruby script that should be installed as a service. The working of the function is fairly obvious, but line 9 is interesting. Here we specify the program that is going to be installed using (as is the case with all files referenced by a service) its absolute file name.

Provided that we have stored our stock observer in a file called `stock_observer.rb`, we can now install it like this:

```
irb(main):002:0> service_name = 'StockObserver'
=> "StockObserver"
irb(main):003:0> install_service(
irb(main):004:1*   service_name,
irb(main):005:1*   'PragBouquet Observer',
irb(main):006:1*   'stock_observer.rb'
irb(main):007:1> )
StockObserver has been installed.
=> nil
```

Our internal service name is *StockObserver*, but in the service list it appears as *PragBouquet Observer*. Even though it's in the service list, it isn't running yet. You could start it using the Windows service control panel (see Figure 6.3, on the next page), but isn't it more fun to do it programmatically?

```
File
Line 1  def start_service(name)
-       Service.start(name)
-       started = false
-       while !started
```



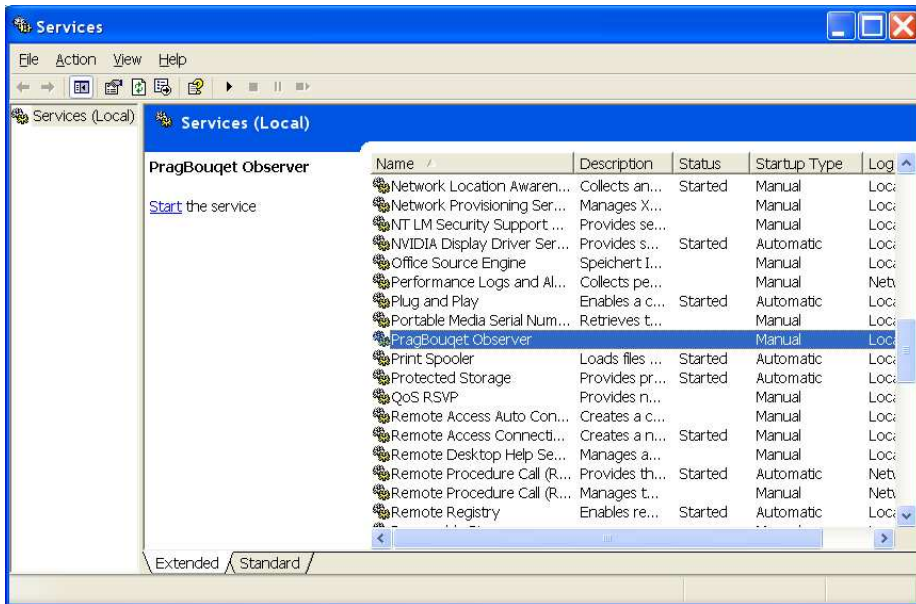


Figure 6.3: PragBouquet Observer in the Service Control Panel

```

5      s = Service.status(name)
-      started = (s.current_state == 'running')
-      break if started
-      puts 'Trying to start service...'
-      sleep 1
10     end
-     puts "#{name} was started."
-     end

```

To safely start a service, call the `start(name)` method of the `Service` class and wait until its status turns to *running*.

```

irb(main):004:0> start_service(service_name)
Trying to start service...
StockObserver was started.
=> nil

```

From now on the stock observer will send SOAP requests to the stock server every minute until the end of time... or until we stop it:

```

Line 1  def stop_service(name)
-      Service.stop(name)
-      stopped = false
-      while !stopped

```

File

```

5      s = Service.status(name)
-      stopped = (s.current_state == 'stopped')
-      break if stopped
-      puts 'Trying to stop service.'
-      sleep 1
10    end
-    puts "#{name} was stopped."
-  end

```

Stopping a service works exactly like starting it: call the `stop()` method of the `Service` class and wait until the service has the *stopped* state.

```

irb(main):005:0> stop_service(service_name)
StockObserver was stopped.
=> nil

```

Finally, we should be prepared to uninstall a service:

```

File
Line 1  def uninstall_service(name)
-      begin
-        Service.stop(name)
-      rescue
5      end
-      Service.delete(name)
-      puts "#{name} was uninstalled."
-    end

```

That's all you need to turn your Ruby programs into Windows services. The `Daemon` class provides many more useful and important methods and the `win32utils` project is full of interesting stuff. If you want to develop Ruby applications on the Windows platform you must have a look at it.

## 6.4 Build and Deployment Process

Compared to deployment processes needed when shipping software to end users on a CD or DVD, deployment processes in enterprise environments are relatively simple. Often, they can be reduced to copying a bunch of files from a development or integration platform to a production host. If anything goes wrong, bugs can be fixed in a single place instead of providing all users with an update.

Additionally, installers for enterprise software do not need colorful wizards with bells and whistles or several hundred kilobytes of End User License Agreements (EULA). Nevertheless, all software—your own and the third party products and the libraries it depends on—has to be deployed somehow. There are several alternatives for Ruby programs.

Many programming languages encourage developers to produce completely self-contained programs that bring everything they need with them. For example, it's not unusual for companies using Java to have several XML parsers (in dozens of versions) lying around on their production systems, because every application ships with its own `xerces-xyz.jar`. Sometimes they are even bundled with their own version of the Java virtual machine. For C++ applications, people often link the required libraries into huge executable images, or risk *DLL hell* by shipping the required shared libraries with each of their applications.

Ruby's (and Perl's, Python's, etc.) approach is different. It's oriented towards a more central installation philosophy, where all libraries are installed only once in the same directory. On UNIX systems this is by default something like `/usr/lib/ruby` or `/usr/local/lib/ruby` and on Microsoft Windows boxes it's `c:\Program Files\ruby\lib` or `c:\ruby\lib`. (If you're looking for the source code of a certain library, these places are good starting points.)

You aren't constrained to have a single, central Ruby installation. You can install it wherever you want (and you can have multiple, independent installations on the same machine). On some platforms it's even possible to turn Ruby programs into self-contained executables that do not even depend on an existing Ruby installation.

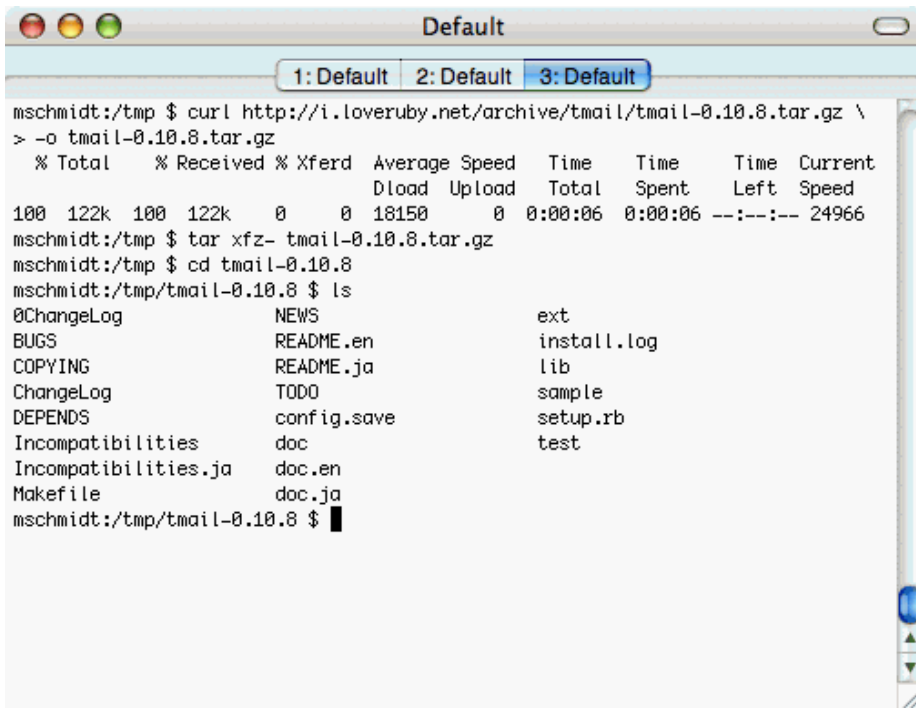
### Deploying With `setup.rb`

Back in the dark and ancient days of Ruby's childhood, nearly every developer who made available a library or script in the Ruby Application Archive (RAA) <sup>24</sup> wrote yet another proprietary installation program that usually copied some files to the central Ruby `lib` directory. These programs were often quick'n'dirty solutions that failed on many platforms: their authors did not know the nitty-gritty details of every environment their software was going to be installed in. Considering that generations of programmers contributed to tools like GNU's `autoconf`, it's not surprising that an individual Ruby coder would struggle with these issues.

