UNIVERSITE CATHOLIQUE DE LOUVAIN

ECOLE POLYTECHNIQUE DE LOUVAIN

A Tool for Offering Immediate Feedback on Violations of Structural Regularities in Ruby source code

Supervisor: MENS Kim
Readers: LAINEZ Marc
LAURENT Nicolas

Thesis submitted for the Master's degree
in computing science
options: Software Engineering and programming Systems
Networking and Security
by SCERAERT Vincent

Louvain-la-Neuve
Academic year 2014-2015

Acknowledgments

I would like to thank my promoter Kim Mens for his constructive comments, critics and advices on my work during this academic year and its feedbacks which allowed me to improve myself as a computer scientist.

Abstract

This work builds upon a scientific paper [1] which describes an approach and tool which allow programmers of object-oriented frameworks or applications to declare and verify structural constraints — called uContracts or usage contracts — on the source code of classes or methods in the object-oriented language Smalltalk. In this thesis we study how to port the approach to the Ruby programming language.

The approach used to develop such a tool for Ruby is to build a *Domain Specific Language* (DSL) on top of the Ruby language. The advantage of a DSL is that it could reduce the learning period of the tool. Indeed, it would make the product more difficult to use if the user must first learn a new language in order to apply it on his code. With a DSL however, the language remains close to the Ruby syntax. The user of our tool just has to declare the constraints in the DSL. The Ruby interpreter will then execute them to verify that his code respects the constraints declared.

To validate the tool, it was tested on itself and on a small database scenario, and its relevance was discussed with professional programmers in the Ruby community. Strengths, limitations and possible improvements of the tool are discussed to assess the quality of the tool.

Our tool is also a validation of the usage contract concept and tries to determine if the concept of uContracts is generic enough to be added in other object-oriented languages.

The code of the tool can be found at `https://bitbucket.org/vincent_sceraert/epl-2014-008/wiki/Home`

Contents

Chapter 1

Introduction

1.1 Context

In the world of Object-Oriented programming and application frameworks, lots of constraints and assumptions made by the programmers, even when well documented, are in fact not checked. In the case of a programming language such as Ruby, where "Convention over configuration" is crucial, we would prefer a way to explicitly check these programming conventions. Whereas lots of measurement and static analysis tools currently exist, they often cannot check constraints at a higher level than what can be expressed by simple pattern matching. It would help Ruby programmers a lot to have an immediate way of checking such implicitly defined programming conventions. Besides, those tools cannot verify structural regularities in the code, focusing on metrics or black box testing.

During the realization of this thesis, we met two Ruby software development companies to try to understand the needs of Ruby programmers and their development practices. This discussion also allowed us to narrow down the scope of the thesis. More specifically, we met the companies Spin42 and Belighted and discussed with them on coding habits and how they could use a tool, such as the one proposed in this thesis work, in their daily work.

1.2 Objectives

Even if lots of alternatives exist in order to perform checks on the code, none of them allow us to verify implicit structural constraints in a source code. In order to achieve such a verification process, the proposed solution in this thesis is the creation of a tool in Ruby, which can execute the verifications. Ruby was chosen for its flexibility and easy modification of the classes.

The tool developed in this thesis is heavily based on a scientific paper[1] introducing the concept and tool of Usage Contracts. A Usage Contract, abbreviated as uContract, is a structural constraint applied on a liable entity. Liable entities are source code entities such as classes/meth-

ods to which the contract is applicable. The constraints express structural regularities such as "all children of class **C** override method *m* which must call *super*" for instance. The main objective of this thesis is to verify whether the concept of uContracts can be applied to another programming languages, and more specifically to create and validate a prototype in Ruby.

The prototype should not be a straightforward porting of the existing Smalltalk tool but should be adapted to fit Ruby's software development practices and be as easy to use as possible by Ruby programmers. Indeed, the way of programming in Ruby is probably different from coding in Smalltalk which may affect the kind of contracts we may want to declare in our tool, as well as how the tool is implemented and used.

The thesis will also try to classify the uContracts that can be expressed in categories in order to assess their genericity or dependence of language, and to present them in an easy to read and understandable catalogue.

1.3 Approach

First, we must mention that the tool will be created with the Ruby 2.0 version. This means that the tool might not be compatible with Ruby 1.9 or other previous versions. The tool will only be tested on the 2.x versions and thus no 1.x compatible version will be created. Besides, the tool will be tested on Windows and Linux computers to be as usable as possible and not limiting the tool to one operating system.

First, we will look at the Smalltalk prototype and list the type usage contracts it defines. Then, we will look what are the usage contracts relevant in Ruby, the ones that cannot be applied in Ruby and think of new types that can suit the Ruby programming language.

Then, we will look at the Ruby mechanisms, such as hook methods, and create a first version of the tool. Weekly, the tool is shown to the promoter (which stands as the client) and modifications are performed according to these weekly meetings.

