Université catholique de Louvain
Ecole Polytechnique de Louvain

LSINF2335

Programming Paradigms

Dynamic Features in Ruby

Project 2

Professor: Prof. Kim Mens

Teaching assistant: Nicolas Laurent

Contact person: Benoît Duhoux (benoit.duhoux@uclouvain.be)

March 12, 2018

Notice

For this project, in case of problems or issues, as opposed to contacting your regular course assistant you are requested to contact Benoît Duhoux, a research assistant who is conducting research on a topic related to this project. For his contact details cf. the cover page of this project.

Objectives

In this project you will have to extend the Ruby programming language with a *dynamic feature adaptation mechanism* that allows programmers to dynamically install and uninstall fine-grained features into existing Ruby classes at run-time. Such features are just Ruby modules containing methods and attributes. The effect of installing a feature is similar to that of including a module. The code of that feature is added to the class, and therefore immediately visible to instances of that class. The dynamic feature adaptation mechanism goes beyond that of traditional mixin-modules in Ruby, however. Features cannot only be added, but also removed dynamically, putting the original behaviour of the class back in place. As for mixin-modules, a class can be extended with several features, but in addition to adding new behaviour (new methods) or overriding existing behaviour (overwriting existing methods), features can also *refine* existing or previously added behaviour. More specifically, a method defined by some feature can contain a special *proceed* construct which will call the previously defined method with that name (either present in the original class or in a previously added feature), execute it, and then jump back to the current feature to finish its execution.

Running Example

In order to provide a better understanding of this dynamic feature adaptation mechanism, let us illustrate it with a concrete example. Consider the case of simple application that provides a variety of features to correct a text fragment. As default behaviour, before installing any specific feature, no correction will happen. When asked to correct a text fragment the application will simply return the same fragment.

```
1 class Text
2
3   attr_reader :text
```

```
4
5   def initialize(text)
6     @text = text
7   end
8
9   # Default text correction behaviour = do nothing
10  def correct_text
11    @text
12  end
13
14 end
```

To actually correct text fragments, some features that will adapt this default behaviour of the application, must be installed first. A first feature that can be installed is one that removes all invalid characters from the text fragment (i.e., only letters, numbers, whitespaces and punctuation symbols should be allowed). The code of this feature is shown below. Note how it refines the default `correct_text` method through a special *proceed* construct.[1]

```
1  module Clean
2
3    # Specify the class for which this feature is defined
4    adapt_classes :Text
5
6    def correct_text
7      text = proceed()
8      return remove_unknown_chars(text)
9    end
10
11   def remove_unknown_chars(text)
12     # authorized_characters is an auxiliary method defined
13     # in another feature depending on the chose language
14     # (for example, English or French)
15     return text.tr("^#{authorized_characters}", '')
16   end
17
18 end
```

In fact there will be several variants of this feature, since the set of valid

[1] This *proceed* construct is not standard Ruby but part of the language extension you are supposed to implement.

characters may depend on the chosen language. For the English language, unprintable characters or accented characters will be considered as invalid.

```
module EnglishCharacters

  # Specify the class for which this feature is defined
  adapt_classes :Text

  def authorized_characters
    # Printable characters from their decimal values
    authorized_chars = ""
    (32..126).step(1) do
      |dec_char|
      authorized_chars << dec_char.chr
    end
    return authorized_chars
  end

end
```

For the French language, the set of allowed characters is the same as for the English language but accented characters will also be accepted. Depending on what feature is installed, we can correct either English or French text fragments. Note that the variant of the feature that accepts French characters refines the one for the English language using a *proceed*. (We thus assume that, at run-time, the `FrenchCharacters` feature will always be installed in combination with and after the `EnglishCharacters` feature, in order to make its dependency on that other feature work.)

```
module FrenchCharacters

  # Specify the class for which this feature is defined
  adapt_classes :Text

  def authorized_characters
    set_chars = proceed()
    return set_chars << "àâçèéêîôùû"
  end

end
```

Another feature cleans excessive whitespaces in text fragments (either at the beginning, inside, or at the end of the text). Normally there should be no whitespaces at the beginning or end of a text fragment, and only one whitespace between words.

```ruby
module Trim

  # Specify the class for which this feature is defined
  adapt_classes :Text

  def correct_text
    text = proceed()
    return remove_useless_spaces(text)
  end

  def remove_useless_spaces(text)
    return text.strip().gsub(/\s{1,}/, " ")
  end

end
```

For example, assuming we have installed all of the above features (apart from the `FrenchCharacters` feature), correcting the English text fragment " hello Rubèy", would return the corrected text fragment "Hello Ruby."

Two other features are defined to capitalize the first character of a text fragment (i.e. the feature `capitalize.rb`) and to add a valid punctuation at the end of the text if needed (i.e. the feature `punctuation.rb`).