Minero Aoki was fed up with the situation. He built a sophisticated solution, called `setup.rb`, that over time became extremely powerful. Before the appearance of the RubyGems project it was the *de facto*

---

<sup>24</sup><http://raa.ruby-lang.org>



```

mschmidt:/tmp $ curl http://i.loveruby.net/archive/tmail/tmail-0.10.8.tar.gz \
> -o tmail-0.10.8.tar.gz
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           Dload  Upload   Total   Spent    Left    Speed
100 122k  100 122k    0     0 18150      0  0:00:06  0:00:06 --:--:-- 24966
mschmidt:/tmp $ tar xzf- tmail-0.10.8.tar.gz
mschmidt:/tmp $ cd tmail-0.10.8
mschmidt:/tmp/tmail-0.10.8 $ ls
ChangeLog      NEWS          ext
BUGS           README.en    install.log
COPYING        README.ja    lib
ChangeLog      TODO         sample
DEPENDS        config.save  setup.rb
Incompatibilities  doc         test
Incompatibilities.ja  doc.en
Makefile       doc.ja
mschmidt:/tmp/tmail-0.10.8 $

```

---

Figure 6.4: Getting tmail From The Internet

---

standard for packaging Ruby software. As a result, there are still hundreds of project out there that depend on it. It's still vital for every serious Ruby programmer to know how it works.

We'll demonstrate the use of `setup.rb` by installing the `tmail` library we used in Section 4.1, “*I’d Rather Use a Socket*”, on page 142 (for those who don’t read books front to back: `tmail` is a library that supports the creation of e-mail messages in SMTP format). Before installing `tmail`, we have to get it somehow. Figure 6.4 shows how to download and unpack the current `tmail` distribution (by the way: the terminal application I’ve used on Mac OS X is `iTerm`).<sup>25</sup> To download the file, of course, you can use `wget`, your favorite web browser, or whatever you prefer instead of the `curl` command I used.<sup>26</sup>

---

<sup>25</sup><http://iterm.sourceforge.net>

<sup>26</sup>Whatever you use, keep in mind that you’d usually have to specify a proxy server in an enterprise environment. For `curl` you’d have to add the option `-x proxy_host:proxy_port`.

Part of the the tar-ball is a file called `setup.rb`. This contains everything needed to install the `tmall` library on an arbitrary platform. The only prerequisites are Ruby and a C compiler. (The C compiler is necessary because `tmall` includes code written in C. If the software to be installed does not contain C extensions, a compiler is not needed.)

`setup.rb` is similar to GNU's `autoconf`; it divides the installation of software in a UNIX environment into three steps. With `autoconf`, you do something like the following:

1. `./configure`
2. `make`
3. `make install`

Step 1 detects the specifics of the current environment: which C compiler is installed, the size of a native int, and so on. The second step builds the software from source. The final step copies everything to its final destination (so this step usually has to be performed by a user having root privileges).

`setup.rb` works in an equivalent fashion—even the meaning of the three steps is the same as in the UNIX installation procedure described previously:<sup>27</sup>

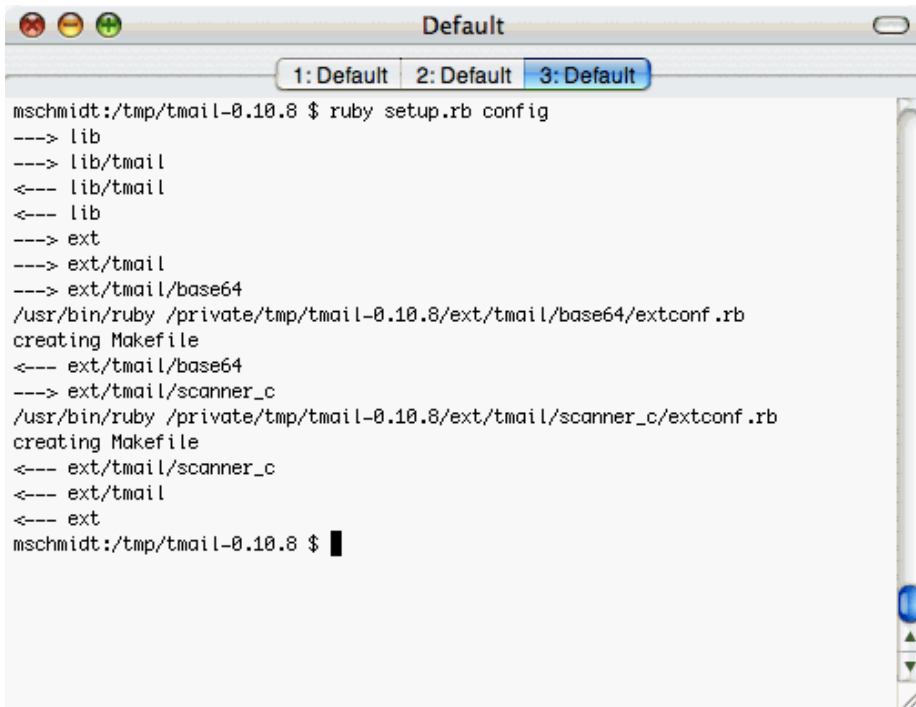
1. `ruby setup.rb config`
2. `ruby setup.rb setup`
3. `ruby setup.rb install`

Let's examine the installation of the `tmall` library step by step. Figure 6.5, on the following page shows the output of running `ruby setup.rb config`. The script automatically explored the environment and created two Makefiles that will be used in the next step to compile all C extensions needed by `tmall`. Platform independent Makefiles are created by `extconf.rb`, which uses the `mkmf` library to do its dirty work. This step's output differs from package to package. For example, the majority of Ruby packages don't depend on C extensions; for these, no Makefiles are created.