The usage contracts kept from the first look at the Smalltalk prototype will be written following a simple but useful methodology : we first write unit tests and think about all possible scenarios for the usage contract use. Once written, we write the usage contract verification code and perform all the tests on it.

Once the tool seems to be stable and has a decent list of implemented usage contracts, the tool will perform a verification on itself. Since we know the constraints on the source code of the Ruby tool, we will write the usage contracts and apply them on the tool to check its correctness.

If the tool returns an unexpected behaviour, there are two possible cases :

- the tool didn't respect a structural constraint, which means that the tool can detect violations of structural regularities.
- the constraint is respected, which means that there is an error in the verification code. However, we can fix it because we have a concrete case.

We will also use/create small applications that look like real world applications and use the Ruby tool on them to check if it is generic enough to tackle a more specific domain, for instance an application with a database.

Once the tool is is ready to use, we will show it to Ruby companies and gather their opinions on the usage, defined constraints, etc. in order to increase its usage in the Ruby world.

1.4 Contribution

This thesis will provide several contributions. They will be split in two parts : the contributions of the tool to be implemented and the contributions of this thesis. In view of better assessing whether we achieved our objectives or not we will give a number for easy reference later in the thesis.

1.4.1 Objectives of the tool

The tool to be created will have several objectives to fulfil.

A1 The tool should try to simplify bug detection and allow the verification of structural coding conventions. Lots of regularities are often written only as comments in the source code but never checked explicitly. We want a tool that can perform such a verification task.

A2 The tool must be easy to learn and easy to master. We don't want the user to "waste" lots of time in order to learn the tool before he can start using it. The tool must be a gain of time, easy to adopt and should have as little ambiguities as possible in its syntax (for example very similar keywords that have completely different behaviours should not appear).

A3 The tool must be easy to extend. It is impossible to handle all possible constraints in the tool. The best way to handle it, is to allow the user to create his own customized constraints specific to his application. This means that the syntax must be generic enough to express as much constraints as possible. The tool must have a clear way to add them without the user reading lots of documentation.

A4 The tool must impose as little constraints as possible on the source code. It must not induce some failures of the source code when the tool overrides classes/methods/variables used in the code we want to verify.

A5 The tool must infer on high level. It must understand class inheritance, module inclusion, dynamic modification, ... We want a tool that understands the semantics and structure of the code. For instance, if a new child is created for a class, we don't want to change the contract. The tool must apply the uContracts on the new child automatically without requiring a programmer intervention.

A6 The tool must check the declared constraints immediately and as fast as possible, meaning amongst other, that constraints have to be executed quickly.

1.4.2 Thesis goals

This thesis will aim to answer several questions :

B1 Is the concept of usage contract generic enough for object-oriented languages other than Smalltalk ?

B2 Does the concept of usage contract fit the practices of the Ruby community ?

B3 Is the tool useful in practice for professional Ruby programmers ?

B4 What are the contract types from the Smalltalk prototype generic enough to be applied on other object-oriented languages ?

B5 What are the strengths and limitations of such a tool, particularly for the Ruby language ?

1.5 Roadmap

This thesis will be separated in several parts, following a logical path. We start from the current development environment and then tackle the tool.

First, we will look at the already existing tools in the Ruby world taking into account several aspects such as metrics, conventions, testing, ect. We will analyse them, discuss about their strengths and weaknesses, see what has already been done by these tools and if we have to reimplement them according to our objectives or define what can be reused. We will also look if some of these tools fit our requirements and objectives.

Next, we will describe the tool that will be created. We will look at its requirements, and what we want to achieve with this tool. We also discuss some of Ruby's features, what are the ideas we can use and for which purposes. We will also see how the tool is implemented in

practice, which are the mechanisms and how to use it.

We will then try to validate the tool by conducting several tests. To perform this, a first test of the tool on itself will be prepared. We will try to apply the tool on a concrete example to evaluate its real usage, and discuss about its possible usages in the real world.

Afterwards, we will discuss about the tool in general. The questions will be "what are its code qualities" and "what are its strengths and weaknesses". We will try to be as objective as possible in order to assess the tool quality.

We will also have a small discussion on the possible improvements of the tool. What can be added to the tool to increase its usefulness or quality ? Some might not be feasible or be extremely complicated to implement, but it is always interesting to think about new ideas and push the tool as far as possible.

We will finish with a conclusion on the tool and check if it achieved the objectives and contributions set at the beginning of the thesis. If some were not met, we will put forward why they could not be achieved and how the requirements could be changed in order to be met.

