Comparative Programming Languages

Assignment two

Ramses De Norre Robin Debruyne Lynn Gyselen
Mathias Spiessens

December 16, 2011

1 Features

Our DSL, which we referred to as Rumadeus, is built as a thin wrapper around the Adameus query system. It is an internal DSL written in ruby (hence the name). We have tried to provide a more readable syntax than the obscure Adameus queries and added some higher level features such as multi-hop bookings and reservations, finding the best priced ticket and others.

Our DSL's main client is a REPL on which commands and their parameters can be given. To start the REPL, one executes in the `src/` directory the following command[1]

```
ruby -I . ./clients/REPL.rb
```

after which you will be greeted by a prompt

```
Rumadeus >
```

A list of available commands and their parameters can be obtained by the `help` or `h` command.

[1] We used ruby versions 1.9.2 and 1.9.3 during development, we cannot guarantee compatibility with other versions because of the general instability of libraries between major versions and our limited time frame.

Below we show and example Rumadeus session which also shows some error handling and corner cases like an empty result set. The parameters can all be entered in a natural format and no parentheses or separators are needed. Furthermore, data such as dates can be entered in a multitude of formats, basically any format which ruby's `Date::parse` method understands.

```
Welcome to Rumadeus!
Type help for a list of commands.

Rumadeus > list_connections 2011-11-05 BRU AFR
(Empty result.)

Rumadeus > list_connections 2012-01-31 TEG AMS
2012-01-31 TEG AMS GWI101 19:30 01:15
2012-01-31 TEG AMS KLM137 18:55 01:05
2012-01-31 TEG AMS DLH169 19:55 01:05

Rumadeus > list_seats GWI101
You did not supply the correct amount of arguments.
Wrong number of arguments (1 for 3).

Rumadeus > list_seats 2012-01-31 GWI101 B
8 175

Rumadeus > list_seats 2012/01/31 GWI101 B
8 175

Rumadeus > list_seats 2012.01.31 GWI101 B
8 175

Rumadeus >
```

Another example that show cases the more high level features of Rumadeus can be found in the file `report/example_scenario.txt`, we did not inline it here because of the large output generated.

A test suite can be found in the `test/` directory and can be run by executing the following command in the `src/` directory (yes, `src/`! Not `test/`):

```
ruby -I . ../test/ts_rumadeus.rb
```

note that the Adameus server should be up and running for some of the tests to succeed.

2 Cool Features

In this section we will mention some nice features of Rumadeus which we would like to highlight.

2.1 Shortest path

Given two airports and a maximum number of hops, Rumadeus can calculate the shortest path in time between these airports. The maximum number of hops is needed because with the current Adameus architecture of one query per telnet connection, a full search of all possible routes was not a feasible option.

If, for example, you want to find the shortest path from Brussels to JFK with a maximum of two transfers on the 31st of January 2012, you could enter the query

```
shortest_with_stops 31/01/2012-01:15 BRU JFK 2 E
```

the system will then make the following suggestion: flight GWI098 from BRU to FRA and flight AAL341 from FRA to JFK. The system takes into account whether there are seats available on the suggested flights such that only flights with seats available are returned. The example used can also be found in the aforementioned file with queries.

2.2 Group bookings over multiple connections

The system has the ability to hold, book and cancel reservations for a variable number of persons and a variable number of connections in a single query. This is also shown in the example queries. Queries which handle such *multi-queries* are always denoted with the keyword `multi`.

2.3 Help!

For a full list of available queries type `help` or `h` in our REPL. The output of help is generated dynamically from the ruby classes by the use of reflection and aliases for the query methods to method names with a common prefix (see further on for more details on this prefix). Help lists all the functionality provided for external use including the input parameters. By giving the parameters meaningful names the user can easily understand the required input when reasonably familiar with the system.