In addition to these features to correct text fragments, we can also define two different features to actually display the text fragment. One such feature prints all characters (even the unprintable characters), which is useful for debugging purposes (i.e. the feature `printing_all_chars.rb`). The other feature just displays the text in standard format (i.e. the feature `printing_standard.rb`).

In the next sections, we will explain the structure of this project and provide some guidance on how you could implement such a dynamic feature adaptation mechanism.

Structure of the project

The structure of the project contains two folders: *feature_execution/* and *text_correctness_app/*.

feature_execution

This folder should contain the implementation of your code to dynamically adapt Ruby classes with features. This folder should contain at least two files: *feature_selector.rb* and *feature_execution_impl.rb* .

The file *feature_selector.rb* should define the class `FeatureSelector`, describing which class the feature must adapt. The class `FeatureSelector` contains two attributes: the name of the feature and the name of the class that the feature must adapt. For example, the expression
`FeatureSelector.new('Clean', 'Text')`
declares that the feature `Clean` adapts the Ruby class `Text`.

The file *feature_execution_impl.rb* should contain a class `FeatureExecutionImpl`. A code template for this class is shown in listing below. The most important method of this class is the method *alter(action, feature_selector)* where *action* is a symbol designating whether you want to adapt (i.e., `:adapt`) or unadapt (`:unadapt`) the code based on the information that the object *feature_selector* provides. For example, if the *action* is `:adapt` and the *feature_selector* is `FeatureSelector.new('Clean', 'Text')` then the code of the `Clean` feature will be added to the class `Text`. If the *action* is `:unadapt` then this feature will be removed from the class `Text`.

```
1  require 'singleton'
2
3  class FeatureExecutionImpl
4    include Singleton
5
6    def alter(action, feature_selector)
7      # TODO add your code here
8    end
9
10   # TODO your auxiliary methods here
11
12 end
```

Feel free to add any additional files to this folder as needed.

text_correctness_app

This folder contains the test case for this project. Its structure is composed of two folders (*features* and *skeleton*) and two files (*main.rb* and *tests.rb*).

The folder *features* contains the definition of all the features for this application, some of which were already exemplified above. A feature is defined in terms of a Ruby module as shown in the listing below (`module PrintingStandard`). Each feature definition starts with a declaration of the classes that this feature will adapt (cf. line 4 in the listing below; note that this declaration can contain one or more classes to adapt). For example, the feature `PrintingStandard` can adapt only the class `Printer`. **Since this macro doesn't exist by default**, you must create it in the folder *feature_execution*. Having to specify explicitly what class(es) a feature is supposed to adapt provides some control to the developer over where this code should get included.

```
1 module PrintingStandard
2
3   # Specify the class for which this feature is defined
4   adapt_classes :Printer
5
6   def printing(text)
7     puts text
8   end
9 end
```

The subfolder *skeleton* contains all the classes of the application with their default behaviour. This simple application consist of only two classes: `Text` and `Printer`. By default, the class `Text` contains one attribute representing the text fragment and one method *correct_text* which simply returns that text fragment. The class `Printer` is a singleton without any default behaviour.

While the file *main.rb* is a script to execute the test case described in this document, based on your implementation, the file *tests.rb* is a test suite based on *Minitest* to verify that your implementation is correct. The listing below shows a fragment of the *main.rb* file and illustrates how our script will call your implementation to adapt and unadapt the feature `PrintingStandard` on the class `Printer`.

```
1 FEATURES_TEXT = {
2   :printing_standard => FeatureSelector.new('
    PrintingStandard', 'Printer')
3 }
4 FeatureExecutionImpl.instance.alter(:adapt, FEATURES_TEXT[:
    printing_standard])
5 Printer.instance.printing("Hello rubyists!")
6 FeatureExecutionImpl.instance.alter(:unadapt, FEATURES_TEXT
    [:printing_standard])
```

The code of this script, the default classes of the application, and the definition of the features, are provided to you. It's up to you to implement the code of the dynamic feature adaptation mechanism to make all this work as expected. Now that we have described the overall architecture of this project, we will provide you some guidance to build the `FeatureExectutionImpl` component to dynamically adapt or unadapt features on some classes.

1 Adding behaviour dynamically

The first functionality to implement is the ability to dynamically add new features to a class. Each method of the feature (in other words, each method of the module) must be installed in the class. For example, the method `printing` of the feature `PrintingStandard` listed above must be copied into the class `Printer` after the installation of the feature `PrintingStandard`.

To adapt a class with a feature, you must call the method *alter(:adapt, feature_selector)* on the single instance of `FeatureExecutionImpl` (cf. line 4 of the listing just above).

During the installation of a feature, if a method of the feature is already defined by a default behaviour in the class to which it is applied, we ask you to overwrite this default behaviour with the new behaviour. However, to be sure you can reset the default behaviour upon uninstalling the feature, this default version must be stores as a backup that can be retrieved later.