Chapter 2

Related work

In this chapter, we will explore the current state of the Ruby world. We will analyse some tools and solutions available which allow us to look at the source code and deduce characteristics on it such as metrics, black box testing, other types of tools looking at the source code. We will try to see if any of these tools can be used to apply the notion of usage contracts, but also what kind of checks are already done, what has not been done and what the usage contracts can offer. First, we will talk about unit tests in general and in Ruby. Then, we will talk about the solution described for SmallTalk[1] before looking at the tools developed in Ruby such as Rubocop. Finally, we will conclude and see what is missing from the current set of Ruby tools and what must be added to meet our requirements. Here below you can find a small overview of the tools that will be discussed :

Types	Description	Tools in Ruby	Section
Unit tests	Black box principle (Input-Output testing)	MiniTest, RSpec, ...	2.1
Metrics	Quality of the code	Rubocop, Reek, ...	2.3.1, 2.3.2
Convention	Code conventions	ruby-lint	2.3.3
Contracts	constraint on the code	contract.ruby	2.3.4

2.1 Unit Testing

Unit tests are a set of test cases. Each test case must be independent from the other ones, and be related to a single small unit of test functionality. A test case is typically a simple input-output verification. For example, an 'add' function with the parameters 2 and 5 must return 7. If the result of the method does not match with the expected result, the unit test fails.

Unit tests are intuitive and allow a programmer to quickly test each situation a function/-class/module may encounter once it is used in a program. However, the programmer must write every unit test implying that if a new class is written for example, the programme must write every unit test for that new class even if it is the same as its parents. We cannot easily reuse

these tests or automatically apply them on other entities without explicitly writing these tests again. Besides, if the programmer omits a test case, he might not detect a bug in his code when he executes his unit tests.

In Ruby we can use, for example, the *MiniTest* gem for unit testing. In order to use it, simply install the MiniTest gem, then create a new class inheriting the class MiniTest::Test and create methods beginning with **test** for the test cases wanted. Once executed, we can look at the output in the console and watch which test cases succeeded/failed. Here below you can find a little example :

```
1  class TestAdd < MiniTest::Test
2    def test_integer
3      assert_equal(7, Operation.add(2, 5))
4    end
5    def test_negative_integer
6      assert_equal(-3, Operation.add(2, -5))
7    end
8    def test_fail # Will fails
9      assert_equal(0, Operation(1, 0)
10   end
11 end
```

These unit tests will generate the following output :

```
1  Run options: --seed 31496
2
3  # Running:
4
5  F..
6
7  Finished in 0.183010s, 16.3925 runs/s, 16.3925 assertions/s.
8
9    1) Failure:
10 TestAdd#test_fail [D:/Ruby Workspace/uContract2/op.rb:18]:
11 0.
12 Expected: 0
13   Actual: 1
14
15 3 runs, 3 assertions, 1 failures, 0 errors, 0 skips
```

We can easily notice that the test that was expected to fail is written in the output with the method which is inappropriate, the received output confirms this.

We can also mention **RSpec**[5] which is a black box verification tool, a *Behaviour Driven*[6] approach.

2.2 Usage contracts in SmallTalk

As previously explained, usage contracts were first implemented in a SmallTalk prototype. Usage contracts (uContracts) describe regularities such as API usage protocols, coding idioms and naming conventions. Usage contracts, which aim to document implicit regularities which sometimes might not be checked, are meant to be explicitly checked. uContracts define structural regularities on liable entities. Liable entities are the target of the usage contracts, they can be either classes, methods or variables.

The idea of usage contract was partially inspired on a tool called *intensional views*[7]. An intensional view is a set of source code entities (classes or methods) which are structurally similar. Instead of enumerating the elements of the view, they are declared as a set by intension. Enumerating the elements is a declaration by extension. The tool used a logic language for declaring the set and the constraints applied on them and the tool then verified the source code by checking validity of the logic program.

The main issue with the intensional views tool however was the need for learning a new logic language in order to define the constraints. Besides, there was a cleavage, a split, between the source code, and the source code constraints and their verifications. This was one of the main reasons why the Smalltalk prototype implementation of usage contract was conceived.

In order to avoid learning a new language, a Domain Specific Language approach (or DSL), was used for the SmalltTalk prototype. An internal DSL is a host language modified in order to make it look like a different one. Thus, the internal DSL of the usage contracts prototype directly uses the syntax of Smalltalk instead of a separated logic language, avoiding the programmers to have to learn a new language for the tool's purpose. Besides, for understandability purposes, the contract declaration must be as readable as possible. This means that the domain specific language has to be designed carefully.