2.4 REPL

The main interface to Rumadeus is a REPL. We preferred this over a script-based DSL because in the given setting it seemed more appropriate to have users dynamically enter commands than having them write up a custom script for every session.

The REPL has a help function and does proper managing of corner cases (like too many or few parameters, unknown commands, wrong formatting of parameters) and error handling.

3 Implementation

The main issue in developing Rumadeus was figuring out how to do method dispatch such that users are presented a number of functions that can be located in multiple

source files. We also wanted to be able to write helper functions in those files without the need to expose those to the user as well. Furthermore the user has to provide parameters in a string-only interface and these all have to be parsed.

To reach these goals, we used ruby's excellent support for dynamic programming and the possibility to hook right into the method dispatch mechanisms by using methods as `public_send` and `method_missing`. In this section we will elaborate a bit on how we used these mechanisms.

3.1 Calling methods from strings

To call a method by means of user provided strings, we used the `public_send` method which allows one to dynamically call a method from a string on any object with given parameters, in the form of strings as well. The user-accessible methods are furthermore all prefixed by a common string, which is used to ensure that only those functions are accessible.

3.2 Dispatch amongst multiple classes

To have method dispatch amongst multiple classes we used the chain of command pattern, which was very easy to implement using ruby's `method_missing`. The main code for this mechanism can be found in the `AbstractQuery` class. In concreto, every query-providing class extends this subclass and specifies, through a template method, a delegate to which method dispatch should go if a given query cannot be found in the current class. By doing so, a request gets forwarded through our chain of query-providing classes until it is either handled somewhere or until it reaches the final class (called `LastResort`) which responds with *unknown command*.

The chain of command makes it easy to extend the DSL by just hooking a new class with extra queries into the existing chain, furthermore an end-user can also easily add in new functionality by e.g. monkey patching the `LastResort` class.

3.3 Help

The help functionality is implemented with the same chain of command as method dispatch and can also mainly be found in `AbstractQuery`. We used the reflective ruby methods `methods`, `instance_method` and `parameters` to find the methods in the current class that start with our query prefix and then obtain their parameters as well. These parameters are then formatted to show whether they are regular, optional or vararg methods, the last two are, respectively, wrapped in parentheses and prepended with a `*`.

By doing this for our entry point of the chain of command and then having each class concatenate the result of the help for the next element of the chain, we obtain a list of all available commands and their parameters.

3.4 The query classes

We provide a concise overview of the different classes responsible for executing queries.

3.4.1 Actions.rb

This class handles and parses the queries that hold, book, look up or cancel a flight in the database. The return values of the database are stored in objects (found in the utilities directory) when the query is successful, otherwise a `Util::ReservationError`, a subclass of ruby's `StandardError`, is thrown. The REPL handles these errors, with their specified message, when they are thrown.

3.4.2 HLActions.rb

Advanced actions concerning multiple flights are provided here. The bookings use a transactional model, this way a roll-back is possible if any booking fails due to, for instance, concurrent access of multiple clients.

3.4.3 Query.rb

In this class the basic queries provided by the Adameus server are wrapped in functions.

3.4.4 HLQuery.rb

This class provides higher level functionality, for example finding the shortest path between two airports on a given date.

4 Ruby Findings

We summarise our first impressions of the ruby language after this project.

4.1 Pros

- Intuitive programming
- Readable language
- Very fast prototyping possible
- Open classes allow for great extensibility (albeit not without risk)
- Method missing and explicit message sending allows to hook directly into the message dispatch mechanism
- Great support for reflection and meta-programming

4.2 Cons

- Weak typing, tests were necessary to avoid erroneous code
- Poor tool support, next to no information (like available methods, types, ...) available at *develop-time* due to the lack of typing and features such as open classes.
- Good for quick prototyping, but it seems to us that it would be more difficult to manage and maintain bigger projects in a language as ruby due to poor refactoring support and the greater risk of wrong usage of a method or class due to the absence of type checking and proper access privilege enforcement.