Usually, the configuration step just detects where the Ruby installation is located on the current system. Its results are stored in a file called `config.save` that can be edited manually if necessary. On my box it looks

---

<sup>27</sup>In older versions an additional file called `install.rb` was part of the `setup.rb` project. It's not supported any longer, but you will still find it in many older packages.



```

mschmidt:/tmp/tmail-0.10.8 $ ruby setup.rb config
--> lib
--> lib/tmail
<-- lib/tmail
<-- lib
--> ext
--> ext/tmail
--> ext/tmail/base64
/usr/bin/ruby /private/tmp/tmail-0.10.8/ext/tmail/base64/extconf.rb
creating Makefile
<-- ext/tmail/base64
--> ext/tmail/scanner_c
/usr/bin/ruby /private/tmp/tmail-0.10.8/ext/tmail/scanner_c/extconf.rb
creating Makefile
<-- ext/tmail/scanner_c
<-- ext/tmail
<-- ext
mschmidt:/tmp/tmail-0.10.8 $

```

---

Figure 6.5: Configuring tmail

---

as follows (all values starting with a '\$' sign can be configured during the installation process; you can also view the current configuration by running `ruby setup.rb show`):

File

```

bin-dir=$prefix/bin
site-ruby=$prefix/lib/ruby/site_ruby/1.8
prefix=/usr
ruby-path=/usr/bin/ruby
make-prog=make
rb-dir=$site-ruby
without-ext=no
ruby-prog=/usr/bin/ruby
site-ruby-common=$prefix/lib/ruby/site_ruby
std-ruby=$prefix/lib/ruby/1.8
data-dir=$prefix/share
so-dir=$prefix/lib/ruby/site_ruby/1.8/powerpc-darwin8.0

```

tmail depends on two C extensions (one for encoding and decoding of Base64 strings and one for scanning e-mails). In Figure 6.6, on the next page you can see how running `ruby setup.rb setup` compiles these.

```

mschmidt:/tmp/tmail-0.10.8 $ ruby setup.rb setup
--> lib
--> lib/tmail
<--- lib/tmail
<--- lib
--> ext
--> ext/tmail
--> ext/tmail/base64
make
gcc -fno-common -arch ppc -g -Os -pipe -fno-common -arch ppc -pipe -pipe -fno-
common -I. -I/usr/lib/ruby/1.8/powerpc-darwin8.0 -I/usr/lib/ruby/1.8/powerpc-da
rwin8.0 -I/private/tmp/tmail-0.10.8/ext/tmail/base64 -c base64.c
cc -dynamic -bundle -undefined suppress -flat_namespace -L"/usr/lib" -o base64.
bundle base64.o -lpthread -ldl -lobjc
<--- ext/tmail/base64
--> ext/tmail/scanner_c
make
gcc -fno-common -arch ppc -g -Os -pipe -fno-common -arch ppc -pipe -pipe -fno-
common -I. -I/usr/lib/ruby/1.8/powerpc-darwin8.0 -I/usr/lib/ruby/1.8/powerpc-da
rwin8.0 -I/private/tmp/tmail-0.10.8/ext/tmail/scanner_c -c scanner_c.c
cc -dynamic -bundle -undefined suppress -flat_namespace -L"/usr/lib" -o scanner
_c.bundle scanner_c.o -lpthread -ldl -lobjc
<--- ext/tmail/scanner_c
<--- ext/tmail
<--- ext
mschmidt:/tmp/tmail-0.10.8 $

```

---

Figure 6.6: Setting Up tmail

---

In our textbook example everything went fine and the GNU compiler on my Mac happily produced the object files needed. However, if you have even a small amount of experience with UNIX, you'll know that a lot can go wrong during this step. Tools like `autoconf` and `setup.rb` made it much easier to build and install software even on exotic hardware and operating systems, but there are still potential incompatibilities, and you have to be prepared for compiler warnings, errors, and so on.

This is especially true for the Microsoft Windows platform, where often no C compiler or `make` command is available. For popular tools and libraries you can sometimes get pre-compiled binaries, but often you are doomed to have to try to build it yourself. This can be a frustrating (and sleep depriving) experience....

Finally, tmail has to be copied to the standard Ruby library directory. You can see how this can be achieved in Figure 6.7, on the following page. Normally Ruby libraries are installed in a system-wide directory, so the final step has to be performed by a user having root privileges. As good citizens we did not log in as root, but used the sudo command<sup>28</sup> instead.

## Underprivileged?

There are times when you do not have the root password and have no write permission for some directories. There are times when you explicitly do not want to install your software in the standard paths for other reasons. `setup.rb` therefore supports some global options and some task-specific options that allow you to override nearly every setting that would normally be determined automatically. Using these, you can control where stuff goes.

Considering good old UNIX traditions the most important global option is `-q` or `--quiet`. This completely mutes the output of `setup.rb`.

The `--prefix` option is often useful for the `config` and `install` tasks, as it allows you to specify the base directory of the installation. If you want to install a package in the `lib/ruby` directory right under your home directory, for example, run

```
$ ruby setup.rb config --prefix=~/.lib/ruby
```

Afterwards, don't forget to set the environment variable `RUBYLIB` accordingly, or to add the path specified in `--prefix` to your load path (by adding it to `$:`) Otherwise, the Ruby interpreter will not be able to find your freshly installed stuff.

A complete list of all options can be found on `setup.rb`'s web site,<sup>29</sup> but beware: the documentation does not completely match the current state of affairs. For example, the `dll` task is documented but not supported any longer.

## Shipping The Status Monitor

Although it took several pages to demonstrate the usage of `setup.rb`, in reality it's simple. Just run the following commands.

---

<sup>28</sup><http://en.wikipedia.org/wiki/Sudo>

<sup>29</sup><http://i.loveruby.net/en/man/setup/usage.html>





```

mschmidt:/tmp/tmail-0.10.8 $ sudo ruby setup.rb install
--> lib
mkdir -p /usr/lib/ruby/site_ruby/1.8/
install tmail.rb /usr/lib/ruby/site_ruby/1.8/
--> lib/tmail
mkdir -p /usr/lib/ruby/site_ruby/1.8/tmail
install address.rb /usr/lib/ruby/site_ruby/1.8/tmail
install base64.rb /usr/lib/ruby/site_ruby/1.8/tmail
install compat.rb /usr/lib/ruby/site_ruby/1.8/tmail
install config.rb /usr/lib/ruby/site_ruby/1.8/tmail
install encode.rb /usr/lib/ruby/site_ruby/1.8/tmail
install header.rb /usr/lib/ruby/site_ruby/1.8/tmail
install info.rb /usr/lib/ruby/site_ruby/1.8/tmail
install loader.rb /usr/lib/ruby/site_ruby/1.8/tmail
install mail.rb /usr/lib/ruby/site_ruby/1.8/tmail
install mailbox.rb /usr/lib/ruby/site_ruby/1.8/tmail
install mbox.rb /usr/lib/ruby/site_ruby/1.8/tmail
install net.rb /usr/lib/ruby/site_ruby/1.8/tmail
install obsolete.rb /usr/lib/ruby/site_ruby/1.8/tmail
install parser.rb /usr/lib/ruby/site_ruby/1.8/tmail
install port.rb /usr/lib/ruby/site_ruby/1.8/tmail
install scanner.rb /usr/lib/ruby/site_ruby/1.8/tmail
install scanner_r.rb /usr/lib/ruby/site_ruby/1.8/tmail
install stringio.rb /usr/lib/ruby/site_ruby/1.8/tmail
install textutils.rb /usr/lib/ruby/site_ruby/1.8/tmail
install tmail.rb /usr/lib/ruby/site_ruby/1.8/tmail
install utils.rb /usr/lib/ruby/site_ruby/1.8/tmail
<--- lib/tmail
<--- lib
--> ext
--> ext/tmail
--> ext/tmail/base64
mkdir -p /usr/lib/ruby/site_ruby/1.8/powerpc-darwin8.0/tmail
install base64.bundle /usr/lib/ruby/site_ruby/1.8/powerpc-darwin8.0/tmail
<--- ext/tmail/base64
--> ext/tmail/scanner_c
mkdir -p /usr/lib/ruby/site_ruby/1.8/powerpc-darwin8.0/tmail
install scanner_c.bundle /usr/lib/ruby/site_ruby/1.8/powerpc-darwin8.0/tmail
<--- ext/tmail/scanner_c
<--- ext/tmail
<--- ext
mschmidt:/tmp/tmail-0.10.8 $

```

Figure 6.7: Installing tmail

### It's Not Perfect

By default the Firefox browser\* running on my Mac stores files I've downloaded from the Internet in the directory ~/Documents/My Downloads. I never experienced any problems with this setting until I've tried to install tmail from this particular location in the file system. After downloading and unpacking the tar-ball, I ran `ruby setup.rb config` which unexpectedly resulted in the following error message:

```
/usr/bin/ruby: No such file or directory -- ~/Documents/My \
  (LoadError)
'system /usr/bin/ruby ~/Documents/My Downloads/tmail-0.10.8/ \
  ext/tmail/base64/extconf.rb ' failed
Try 'ruby setup.rb --help' for detailed usage.
```

Obviously, `setup.rb` dynamically creates a command line string that gets executed with the `system()` method. Unfortunately, this command line string does not work for directory names containing blanks. The lesson I've learned is: Do not try to run `setup.rb` (or `extconf.rb`) from a directory whose name contains spaces!