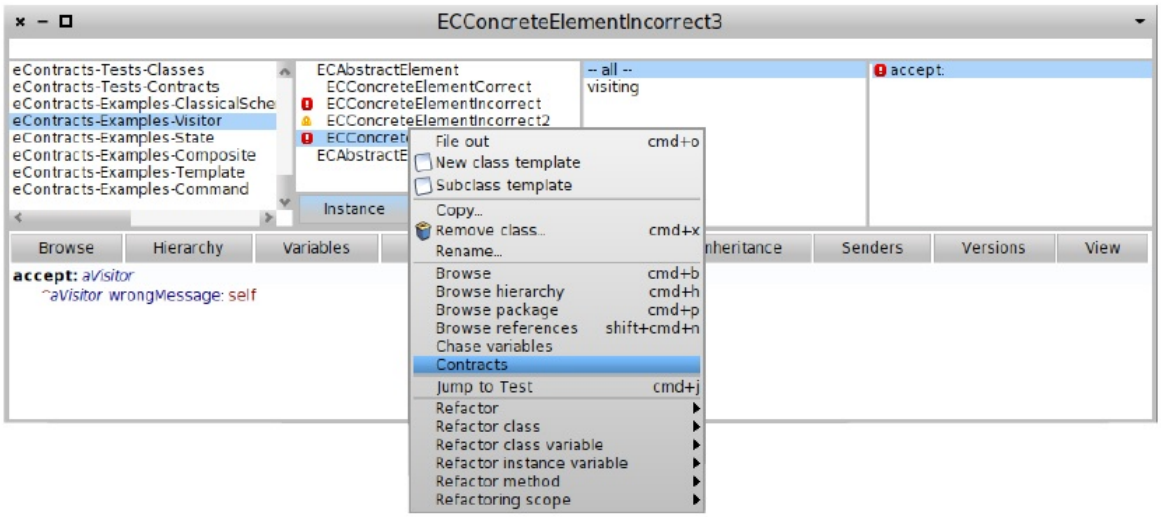
One other reason why an Internal DSL was chosen for the usage contracts tool over an External DSL, was the location of the contracts. With an internal DSL, the contracts are written as part of the source code and do not raise syntax errors. They can be interpreted/compiled

with the source code. An external DSL would force us to split contracts and targets, which can make their connection harder to understand or implement.

Here is a little example of a usage contract in the Smalltalk prototype. The usage contract is applied on the method *initializeWithDatabase* and checks if the method calls *super*, and if it is overridden it suggests that the method should be in the protocol *initialize-release* :

```
<selector:#initializeWithDatabase:>
        contract
        require:
                (condition beginsWith:(condition doesSuperSend))
        if: (condition isOverridden).
                contract suggest:
                (condition methodInProtocol:'initialize-release')
```

Here is a small visual of the Smalltalk tool where failure of only suggested contracts is indicated in yellow and failure of at least one required contract in red.



We cannot simply use the usage contract tool on Ruby code, because Ruby had structures that don't exist in Smalltalk (module for instance). Besides, the Smalltalk tool was not implemented in and for Ruby.

2.3 Other Ruby tools

Lots of tools on source code analysis already exist in the Ruby world. However, this does not mean that they fit our demands. We will look at some existing Ruby tools that can perform checks on a source code, and investigate if they can be used to exploit the usage contracts notion or if they already handle constraint checks that we want to achieve.

2.3.1 Rubocop

RuboCop is a static code analyser for Ruby. It allows a programmer to assess metrics of the code on several aspects such as variable name length, method length, ... The verification scripts are written in yml files. yml is a short name for **YAML** which is a human readable data format language and a recursive acronym for *YAML Ain't Markup Language*. YAML takes lots of idea from XML or concepts from programming languages from C or Python.

In Rubocop, the programmer must explicitly write which yml files are used for the verification of the source code. For example, the programmer could write several tests with different metrics (variable name length or instance). The programmer must give the name of the file used to know which type of checks to discard and the metrics to use.

However, Rubocop does not look at the internal method logic or the class inheritance hierarchy. For example, Rubocop cannot check if a class has a given class in its inheritance tree or that all classes overriding a given method of a given class respect some constraints such as calling the method super.

Rubocop applies its verifications on files, not on classes/methods. Besides, the set of files checked is declared by extension : we need to explicitly write which files are checked or if we give a folder, which files are not checked. This can become heavy if we want to check a large project.

Let's take a very small example of a yml file called *simple.yml* and how it is used. We apply the checks on all files of the folder Contracts except the file Contract.rb. First, we set that the variable names must be snake_case and not camelCase. Then, we define that a class can't have more than 100 lines. Finally, we disable the check on the lines length.

```
AllCops:
  Include:
    - 'Contracts/*'
  Exclude:
    - 'Contracts/Contract.rb'

Metrics/ClassLength:
  CountComments: false  # count full line comments?
  Max: 100

Style/VariableName:
  EnforcedStyle: snake_case

Metrics/LineLength:
  Enabled: false
```

In a next step, we call Rubocop in a terminal by giving the config file :

```
rubocop -c simple.yml
```

We can thus observe that Rubocop does not seem to be a good tool fitting our demands, based on the previous points.

2.3.2 Reek

Reek is a tool which looks at classes and methods and reports code smells. Just like Rubocop, Reek verifies the source code based on a configuration file given by a programmer. Reek analyses the code of the given files/folders. For example, it can report methods which have more than 4 parameters or a method call being made more than 3 times in the methods of the analysed file.

Even if Reek seems to be a little more sophisticated than Rubocop, it does not take in account class hierarchies, module inclusion, ..., meaning we cannot define a check on all children of a given class.

