Assignment Two – Ruby
DYPL, Dynamiska programmeringsspråk
Spring Term 2012

2nd February 2012

Introduction

The exercise is split into two parts: code generation and library extension. They should be implemented as separate modules that can be loaded with require and used independently. The modules should be named `code_generation.rb` and `array_extension.rb`.

Code Generation

The code generation module reads data descriptions from file and creates appropriate classes that know how to parse descriptions of themselves from YAML files. The library extension extends the standard Array class with SQL-like functions to search for (multiple) occurences of objects with certain properties in arrays.

A program using your libraries might look something like this:

```
# Load your array extension lib, which augments Array
require 'array_extension'

# Load your code generation lib, which defined the module
# Model that performs the code generation
require 'code_generation'

# Create a class Person from person.txt
person_class = Model.generate( 'person.txt' )

# Load an array of Person objects from entries.yml
```

```
persons = person_class.load_from_file( 'entries.yml' )

p = persons.select_first_where_name_is( 'David' )
# <Person name='David' ... >
puts p.name # Prints 'David'

ps = persons.select_all_where_age_is_in( 18, 65 )
# [<Person age='63' ... >, <Person age='20' ...> ... ]
```

1 Extending the Array Class

The Array class should be extended with SQL-like features. Extending should be done through the open classes feature—not through subclassing. The new features should be added to the existing Array class.

The Standard Form

The instance methods supported should have the following signature:

```
Array#select_first( :attribute => value )
Array#select_first( :attribute => [...] )
Array#select_first( :name => :attribute,
                    :interval => { :min => value,
                    :max => value2 } )
Array#select_first( :name => :attribute,
                    :interval => { :max => value2 } )
```

The first method returns the first object in the array that responds to the message attribute by returning value. For example, the following piece of code should return "Deeo":

```
["a", "aa", "Deeo", "odeo"].select_first( :size => 4 )
```

(size is a method in strings that return their length. This piece of code returns the first string in the array that returns 4 as a result of a size message.)

The second method returns the first object in the array that responds to the message attribute by returning a value contained in the argument array. Thus, the following piece of code would return "aa":

```
["a", "aa", "Deeo", "odeo"].select_first( :size => [4, 2] )
```

The third and fourth are similar, but works on intervals.

Each method should also exist in a `select_all` version that does not stop and return the first found object, but a list of all objects that correspond to the selection criteria (or an empty list if no such objects are found).

2 Increasing the Usability

Code using the methods above can be made more readable if we allow a different form of invocation. Instead of using

```
select_first( :name => 'Tobias' )
```

it should be possible to simply write

```
select_first_where_name_is( 'Tobias' )
```

which should be equivalent. The more general form is thus:

```
Array#select_first_where_<attribute>_is( value )
Array#select_first_where_<attribute>_is( [] )
Array#select_first_where_<attribute>_is_in( value, value2 )
```

Returning to our second example above, it should thus be equivalent to this piece of code:

```
["a", "aa", "Deeo", "odeo"].select_first_where_size_is([4, 2])
```

which should return "aa".

And for `select_all`:

```
Array#select_all_where_<attribute>_is( value )
Array#select_all_where_<attribute>_is( [] )
Array#select_all_where_<attribute>_is_in( value, value2 )
```

Note that the array cannot possibly know what kinds of values it will hold, so it will not actually have any of the methods just shown. Rather, you will have to trap calls to non-existing methods and in a clever (by looking at their names) way figure out what the intention is, and transforming them into a call on the standard form using the `select_first`, etc. versions. To optimise your code for multiple calls of the same on-existing method, the non-existing method trap should add the missing method to the object. Thus, the first call to

```
["a", "aa", "Deeo", "odeo"].select_first_where_size_is([4, 2])
```

would not only figure out that the caller means `select_first( :size => [4, 2] )`, but also add the method to the array.