---

\*<http://www.mozilla.org/products/firefox>

```
$ curl http://i.loveruby.net/archive/tmail/tmail-0.10.8.tar.gz \
> -o tmail-0.10.8.tar.gz
$ tar xzf- tmail-0.10.8.tar.gz
$ cd tmail-0.10.8
$ ruby setup.rb config
$ ruby setup.rb setup
$ sudo ruby setup.rb install
```

So where does all the magic come from? Even Ruby programmers do not believe in magic (although many of them firmly believe that `matz` is from outer space. Some concepts are just too advanced...). You may ask yourself what you have to do to package your own stuff and make it available on nearly all platforms currently supported? The answer is quite simple: probably nothing if you did what good programmers (and that's all of you, isn't it?) do anyway: You have to put your software into a certain directory structure and you have to add a copy of `setup.rb` to your project. That's it!

`setup.rb` expects you to use the following directory layout:

`project-root/`

**setup.rb destination**

|      |                                         |
|------|-----------------------------------------|
| lib  | /usr/local/lib/site_ruby/1.8            |
| bin  | /usr/bin                                |
| data | /usr/share                              |
| ext  | /usr/local/lib/site_ruby/1.8/i386-linux |

---

Figure 6.8: Typical Directory Mapping For setup.rb

---

```

setup.rb
lib/
bin/
ext/
data/

```

The different directories are mapped to certain destinations. A typical setup for Linux running on an Intel processor can be seen in Figure 6.8. These values will differ from platform to platform, but you should get the idea.

It's OK to put more than one module into a package. Simply add sub-directories to your project's root directory. The classic compiler example looks like this (note: the packages directory is mandatory):

```

my-compiler/
  setup.rb
  packages/  --> mandatory directory name!
    scanner/
      bin/
      lib/
      ext/
      data/
    parser/
      bin/
      lib/
      ext/
      data/
    codegen/
      bin/
      lib/
      ext/

```

data/

So, let's get a bit more concrete and create a package for the status monitor we have developed in Section 4.1, “I’d Rather Use a Socket”, on page 142. Our directory layout is simple:

```
sm/
  setup.rb
  create_smon.sql
  bin/
    control_sm.sh
  lib/
    pre-install.rb
  sm/
    status_monitor.rb
    status_monitor_client.rb
    sms.rb
```

setup.rb was copied verbatim from Minero Aoki’s web site. create\_smon.sql contains everything to create the MySQL database needed by the status monitor:

File

```
create database smon;

use smon;

create table log_entries(
  id int unsigned not null primary key,
  application varchar(64) not null,
  level enum('debug', 'info', 'warn', 'error', 'fatal'),
  message text,
  created timestamp not null
);
```

In the bin directory we placed a shell script called control\_sm.sh that controls the status monitor—it lets the users of the monitor start and stop it.

The lib/sm directory contains all the files we created for the status monitor: the status monitor itself (status\_monitor.rb), the library that encapsulates the access to the SMS server (sms.rb), and the Ruby library for accessing the status monitor (status\_monitor\_client.rb).

There’s only one file left to be explained: pre-install.rb in the lib directory. setup.rb has a nice feature that makes it possible to execute a Ruby script before and after every installation phase. Such scripts are named (pre|post)-(config|setup|install).rb. In our case the script gets executed before the status monitor is installed. Unsurprisingly, the script installs the status monitor database:

File

```

Line 1  # pre-install.rb creates the status monitor database
-      # before installing the status monitor itself.
-
-      system('mysql < ../create_smon.sql')

```

One of the biggest problems with installation processes is testing them, as you normally don't have access to the final production environment. In addition, installation processes often perform tasks that cannot be reverted easily, such as dropping or modifying databases or files. Fortunately, `setup.rb`'s inventor was well aware of these restrictions and added useful options that make testing much easier. Probably the most important one is `--no-harm`, that simulates the installation process without actually touching anything. It will not create or copy any directories or files. In Figure 6.9, on the next page you can see a sample run.

Oops! Didn't we say that nothing would be touched? In the lower third of the screenshot MySQL complains about the existence of our `smon` database. It does already exist, because we have installed it for running our local tests. The problem is that `setup.rb`'s philosophy regarding the pre- and post-script mechanism is a bit more local. It was not primarily designed for processes like installing complete databases, but for creating temporary files that are needed for further steps like generating C code by executing commands like `lex` and `yacc`.

Fortunately, there is the so called "Hook Script" API<sup>30</sup> that allows us to read nearly all options that have been passed to `setup.rb`. For example, calling `get_config('prefix')` in a pre- or post- script determines the current value of the `--prefix` option (`/opt/pb` in our case). Unfortunately, this is only true for configuration options and not for options that have been passed to the different installation steps such as `--no-harm`.

Open Source to the rescue: after examining the source code of `setup.rb`, I quickly discovered that the Hash object `@options` contains everything we need to make the status monitor installation complete:

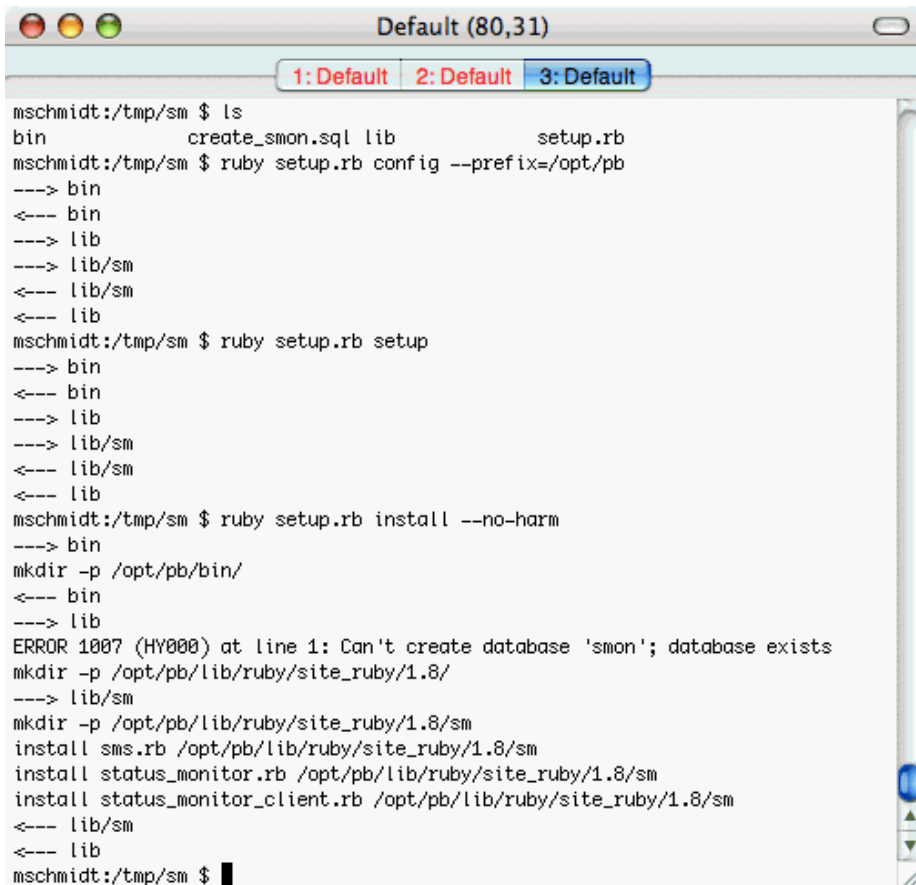
File

```

Line 1  # pre-install.rb creates the status monitor database
-      # before installing the status monitor itself.
-
-      if @options['no-harm']
5         puts 'Normally, we would install the database now!'
-      else
-         system('mysql < ../create_smon.sql')
-      end