Let's again show a small example of a reek configuration file. We will only show some points to avoid a too long example. We will set the maximum number a method is called in a given method at '1'. Then, we define that maximum '2' methods can have the same number and names of parameters (if more, maybe an object is needed instead). We also set the maximum number of parameters at '3'. Finally, we set the maximum number of instances variable at '9' and maximum number of methods in a class at '25'.

```
---
DuplicateMethodCall:
  enabled: true
  exclude: []
  max_calls: 1
  allow_calls: []
DataClump:
  enabled: true
  exclude: []
  max_copies: 2
  min_clump_size: 2
LongParameterList:
  enabled: true
  exclude: []
  max_params: 3
  overrides:
    initialize:
      max_params: 5
TooManyInstanceVariables:
  enabled: true
  exclude: []
```

```
  max_instance_variables: 9
TooManyMethods:
  enabled: true
  exclude: []
  max_methods: 25
```

We then call reek in a terminal with the following command :

```
reek -c config.reek Contracts/*
```

Reek will then enumerate the files and for each one which checks failed. Let's notice that some checks are not always pertinent like the **DuplicateMethodCall**. In our example, we set it at one. If in a method we perform an if-else and each branch call the same method but with different parameters, the check of DuplicateMethodCall will fail because it will find two calls in the same method. However, it is impossible to have an execution where the call is performed twice. This shows that reek does not look at the method flow.

2.3.3 ruby-lint

ruby-lint is a static code analysis tool for Ruby. It mainly focuses on logic related errors, such as the use of non existing variables, instead of metrics. ruby-lint does not look at the number of line per method for instance. ruby-lint analyses a set of files given in arguments (for example *ruby-lint Person.rb Student.rb*).

The first arising question is how ruby-lint differs from the tools previously described (for example RuboCop). The main distinction with RubCop is that ruby-lint focuses on technical problems rather than style related issues (for example a community driven Ruby style guide). ruby-lint performs basic check styles but not as complex as RuboCop or Reek.

This tool helps developers to find nasty bugs (uninitialized instance variables for instance) but cannot look at hierarchies, module inclusion, S-EXP analysis, ... Plus, we cannot write a rule which automatically applies only on methods overriding a method from a given parent.

For these reasons, ruby-lint is not entirely the tool we want. It may be really useful to debug but cannot tackle all our requirements.

2.3.4 contract.ruby

A very interesting tool is contract.ruby. The name seems to match at least some of our demands, and is based on the *Design by Contract* approach. In fact, the concept of Design by Contract is central to an other programming language called *Eiffel*[4]. Design by Contract assumes that all called components must respect a precondition and a postcondition. The main goal is pre/post assurance. If the pre/post are not explicitly checked, for example only described in the comment

above a method, this approach becomes insecure. This insecurity leads lots of programmers to use the defensive approach, where the code handles all possible scenarios even the one that should not exist (for example a negative parameter because the method uses a square-root on it) in order to catch all behaviours and avoid critical errors.

contract.ruby is a tool, looking like Unit tests, where you can execute a black box check. In fact, contract.ruby explicitly checks the pre/postcondition on the input and output types. If we take the previous example in the Unit tests section, we can express the same check directly in the targeted class. In the following example, if we call Operation.add(1, "2") we get an error message in the console telling that it expected a numeric but received "2". :

```
1  require 'contracts'
2  include Contracts
3  class Operation
4    Contract Num, Num => Num
5    def self.add(num_1, num_2)
6      return num_1 + num_2
7    end
8  end
9
10 puts Operation.add(1, 2)
11 puts Operation.add(1, "2")
```

This piece of code gives us the following output. The first line is the result of the line *puts Operation.add(1, 2)*. Then, we observe the result of the violation done by the second *puts* :

```
3
Contract violation for argument 2 of 2: (ParamContractError)
        Expected: Num,
        Actual: "2"
        Value guarded in: Operation::add
        With Contract: Num, Num => Num
        At: D:/Ruby Workspace/uContract/contractruby.rb:6
        (...)
```

Even if this tool is quite powerful and useful, it does not tackle all cases we want like hierarchy check, module inclusion check, method structure, ... Since this tool is already developed and seems to work well, we could exclude the cases handled in the contract.ruby tool from the tool we might need to develop.

2.4 What's missing

We found several existing tools which allow us to check metrics on Ruby code in order to improve its quality. We also found tools to declare black box tests and evaluate test cases, or dynamically report wrong usage of functions.

However, we don't have a tool that allows us to declare constraints on the code structure, classes, their hierarchy or their methods which will be automatically checked. We cannot, for example, automatically check that all children of the class **Person** call *super* in their *initialize* method.

Besides, these tools are not easily scalable because we must declare our tests for every function in **contract.ruby** or write the unit tests for every new class. A good solution would be a way to declare contracts once and they will be automatically applied on the wanted targets (children of a class, all methods in a given class, ...).

For all these reasons, we need a new tool which can tackle our requirements and problems. The tool must meet all the requirements and have an overall quality at least equals to already existing tools.