2 Deleting behaviour dynamically

Once we can dynamically add features to existing code, you should implement the ability to remove these features, to revert the code to its original behaviour. When we unadapt the behaviour of a feature of a class, we want

to remove dynamically all the methods of the feature that were previously added to the class. If a previous version of that method wasn't declared, the method must be totally removed. Otherwise the original version of that method before the feature was installed must be re-installed.

To unadapt a class with a feature, you must call the method *alter(:unadapt, feature_selector)* on the single instance of `FeatureExecutionImpl` (cf. line 6 of the listing just above).

3 Managing multiple adaptations

Now that you can alter a class to add or remove a single feature, it would be interesting for your implementation to allow for several feature adaptations of a same class. In this step of the project, we thus want you to allow a developer to adapt (resp. unadapt) a class with several successive features that refine the class' behaviour (resp. that remove some of this behaviour).

This implies that, if several features refine a same method, we must store not only the original behaviour of that method as defined by the class, but a history of all the adapted behaviours as defined by each of the features that adapt this class. However, only the last one will be deployed in the code. The history is needed to reinstall the last most-recent adaptation if we remove the last included feature, and so on.

The listing below shows you how you can adapt a class `Text` with two features (`Trim`, `Punctuation`). To unadapt a feature, change the first argument of the method *alert* with the symbol *:unadapt*. Note that it should be possible to unadapt the features added to this class in any order, so not necessarily in the same order as in which the features were added.

```
1 FEATURES_TEXT = {
2   :trim => FeatureSelector.new('Trim', 'Text'),
3   :punctuation => FeatureSelector.new('Punctuation', 'Text')
4 }
5 FeatureExecution.instance.alter(:adapt, FEATURES_TEXT[:trim])
6 FeatureExecution.instance.alter(:adapt, FEATURES_TEXT[:punctuation])
```

4 Adding the *proceed* mechanism

At this stage, you can refine the behaviour of some class by applying multiple adaptations to that class. However, when you adapt a method for that class, this adapted behaviour cannot call the previous version of this method yet, which could be very useful in order to be able to incrementally modify a class' behaviour at runtime. We therefore propose you to extend the language to allow such calls. This mechanism is known as the *proceed* mechanism. It could be regarded as a kind of *super* mechanism, except that the *proceed* doesn't refer to the superclass but to the previously installed adaptation.

For example, when we want to correct a text by combining two features such as illustrated in the listing below, it is interesting to firstly trim the text and then add punctuation to correct the text. The listing shows you how we can use a *proceed* between features, so that the feature added last calls the behaviour of the corresponding method in a previous feature (or in the original code). Assume the feature `Trim` is installed before the feature `Punctuation`. In the stack of adaptations of the method *correct_text* on the class `Text`, you will have the default behaviour of this method as defined on the original class `Text`, then the adaptation of this method provided by the feature `Trim` and finally the adaptation of the feature `Punctuation`. Thus, when we call the method *correct_text*, the execution will firstly execute the adaptation from the feature `Punctuation`. As it calls *proceed*, the application will then call the method of the previous adaptation, i.e. the behaviour defined by the feature `Trim`. In turn, this feature calls *proceed* and will execute the default behaviour of this method (that returns only the text). After that, each of the proceed calls return to execute what remains to be executed.

```
1  module Trim
2    adapt_classes :Text
3
4    def correct_text
5      text = proceed()
6      return remove_useless_spaces(text)
7    end
8
9    def remove_useless_spaces(text)
10     return text.strip().gsub(/\s{1,}/, " ")
11   end
12 end
13
14 module Punctuation
```

```
15   adapt_classes :Text
16
17   def correct_text
18     text = proceed()
19     if not ".?!".include? text.strip()[-1]
20       text << '.'
21     end
22     return text
23   end
24 end
```

As explained before, you must extend the language to manage such a *proceed* mechanism, i.e. the method *proceed* must be integrated as a keyword in the extension of the language.

Modalities and deliverables

In order to succeed for this project with a passing grade, all of the above functionalities of the dynamic feature adaptation mechanism, and the proceed mechanism in particular, **must** be implemented correctly. Both the `main.rb` script as well as the test cases described in the file `tests.rb` should work correctly. To further evaluate your solution, we will run your code on some additional secret tests.

The project deadline is **Thursday the 29th of March, at 23:59**. This is a strict deadline since the lab session on Friday morning the 30th of March will be an experiment that will build upon what you implemented for this mission.

You must post on Moodle (section: Project 2 submission page) a single zip file containing all the code of your project, according to the project structure that was explained in the Structure section above. If you don't follow this structure of we cannot get to run your code, no further review of your solution will be made.

As for the first mission, we do *not* request a separate report for this mission. Your code *is* the report. So make sure it is readable and well commented. Treat your code as a document. Make it readable. Don't hesitate to include small commented paragraphs with explanations, assumptions or implementation choices, where needed. And as before, we will be intransigeant towards plagiarism.