```

<sup>30</sup><http://i.loveruby.net/en/man/setup/hookapi.html>



```

mschmidt:/tmp/sm $ ls
bin          create_smon.sql lib          setup.rb
mschmidt:/tmp/sm $ ruby setup.rb config --prefix=/opt/pb
--> bin
<--- bin
--> lib
--> lib/sm
<--- lib/sm
<--- lib
mschmidt:/tmp/sm $ ruby setup.rb setup
--> bin
<--- bin
--> lib
--> lib/sm
<--- lib/sm
<--- lib
mschmidt:/tmp/sm $ ruby setup.rb install --no-harm
--> bin
mkdir -p /opt/pb/bin/
<--- bin
--> lib
ERROR 1007 (HY000) at line 1: Can't create database 'smon'; database exists
mkdir -p /opt/pb/lib/ruby/site_ruby/1.8/
--> lib/sm
mkdir -p /opt/pb/lib/ruby/site_ruby/1.8/sm
install sms.rb /opt/pb/lib/ruby/site_ruby/1.8/sm
install status_monitor.rb /opt/pb/lib/ruby/site_ruby/1.8/sm
install status_monitor_client.rb /opt/pb/lib/ruby/site_ruby/1.8/sm
<--- lib/sm
<--- lib
mschmidt:/tmp/sm $

```

---

Figure 6.9: Simulating Status Monitor Installation

---

From now on only a polite message will be printed to the console when we run `ruby setup.rb install --no-harm`.

Of course, relying on the internals of an implementation like this is generally a bad idea, but in this case we will get away with it, because the probability is low that `setup.rb` is going to change. This is especially true for our local copy.

## RubyGems

As Ruby grew in popularity, and more and more great libraries were released, it became quickly obvious that a full-blown standardized pack-

aging and installation system was needed. `setup.rb` does a fairly good job for typical installation tasks, but it lacks a lot of important features. For example, it is not possible to install different versions of a particular package in parallel. Uninstalling packages isn't possible, either.

To create a more advanced solution David A. Black, Paul Brannan, Chad Fowler, Richard Kilmer, and Jim Weirich sat together on a Ruby conference some years ago and designed (and implemented!) a first version of a new packaging system. They called the packages *Gems* and the package system RubyGems.<sup>31</sup>

Since these modest beginnings, RubyGems has been rewritten completely and greatly enhanced. It has become the de-facto standard for distributing Ruby libraries and applications. At the time of this writing RubyGems was not part of the official Ruby distribution, but chances are good that things have changed by the time you're reading this. You can see if RubyGems is installed already by running `gem list`. This command prints a list of all Gems that are installed on your system. The output should look like this:

```
mschmidt:/tmp> gem list

*** LOCAL GEMS ***

actionmailer (1.1.3, 1.0.1)
  Service layer for easy email delivery and testing.

actionpack (1.11.0, 1.9.1)
  Web-flow and rendering framework putting the VC in MVC.

actionwebservice (0.9.3, 0.8.1)
  Web service support for Action Pack.

activerecord (1.13.0, 1.11.1)
  Implements the ActiveRecord pattern for ORM.
...

xml-simple (1.0.7)
  A very simple API for XML processing.
```

If instead you get an error message like “gem: No such file or directory”, you probably don't have RubyGems installed. You'll need to download and install it. It might be the last Ruby program you have to install manually....

---

<sup>31</sup><http://rubygems.rubyforge.org>

RubyGems comes with an excellent documentation system (try `gem --help` and play around with the options of the `list` command a bit), there's a great user guide on the project's home page,<sup>32</sup> and Chad Fowler has written a chapter about it in [?]. Because of all this, I'm not going to write yet another full tutorial, but I'd like at least to explain briefly how to use RubyGems.

A Gem is a self-contained package that contains everything belonging to a particular Ruby library or application. It doesn't matter if it's a program written in pure Ruby or it's a C extension, if it contains extensive documentation or comes with a big test suite. Everything belonging to the package can be found in the `.gem` file.

In the next section we will need the `rake` command that is available as a Gem, so let's install it now:

```
mschmidt:/tmp> sudo gem install rake
Password:
Attempting local installation of 'rake'
Local gem file not found: rake*.gem
Attempting remote installation of 'rake'
Updating Gem source index for: http://gems.rubyforge.org
Successfully installed rake-0.6.2
Installing RDoc documentation for rake-0.6.2...
```

Because Gems are usually installed in a central directory such as `/usr/local/lib/ruby/gems`, you have to run `gem` as the root user. As good citizens we use the `sudo` command instead of `su`.

RubyGems tries to find a local installation of the `rake` Gem first. If it doesn't find one, it tries to get it from the central Gems server.<sup>33</sup> Then it downloads the package, unpacks it, and installs it. The installation process usually is more than a simple `cp` or `install` command. Often, C extensions and documentation files are built before copying the result to the central Gems directory.

The `update` option installs the most current version of an existing package.

```
mschmidt:/tmp> sudo gem update activerecord
Password:
Upgrading installed gems...
```

---

<sup>32</sup><http://docs.rubygems.org/>

<sup>33</sup>If you're trying to install Gems on a computer in your company you probably have to specify a proxy server:

```
mschmidt:/tmp> gem install -p http://proxy:port rake
```



```

Attempting remote upgrade of activerecord
Attempting remote installation of 'activerecord'
Install required dependency activesupport? [Yn] Y
Successfully installed activerecord-1.13.2
Successfully installed activesupport-1.2.5
Installing RDoc documentation for activerecord-1.13.2...
Installing RDoc documentation for activesupport-1.2.5...
Gems: [activerecord] updated

```

gem automatically downloaded the current ActiveRecord version, along with its dependency ActiveSupport (another important feature of Gems: they can depend on each other and the gem command knows about it). Then, it installed the two libraries and their documentation, which is freshly generated from RDoc comments in the source code.

To find out which versions of ActiveRecord are installed, we use the list option again:

```

mschmidt:/tmp> gem list activerecord

*** LOCAL GEMS ***

activerecord (1.13.2, 1.13.0, 1.11.1)
Implements the ActiveRecord pattern for ORM.

```

There seem to be different versions of ActiveRecord installed on our system. But how can we use them in our Ruby programs? Until RubyGems actually becomes the Ruby packaging standard, you have to call `require 'rubygems'` whenever you want to use a Gem.<sup>34</sup>

```

require 'rubygems'
require 'active_record'

```

Note that even though the Gem is called *activerecord*, we used `'active_record'` and not `'activerecord'` in the require statement.