Chapter 3

Running example

3.1 Implicit constraints

As a small example of what kind of structural regularities the tool could express, let's consider the example of a simple drawing framework, with a class Figure from which every other figure such as Rectangle or NSide inherit. Naturally, we have some constraints that must be respected by the classes inheriting the class Figure. However, these constraints are not easy to express explicitly in the code :

1. The children must call *super* in their *initialize* method
2. The children must override the *draw* method
3. The *draw* method of the children must call *refresh*
4. If the method *undo* is implemented by a child, it must also implement *redo*

3.2 Contracts declaration

Fortunately, those kind of constraints can be easily and concisely expressed. We declare the corresponding usage contracts in the Figure class definition here below :

```
1 class Figure
2
3   # Children of this class must call super in their initialize method
4   contract.requires(:methodsOverriding, :initialize).call( :super )
5
6   def initialize(canvas, cood)
7     @canvas = canvas
8     @coord= coord
9   end
```

```
10
11    def refresh
12      @canvas.refresh
13    end
14
15    # The children must override the draw method
16    contract.requires(:subclasses).override(:draw)
17
18    # The draw method of the children must call refresh
19    contract.requires(:methodsOverriding, :draw).call(:refresh)
20
21    def draw
22      ... some default behaviour of the draw method ...
23    end
24
25    # If the method undo is implemented by a child, it must also
        implement redo
26    contract.requires(:methodsOverriding,
        :draw).has_conditional_contracts do
27          implement(undo)
28          implement(redo)
29    end
30
31  end
```

3.3 Usage

Even if these rules seem quite simple and easy to remember, it is also easy to forget one of them. Therefore the programmer could lose lots of time before noticing the error. However, with the usage contracts, he will get a warning which explains that the usage contract is not respected. Also, a new user will easily notice these mistakes and learn the usage of the framework quicker than without the usage contracts.

Chapter 4

Tool description

This section will describe the tool we aim to build. We will first analyse its requirements, what we want to obtain with our tool, what kind of usage contracts we want. Then, we will look at Ruby's structure, mechanisms and limitations, and what can be used for building our tool. Finally, we will describe the created tool's implementation and limitations before a small conclusion on the tool created.

4.1 Requirements analysis

The tool will be written in Ruby for the Ruby programmer's ease. We don't want the programmer to learn a new language in order to use the tool. That's why the tool will use an internal DSL like the Smalltalk prototype. It also means that the programmer can put our usage contracts declarations in the Ruby source code and it will be executed alongside the source code without any syntactical error.

The focus for the constraints, that can be declared by the tool, was on the static ones. Static constraints only look at the syntax expression tree but not at the variables value during run time for instance. A key aspect was efficiency, meaning that the contracts' verification must be executed as fast as possible (immediate feedback, cfr. contribution *A6*). In order to perform such checks, the code is interpreted by the Ruby interpreter and the contracts are executed when interpreting the code. Dynamic checks were not considered because contract.ruby already tackles some of these constraints and these constraints are more complex to implement but also heavier to compute.

Our tool must be able to specify two types of usage contracts : **requires** and **suggests**. **requires** means that the contract must pass, or it would certainly create bugs or unwanted behaviours. **suggests** defines a warning, for example a variable name which doesn't respect a convention but will not raise error (for readability purpose) and must not necessarily pass.

In order to check a method's structure, we need to analyse its source code. Unfortunately, no function exists for retrieving the source code of a method or a class. We need a program to get the code of a method from the file it belongs to. To gain time in the tool development, writing our own parser for retrieving a method's code is dropped because well written and complete gems already exist. With such gems, the code retrieved can be transformed into a S-exp tree before being analysed.

The tool must be easy to extend. For example, any programmer should be able to easily add new types of usage contracts to fit his needs. He can also modify the code of any already existing contract if needed. New usage contract types must be reusable if used multiple times and if only used once, fast to write. The tool must also be as generic as possible to tackle as many application areas as possible.

The tool output must be as readable and understandable as possible. The programmer must instantaneously see the results of his declared contracts in order to manipulate them if wanted. The programmer must be able to reduce the list of visualized contracts and sort them for readability. The tool must also allow the programmer to store the results in a file, so that they are easy to share with other programmers of a team or a project leader.

4.2 Proposed solution

First, let's look at the alternatives considered for the tool implementation and see which alternatives were retained or discarded and why.

4.2.1 Easy to add in a project

The tool must be easy to use on source code. This means that we don't want to force the programmer to change its class hierarchy to add the tool in its project for instance. We discard the option of inheriting a class to use the tool. We could use a module inclusion but it could also raise errors when a programmer forgets to include it. The easiest way could be to simply require the tool with the **require** or **require_relative** keywords.

Besides, only one require should be needed. The tool requires itself all the components it needs for the computations and features.