Run-Time Code Generation

The run-time code generation part of the assignment will dynamically add classes to a running program given a simple description provided in a text file. The module Model should be implemented for this purpose with the method generate that accepts a file path to a specification file and returns a class object for the generated class.

Syntax of Specification File

Below is a simple description of the syntax of a specification file:

```
title     :Title
attribute :name_1, Type
...
attribute :name_n, Type
constraint :name_1, "boolean expression_1"
constraint :name_1, 'boolean expression_2'
...
constraint :name_n, "boolean expression_m"
```

Where `:Title` should be the class name of the defined class, the attributes should be attributes of the generated class. The accessors and mutators should make sure that the values held by an attribute has the correct type, (expressed here as a class), or nil. If the object has the wrong class, an error should be raised. All ways of creating strings in Ruby should be supported for the boolean expressions.

Last, the constraints specify a set of additional constraints on the objects of the attributes that must be upheld at all times, just as the type. Note that an attribute can have several constraints.

Example:

```
title     :Person
attribute :name, String
attribute :age,  Fixnum
constraint :name, 'name != nil'
constraint :name, 'name.size > 0'
constraint :name, 'name =~ /^[A-Z]/'
constraint :age, 'age >= 0'
```

This code should generate a class Person with the attributes name, a string, and age, a fixed numeral. Whenever name is read or updated, the class

should make sure that name is not nil, that the name is longer than 0 characters, and that it starts with an uppercase letter. Similarly, age must never be negative. Attempts at wrongfully updating an attribute should raise a RuntimeError.

When called, the generate method should parse a file such as the above, and create a person class with an implementation that satisfies all the specified constraints. Importantly, the person class should know how to parse a YAML file containing person entries and return an array with such objects parsed from a file (see the `load_from_file` call in the first example). This should be handled by a class method that accepts as argument the path to a YAML file with the data to be parsed. Entries that do not satisfy the constraints or don't contain all the specified attributes should be ignored. Attributes that are not mentioned in the specification should be ignored.

The loading should be order-preserving, meaning that if A is before B in the YAML-file, then A should be before B in the array returned by `load_from_file` provided A is a valid object in the file. This is YAMLs standard behaviour, so only be careful not to mess that up.

Recap, What Should You Do

1. Create a loadable module called `array_extension` that augments the class Array with the method on the "standard form". The new array should also trap calls to non-existing methods and, if the calls follow the more readable form, be transformed into calls on the standard form, and added to the Array class.

2. Create a lodable module called `code_generation` that defines the module Model with the method generate that reads a file in the specification file format and returns a proper Ruby class object that corresponds to the description. Generated classes should all implement the method `load_from_file`, that loads instance of the class in point from a YAML file into an array that gets returned.

Hints

Note how the syntax of the specification file is actually valid Ruby syntax. The idea is to use some form of eval to avoid having to parse the file—just load it and eval it in the context that understands messages like title, attribute and constraint.

YAML stands for Yet Another Markup Language and there are good Ruby modules for parsing it around. Don't implement your own YAML parser.

Develop the modules one at a time—they should be independent of each other.

There is a Ruby Code & Style article at Artima that is relevant: `http://www.artima.com/rubycs/articles/ruby_as_dsl.html` here

Testing Before Handing In

Tobias has made a test file that does most of the quantitative checking in the grading process. This file might be of great help to you and is available at `http://people.dsv.su.se/~beatrice/DYPL/unittest.rb` here.

This file also helps in churning out bugs due to subtleness in the specification. A high percentage of those who get their program back did not use the unittests.

A Sample YAML File:

```
persons:
  - name        : Alice
    age         : 20
    length      : 175
    weight      : 65
  - name        : Bob
    length      : 192
    weight      : 89
  - name        : Caligula
    age         : 400
    length      : 165
    misc        : Really old guy
  - name        : Daniel
    age         : 63
    length      : 172
    weight      : 70
    misc        : Quite old too
  - name        : Elisabeth
    age         : 32
    length      : 168
    weight      : 68
```