If you don't want to add `require 'rubygems'` to all of your programs, you can invoke the Ruby interpreter and use the `-r` option:<sup>35</sup>

```

ruby -rubygems . . .

```

Even better, set the RUBYOPT environment variable:

```

mschmidt:/tmp> export RUBYOPT=rubygems

```

---

<sup>34</sup>RubyGems is currently in transitions. It is very likely to become built-in to a future version of Ruby, but until then it allows you to write your require statements as if it already were the standard tool. Someday you only have to remove all the `require 'rubygems'` statements.

<sup>35</sup>Yes, there is a file called `ubygems.rb`.

If you want to use a particular version of a Gem, you have to use the `require_gem(gem,*version_requirements)` function:

```
irb(main):001:0> require 'rubygems'
=> true
irb(main):002:0> require_gem 'activerecord', '= 1.11.1'
=> true
irb(main):003:0>
```

In this case we loaded version 1.11.1 of ActiveRecord. If we try to load a version that is not installed on our system, RubyGems complains with an informative error message:

```
irb(main):003:0> require_gem 'activerecord', '= 1.11.0'
Gem::LoadError: RubyGem version error: activerecord(1.11.1 not = 1.11.0)
```

To specify a particular Gem version, you have to pass a list of version requirements to `require_gem()`. Every version requirement consists of an operator and a version number. The operators are `=`, `!=`, `<`, `>=`, `<=`, and `~>`. Most of them should be self-explanatory, so

```
require_gem 'activerecord', '> 1.1'
```

loads the most current ActiveRecord Gem whose version is greater than 1.1. If instead we'd used:

```
require_gem 'activerecord', '> 1.1', '<= 1.2'
```

RubyGems would load the most current ActiveRecord Gem in the version range 1.1–1.2.

However, the `'~>'`-operator (read: approximately greater than) looks unfamiliar. It's called the *pessimistic operator*, because it makes the pessimistic assumption that major releases of software are often incompatible with former releases. Using that reasoning, it's optimistic to

```
require_gem 'activerecord', '> 2.1'
```

because this may end up loading ActiveRecord version 18.7.1, which may no longer support some of the methods that you're using in your script. A more pessimistic approach would be to constraint the version number to start "2."

```
require_gem 'activerecord', '> 2.1', '< 3.0'
```

That's what the pessimistic operator does. The following is (in effect) identical to the previous code:

```
require_gem 'activerecord', '~> 2.1'
```

RubyGems provides many more useful features for the Gem user. You can also write your own Gems. All this isn't necessary to get your enterprise integration software up and running, but if you want to become a real Ruby hacker, you'd better read the RubyGems manual.

## Conclusion

When creating Open Source software that will be disseminated via Internet, RubyGems is the best choice today—it has been designed for this specific purpose. For the deployment of enterprise software `setup.rb` is still a useful tool.

`setup.rb` dictates a directory layout you have probably used anyway and there is no reason why `setup.rb` and RubyGems should not coexist peacefully.

## 6.5 Project Automation with Rake

In the lifetime of every software project there comes a point when you have to automate things. It might be sufficient to run the compiler manually and to copy the resulting executable to its final destination by hand if you have to compile only two files. But as soon as the number of source files reaches a critical mass (i.e. three files), you'd better think about automating tasks like compiling, linking, packaging, and so on.

When developing software, impatience is a good guide. Whenever you're working with a compiled programming language such as C/C++, Java, or C#, you think about project automation automatically to decrease compile time. It does not make sense to compile every single file every time. Instead you want to compile only those files that are newer than their corresponding binary objects. Integrated Development Environments (IDE) such as Eclipse and tools such as `make` manage these things for you.

With interpreted languages like Ruby this compilation and build management isn't necessary. There are, however, still some tasks that have to be managed the “make way”. For example, if you use SOAP, you'd like to call `wsdl2ruby.rb` (see Section 5.3, *Web Services Description Language*, on page 204) only when it's necessary.

Also, compiling files is not the only task that can be automated in a typical software project. Usually, you have to generate documentation

using `rdoc`, package and deploy your software, and run all unit tests. Often, these tasks depend on each other and have to be performed in a particular order.

Many project automation tools exist: `make`,<sup>36</sup> `Ant`,<sup>37</sup> and `A-A-P`,<sup>38</sup> to name a few. Although they differ slightly in functionality, they have one thing in common: their input file formats are rarely convenient. For example, `make` depends on significant whitespace characters and `Ant` expects you to write bloated XML documents. For simple tasks that is OK, but as soon as you have to implement some logic, you're doomed.

Jim Weirich was fed up with this situation. Because he likes Ruby, he has developed a project automation tool called `Rake`.<sup>39</sup> Jim did not invent yet another input file format. Instead, he used Ruby to drive his build tool. This Ruby lives inside things called *Rakefiles*.

## Rakefiles

Before this section gets too theoretical (after all, this is a pragmatic book), we'd better run `rake` and see what happens:

```
mschmidt:/tmp> rake
rake aborted!
No Rakefile found (looking for: rakefile, Rakefile, rakefile.rb, \
Rakefile.rb)
/usr/local/lib/ruby/gems/1.8/gems/rake-0.6.2/lib/rake.rb:1373:in \
'load_rakefile'
```

`rake` automatically searches for an input file called `rakefile`, `Rakefile`, `rakefile.rb`, or `Rakefile.rb`. From the fact that it searches for files with the extension `.rb` we conclude that it expects a Ruby file. Let's create our first `Rakefile`:

```
puts 'Hello, Rake!'
```

And run `rake` again:

```
mschmidt:/tmp> rake
(in /tmp)
Hello, Rake!
rake aborted!
Don't know how to build task 'default'
```

---

<sup>36</sup><http://www.gnu.org/software/make>

<sup>37</sup><http://ant.apache.org>

<sup>38</sup><http://www.a-a-p.org>

<sup>39</sup><http://rake.rubyforge.org>

We're getting closer. `rake` has executed our Ruby code, but complains that we did not specify a task called 'default'. So, what is a Rake task and what is the default task?

Rakefiles consist of ordinary Ruby code that uses a set of methods and classes defined by Rake. Typically, a Rakefile contains several tasks. Each has a name and is associated with a block of Ruby code. Tasks can depend on other tasks—a task may have some prerequisites that have to be fulfilled before it gets executed. If you do not specify a task to be executed, the task called 'default' is run. Tasks can be defined with the `task(args,&block)` method. Here's the new version of our Rakefile:

```
task :default do
  puts 'Hello, Rake!'
end
```

This time everything works as expected:

```
mschmidt:/tmp> rake
(in /tmp)
Hello, Rake!
```

In most Rakefiles the default task doesn't have code in its body (Rake calls this body an *action*). Instead, the default task will list other tasks as dependencies, and these tasks will run by default when `rake` is executed without any overriding parameters.

Let's refactor our Rakefile and store it in a file called `hello.rb`:

File

```
Line 1  # The refactored Rakefile.
-      task :default => [:hello]
-
-      desc 'Prints a nice greeting!'
5      task :hello do
-        puts 'Hello, Rake!'
-      end
```

This way we can reuse the ingenious 'hello' task and we can run it separately: `rake -f hello.rb hello`.<sup>40</sup> The 'default' task was reduced to a list of dependencies that consists of only a single element in our case.