4.2.2 Basic usage contracts

The tool should contain some predefined contracts for the most used cases. These usage contract types were inspired by the Smalltalk prototype for most of them and some were deduced from Ruby mechanisms not present in Smalltalk such as Modules. It seems quite obvious that the tool cannot handle all possible constraints that can be applied on source code. Here is a list of the kept usage contract types; more information on their assumptions and signatures can be found in the appendix :

- A class or method must assign a variable.
- A method must begin with a given call.
- A bidirectional conditional contract : if the first contract passes, the second must passes and vice versa. The contract also passes if both contracts fail.
- A method must make a given call
- A conditional contract, if the first contract passes, the second must pass but if the first fails, the result of the second is not relevant and the conditional contract passes.
- A contract that allows block as verification code for custom contracts.
- A contract that checks if a given table exists in the database.
- A disjunctional conditional contract, one or both of the contract must pass.
- A method must end with a given call.
- A class must extend a given class.
- A method must have an explicit return statement.
- A class must implement a given method.
- A class must include a given module.
- A given method must be private.
- A given method must be public.
- A method cannot have more than x instructions.
- A class must override a given method received from a parent.
- A method must begin with a given static call.

The programmer is free to create new ones to fit his own needs. The tool contains basic checks but can be extended to expand the list of handled constraints.

4.2.3 UContract class

In the tool, there is a class called **UContract** which defines an usage contract instance. Each contract declared in the source code corresponds to an instance of UContract. The instance contains every information about a contract verification as instance variables. For the reader ease, the variables will be illustrated with the running example. We can first think about the information we could need to define an usage contract :

- Is the contract a violation which can lead to an error (requires) or a simple warning (suggests).
- What is the method targeted or a special value to declare that the contract targets a class.
- What is the class/module targeted.
- Is the contract applied on the class where it is declared or its children only or all its hierarchy including the class where the contract is declared.
- The result of the contract.
- When was performed the last check of the contract.
- What are the declared exceptions for the contract (methods or classes not targeted by the contract. For instance all children of the class Figure excepted the class DummyFigure).
- What is the type of contract (calls a method, implements a method, ...).
- What are the arguments of the contract.
- What is the block given to the contract.

All these information are detailed later in the text.

The programmer will create a new blank UContract instance and set the information with the DSL methods. Since there are lots of possible contract types, the UContract class definition will not contain the verification code for each type of contract but rather retrieves the code from an other class. Each contract type has its own class where the code of its verification is put, the UContract instance simply chooses the one needed according to the used keyword.

For instance, the first keyword creates an instance of UContract, the second set its type to requires (leads to an error if not respected) and its target (here the class where the contract is declared) and the last keyword is the type of the contract with its arguments (here the class must implement the method *draw*). The verification of the contract type *implements* is in the class **ImplementContract** and the instance of the UContract simply calls *apply* on the class **ImplementContract** and retrieves the result of the verification :

```
1 contract.requires(:class).implements(:draw)
```

4.2.4 Override Module and Class

As previously mentioned, the tool will define an internal DSL. Since in Ruby **Module** is the parent of class **Class**, it is also a parent of all classes. The quickest way to add DSL methods to all classes (methods understood by the classes) is to add in the Module definition the wanted DSL methods and mechanisms. Thus, the DSL methods are class methods understood by the classes.

Several behaviours must be understood by the classes. **Module** must handle the adding of a new method to its definition, when it is included by a class or a module and the DSL methods. The class **Class** (and by extension all classes of the targeted project) will inherit all these mechanisms but must also implement the mechanism when a class is inherited since a Module cannot be inherited.

4.2.5 Use of reflection

In order to be as transparent as possible, the tool will use reflection with hook methods. Ruby has a quite long list of hook methods[2] :

Method-related hook methods

method_missing	called when an unknown method call is performed on a class
method_added	called when a new method is added in a class definition
singleton_method_added	called when a new class method is added in a class definition
method_removed	called when a method is removed from a class definition
singleton_method_removed	called when a class method is removed from a class definition
method_undefined	*not documented*

Class and Module related hook methods

inherited	called when a class is inherited by another class
included	called when a module is included by another class/module
extended	called when a class is extended by another class
extend_object	extends the specified object by adding this module's constants and methods (which are added as singleton methods)
initialize_copy	performs the initialization when an object is copied
append_features	when a module is included in another one, Ruby calls append_features in the included module, passing it the receiving module
const_missing	called when a reference is made to an undefined constant

We can imagine some usage of these methods :

- each time a method is added, we can check with *method_added* that it passes the usage contracts applied to it. The same goes for *singleton_method_added* in the case of class

method.

- each time a class is inherited, the *inherited* method allows us to copy the usage contracts applying to the children of the class inherited. The same goes to *included* for the modules inclusion.

- for readability, usage contracts' verification code (code of **assign** contract, **call** contract, ...) could be in other classes (and in the same folder) and not all in a single class/file. We could use method_missing (method called when a method which is not understood by a class is called) on the UContract instance to dispatch the method calls and apply the right verification method.

```
1 contract.requires(:class).implements(:draw)
```

Here the UContract instance created by *contract* will not understand *implements*. In the method_missing method, it will look at the keywords defined for the predefined usage contracts, find a correspondence for *implements* and call the method *apply* for the class corresponding to the keyword implements.