In addition, we have written a description for the *hello* task with the `desc` task. These descriptions will be output when you pass the `-I` option to `rake`. With `-f` you can set the Rakefile to be used and the `-P` option will print all tasks and their prerequisites:

---

<sup>40</sup>It's possible to specify a list of tasks on the command line that should be executed:  
`rake task1 task2....`

### Some Syntax Notes

Rakefiles tend to look nice. In part, that's because they use some of Ruby's syntactic sugar. There are two things that often confuse unexperienced Ruby programmers when they work with Rake:

1. The prerequisites of a task are passed as a Hash with the => notation:

```
task :default => [:hello]
```

This is the same as

```
prerequisites = Hash.new
prerequisites[:default] = [:hello]
task(prerequisites)
```

2. Because of the different precedence of do/end blocks and their curly counterparts ({ }), you have to be careful when using the latter. The following statement produces a parse error:

```
task :default { puts 'Hello, Rake!' }
```

The code block is associated with the :default symbol and not with the task() method. Put :default into parentheses and everything is fine again:

```
task(:default) { puts 'Hello, Rake!' }
```

I recommend to not use the curly braces syntax in Rakefiles.

```
mschmidt:/tmp> rake -f hello.rb -T
(in /tmp)
rake hello # Prints a nice greeting!
mschmidt:/tmp> rake -f hello.rb -P
(in /tmp)
rake default
  hello
rake hello
```

### Tasks and Actions

Having a mechanism for defining tasks and their dependencies is useful. It's even more useful that you can implement the actions needed when executing in the same Ruby code. And, to get you started, Rake comes with a lot of useful predefined tasks. We'll use one of these,

the PackageTask, to create a package of the Ruby files belonging to our status monitor. Here's the content of the lib directory:

```
mschmidt:~/work/sm> ls lib
StatusMonitor.java      status_monitor.rb
status_monitor.pm      status_monitor_client.rb
```

We'd like to create a .tgz file that contains all the Ruby files in the lib directory (along with the directory itself). The Rakefile which does this looks like this:

File

```
Line 1  require 'rake'
-       require 'rake/package_task'
-       include Rake
-
5       SM_VERSION = '0.0.1'
-
-       PackageTask.new('sm-ruby', SM_VERSION) do |p|
-         p.need_tar = true
-         p.package_files.include('lib/**/*.rb')
10      end
```

First we load the two libraries we need to use the standard Rake tasks. In line 7 we define the *package* task with PackageTask. Its constructor expects the name of the file to be created and the version information that should be appended to the file name. The rest is defined in a code block that gets the current PackageTask instance. By setting the need\_tar attribute to true, we tell the task that we want to have a .tgz file. package\_files is an instance of class FileList which we set to all .rb files below the lib directory (see Section 6.5, *File Lists*, on page 302 for more details). What's the name of the task and how do we call it? -T to the rescue:

```
mschmidt:~/work/sm> rake -T
(in /Users/mschmidt/work/sm)
rake clobber_package # Remove package products
rake package         # Build all the packages
rake repack         # Force a rebuild of the package files
```

Obviously, Rake defines three tasks automatically for us as soon as we define a package task. So, let's use them:<sup>41</sup>

```
mschmidt:~/work/sm> rake package
(in /Users/mschmidt/work/sm)
mkdir -p pkg
```

<sup>41</sup>At the time of writing, Rake has some problems with Ruby 1.8.4. If you run into any problems (such as "no such option: noop") try to install the latest version. `gem update rake`.

```

mkdir -p pkg/sm-ruby-0.0.1/lib
rm -f pkg/sm-ruby-0.0.1/lib/status_monitor.rb
ln lib/status_monitor.rb pkg/sm-ruby-0.0.1/lib/status_monitor.rb
rm -f pkg/sm-ruby-0.0.1/lib/status_monitor_client.rb
ln lib/status_monitor_client.rb \
  pkg/sm-ruby-0.0.1/lib/status_monitor_client.rb
cd pkg
tar zcvf sm-ruby-0.0.1.tgz sm-ruby-0.0.1
sm-ruby-0.0.1/
sm-ruby-0.0.1/lib/
sm-ruby-0.0.1/lib/status_monitor.rb
sm-ruby-0.0.1/lib/status_monitor_client.rb
cd -
mschmidt:~/work/sm> ls pkg
sm-ruby-0.0.1          sm-ruby-0.0.1.tgz
mschmidt:~/work/sm>

```

To execute the package task, Rake calls standard UNIX commands such as `cd`, `ln`, and `rm`. Do you need a `.zip` file, too? Add `p.need_zip = true` to the `PackageTask` task and run it again:

```

mschmidt:~/work/sm> rake package
(in /Users/mschmidt/work/sm)
cd pkg
zip -r sm-ruby-0.0.1.zip sm-ruby-0.0.1
  adding: sm-ruby-0.0.1/ (stored 0%)
  adding: sm-ruby-0.0.1/lib/ (stored 0%)
  adding: sm-ruby-0.0.1/lib/status_monitor.rb (deflated 43%)
  adding: sm-ruby-0.0.1/lib/status_monitor_client.rb (deflated 50%)
cd -
mschmidt:~/work/sm>

```

Rake creates the `.zip` file and is clever enough to realize that the `.tgz` file does not have to be created again. Anyway, we will clean up our mess:

```

mschmidt:~/work/sm> rake clobber_package
(in /Users/mschmidt/work/sm)
rm -r pkg
mschmidt:~/work/sm>

```

But what if you want to have different package tasks? Maybe we need one that creates the status monitor files for Ruby programmers, and another one that creates it for Java programmers? Rake automatically gives the package task a name and there is no way to set it. But Rake has a much better solution: *namespaces*.

*namespaces*

File

```

Line 1  require 'rake'
-       require 'rake/packagetask'
-       include Rake
-
5  SM_VERSION = '0.0.1'

```



```

-
- namespace :ruby do
-   PackageTask.new('sm-ruby', SM_VERSION) do |p|
-     p.need_tar = true
10    p.need_zip = true
-     p.package_files.include('lib/**/*.rb')
-   end
- end
-
15 namespace :java do
-   PackageTask.new('sm-java', SM_VERSION) do |p|
-     p.need_tar = true
-     p.need_zip = true
-     p.package_files.include('lib/**/*.java')
20    p.package_files.include('build.xml')
-   end
- end

```

You can wrap any task in a namespace. Rake automatically puts this namespace in front of the task name (separated by a colon). This time we have defined two package tasks, one for the Ruby package and one for the Java package. You can use them like this:

```

mschmidt:~/work/sm> rake -T
(in /Users/mschmidt/work/sm)
rake java:clobber_package # Remove package products
rake java:package         # Build all the packages
rake java:repackage       # Force a rebuild of the package files
rake ruby:clobber_package # Remove package products
rake ruby:package         # Build all the packages
rake ruby:repackage       # Force a rebuild of the package files

```

Rake comes with a lot of standard tasks:

- The `file` task creates a file from a list of other files. In C/C++ projects, for example, an executable is built by linking a list of object files.

```

file 'game' => %w(alien.o joystick.o screen.o sound.o) do |t|
  sh "c++ -o #{t.name} #{t.prerequisites.join(' ')}"
end

```

Whenever one of the object files is more recent than our executable, the `game` file is rebuilt. We build it by invoking the C++ compiler with the `sh()` method. This method, which comes with Rake, simply executes shell commands. The code block gets passed the task and we use some of its properties (the task's name and prerequisites) to create the command line string.

Note we did not specify how to create the object files themselves.

You could do this with a file task for every object file, but we'll show you a better way using the rule task.

- The rule task specifies what Rake should do if it encounters a file name that has no task associated with it. For example, you can define how to create `.o` files from `.cc` files:

```
rule '.o' => ['.cc'] do |t|
  sh "c++ -c #{t.source} -o #{t.name}"
end
```

You can access the names of both the source and the destination files in the task's action. We use them to invoke the C++ compiler with Rake's `sh()` method. `rule` works recursively; if it cannot find the `.cc` file it needs, it would try to find a rule for creating `.cc` files. This is useful when you're working with code generators such as `lex` and `yacc`.

Regular expressions can be used to specify the output files and the source files can be determined by a code block. Our last rule was an abbreviation for<sup>42</sup>

```
rule(/.o$/ => [ proc { |dest| dest.sub(/\.[^.]+$/, '.cc') } ]) do |t|
  sh "c++ -c #{t.source} -o #{t.name}"
end
```

- The directory task creates paths in the filesystem. For example, the `:default` task in the following Rakefile

File

```
require 'rake'
data_dir = 'pkg/test/data'
task :default => [data_dir]
directory data_dir
```

will create the directory `pkg/test/data` if it does not already exist.

- The `clean` and `clobber` tasks can be used to clean up the working directory. `clean` removes all *temporary* files that have been created during the build process and `clobber` removes *all* files that have been added to the original package. The files to be removed are specified by the file lists `CLEAN` and `CLOBBER`, respectively. (`CLOBBER` always contains all the files in `CLEAN`.)

```
require 'rake/clean'
CLEAN.add('**/*.o')
CLOBBER.add('**/*~', '**/*.bak', '**/*.tmp')
```

<sup>42</sup>In this case it's necessary to put the arguments of `rule()` in brackets.

- RDoc is the de-facto standard for documenting Ruby code. The RDocTask task runs the rdoc command to create documentation from the comments in your source files. A typical RDocTask looks as follows:

```
Rake::RDocTask.new(:docs) do |rd|
  rd.main = 'README.rdoc'
  rd.rdoc_files.add('README.rdoc', 'lib/**/*.rb')
  rd.options << '--all'
end
```

In this case we gave the RDocTask the name docs. Rake automatically creates two additional tasks called clobber\_docs (remove the generated documentation), and redocs (force a rebuild of the documentation) for us.

- You can create a Gem for your project with the GemPackageTask. It works exactly like the PackageTask. The only difference is that it expects a Gem specification:

File

```
Line 1  require 'rubygems'
-       require 'rake/gempackagetask'
-       include Rake
-
5       SM_VERSION = '0.0.1'
gem_spec = Gem::Specification.new do |s|
-       s.name = 'sm'
-       s.version = SM_VERSION
-       s.platform = Gem::Platform::RUBY
10      s.files = FileList['lib/**/*.rb']
-       s.requirements << 'none'
-       s.require_path = 'lib'
-       s.autorequire = 'status_monitor_client'
-       s.summary = 'Status monitor for PragBouquet applications.'
15      s.description = 'Use the status monitor to report errors.'
-       end
-
-       GemPackageTask.new(gem_spec) do |pkg|
-         pkg.need_zip = true
20         pkg.need_tar = true
-       end
```

The preceding Rakefile creates a .tgz, a .zip, and a .gem version of our status monitor package:

```
mschmidt:~/work/sm> rake package
(in /Users/mschmidt/work/sm)
mkdir -p pkg
mkdir -p pkg/sm-0.0.1/lib
rm -f pkg/sm-0.0.1/lib/status_monitor.rb
ln lib/status_monitor.rb pkg/sm-0.0.1/lib/status_monitor.rb
```

```

rm -f pkg/sm-0.0.1/lib/status_monitor_client.rb
ln lib/status_monitor_client.rb \
  pkg/sm-0.0.1/lib/status_monitor_client.rb
cd pkg
tar zcvf sm-0.0.1.tgz sm-0.0.1
sm-0.0.1/
sm-0.0.1/lib/
sm-0.0.1/lib/status_monitor.rb
sm-0.0.1/lib/status_monitor_client.rb
cd -
cd pkg
zip -r sm-0.0.1.zip sm-0.0.1
  adding: sm-0.0.1/ (stored 0%)
  adding: sm-0.0.1/lib/ (stored 0%)
  adding: sm-0.0.1/lib/status_monitor.rb (deflated 43%)
  adding: sm-0.0.1/lib/status_monitor_client.rb (deflated 50%)
cd -
  Successfully built RubyGem
  Name: sm
  Version: 0.0.1
  File: sm-0.0.1.gem
mv sm-0.0.1.gem pkg/sm-0.0.1.gem
mschmidt:~/work/sm> ls pkg/
sm-0.0.1          sm-0.0.1.gem      sm-0.0.1.tgz      sm-0.0.1.zip

```

You can find more tasks on the internet. For example, there are tasks that create code statistics or publish files with ssh. Before you write a new task, find out if someone else had a similar problem and solved it already.

## File Lists

Managing software projects often comes down to managing files. Files have to be created, removed, and compiled. Files depend on each other. So it's only natural that Rake supports the handling of file lists with a separate class called `FileList`.

`FileList` objects are arrays with some additional methods for managing files specified by file name patterns. These patterns are evaluated lazily—when the first object is requested from a `FileList` object. In a typical Java project, you will probably find a file list that looks like this:

```
files = FileList['build.xml', 'lib/**/*.jar', 'src/**/*.java']
```

Once a `FileList` has been created, more files can be added with both the `add(*filenames)` or `include(*filenames)` methods.

```
files.add('README')
files.include('MANIFEST')
```

If you pass an array to any of these methods, every element of the array is added to the file list. Excluding files is possible, too:

```
files.exclude('src/**/Test*.java')
```

By default, file lists will ignore files:

- containing 'CVS' or '.svn' in the file path
- ending with .bak or '~'
- named 'core'

You can clear the default exclude pattern with the `clear_ignore_patterns()` method. Calling `select_default_ignore_patterns()` sets it back to the initial default value.

When automating software project tasks you often have to rename files or change their extensions. `FileList` supports these actions, too.

File

```
Line 1  files = FileList['lib/**/*.rb']
-      puts "Original files:"
-      puts files
-      puts
5
-      backup_files = files.gsub(/$/, '.bak')
-      puts "Backup files:"
-      puts backup_files
-      puts
10
-      rhtml_files = files.ext('.rhtml')
-      puts "rhtml files:"
-      puts rhtml_files
```

Prints (for the directory of the status monitor):

Original files:

```
lib/status_monitor.rb
lib/status_monitor_client.rb
```

Backup files:

```
lib/status_monitor.rb.bak
lib/status_monitor_client.rb.bak
```

rhtml files:

```
lib/status_monitor.rhtml
lib/status_monitor_client.rhtml
```

`FileList` has more interesting methods. Have a look at them before you start working with Rake. It's an independent class and maybe you can use it in your own projects, too.

## Conclusion

Rake is well accepted in the Ruby community, is actively maintained, and will certainly become more and more popular (perhaps even outside the Ruby world). Interestingly, most of the Rake tutorials you can find on the internet concentrate on topics such as building C/C++ programs. That's nice for demonstrating some of Rake's features, but it's a bit misleading, too. Rake is in an early stage of development and does not work well with tools such as GNU's autoconf, which are important when building portable tools. At the moment I recommend not using Rake for anything other than Ruby projects.

Rake might not even be optimal choice for all Ruby projects. One of the biggest advantages of the Java build tool ant is that it offers useful commands like `ftp` and `scp` that are independent of the underlying operating system. Hence, you do not have to write and maintain `.sh` and `.bat` versions of the same build and deploy tools. Unfortunately, Rake does not work this way. It calls the tools it needs with the `system()` call. If no `tar` command is found in your path, the `PackageTask` won't work. For development teams that are working on different operating systems it might be better to use a platform-independent tool such as Ant (even though its input files are a pain to maintain).

**Missing: legacy  
tests: Fill out  
chapter!**

## 6.6 Testing Legacy Applications

**Missing: need to  
define local sales  
blurb**