Their actual usage is explained later and more precisely so the reader doesn't need to understand them now.

It seems quite obvious that the source code verified may no longer run if it uses the hook methods listed above. For example, if the *method_added* is overridden by the class we want to check, some contracts will no longer be checked. The programmer must be careful when using the tool.

Since we want a well written tool, we will use gems previously (such as Reek, Rubocop, ...) cited to access metrics. Metrics will be used to detect code smells, too short or long names, classes and method length, ... These checks will help us to increase the quality of the code.

4.3 Implementation

4.3.1 Proposed syntax

As previously explained, we want an internal DSL. We must think about an easy to understand and easy to use syntax for the DSL used by the tool. We chose a DSL which is close to a contract declaration in the format of a sentence. The following sentence is an example of a contract declaration : *the contract requires that the method draw calls the method refresh.*

The first thing to do is to extract the important words from the sentence above : *contract*, *requires*, *method*, *draw*, *calls*, *refresh*. The DSL used by the tool is really close to the previous sequence of words. Let's first take some examples :

```
1 contract.requires(:method, :draw).calls(:refresh)
2 contract.requires(:allMethods).max_length(20).exception(:methods,
     :called, :initialize)
3 contract.requires(:class).implements(:refresh)
4 contract.requires(:subclasses).override(:refresh).exception(:methods,
     :nameBeginWith, :Test)
```

Let's explain the flow of execution of the previous contract declarations. First, we create a new blank contract by calling *contract*. Then we set the type of the contract to :violation by calling *requires* on the blank contract, and set the liable entity. *requires* needs 1 or 2 arguments depending on the first argument given. if the first argument is :method, :methodsOverriding, or :methodAndOverrider, the second argument is the method targeted. The other keywords don't need a second argument. Here is the list of the possible keywords :

Keyword	Meaning
:method	the method with the name given by the second parameter
:methodsOverriding	the methods overriding the method with the name by the second parameter
:methodAndOverriders	the methods overriding the method and the method itself with the name given by the second parameter
:allMethods	all the methods of the current class
:allSubclassesMethods	all the methods of the children class
:allHierarchyMethods	all the methods of the current class and its children
:class	the class where the contract is declared
:subclasses	the class' children where the contract is declared
:hierarchy	the class and its children
:directSubclasses	the class' direct children where the contract is declared but not the children's children

Finally, we can set an exception to the contract verification.

- The *exception* method takes as first argument the target : **:methods | :classes**.
- The second argument is a rule : **:nameBeginWith | :nameEndWith | :called | :nameContainWord | :nameDoNotContainWord | :andChildrenOf**.
- The third argument is a pattern, be careful that the pattern matching is case sensitive and

applies on the name of the method|class.

For example, here below a contract which requires that all children of the class implement *openDB* except classes with a name which starts with 'Test' because these are unit tests looks :

```
1 contract.requires(:subclasses).implements(:openDB).exception(:classes,
    :nameBeginWith, :Test)
```

4.3.2 UContract class

We will now discuss about the implementation of the UContract class. First, let's describe the instance variables along with the value considered and their meaning. We will use the contracts from the running example to illustrate their meaning.

- @type : Type of the contract (:violation, :suggestion) with :violation as default.
  the keyword *requires* in the contract declaration means that the contract is a violation contract. *suggests* can replace *requires* in the contract declaration to define a suggestion.

```
1 contract.suggests(:allMethods).calls(:super)
```

- @method_name : Method targeted by the contract. There are some special values :
  - *nil* as default. (the target has not been set correctly, lack of keyword *requires* or *suggests* for instance)
  - "." if the contract applies to all methods of a class.
  - "@" if the contract is not applied on a method but on the class itself.

  The three values above were chosen because they cannot be used in a running code, a syntax error will arise if used. No methods can be named **nil**, **.** or **@** in a running Ruby code.
  Let's now take 3 examples to illustrate the possible values :

```
1 # @method_name = :initialize
2 contract.requires(:methodsOverriding, :initialize).call( :super )
3 # @method_name = "@"
4 contract.requires(:subclasses).override(:draw)
5 # @method_name = "."
6 contract.requires(:allMethods).max\_length(20)
```

- @class_name : Name of the class targeted.

- @target : Target of the contract. Here, there are four possible values :
  - :self for the class itself

```
1 contract.requires(:allMethods).max\_length(20)
```

  - :children for only subclasses

```
1 contract.requires(:methodsOverriding,
    :initialize).call(:super)
```

  - :hierarchy for itself and subclasses

```
1 contract.requires(:methodAndOverriders,
    :initialize).call(:super)
```

  - :directSub for direct subclasses only

```
1 contract.requires(:directSubclasses).max\_length(20)
```

  this means that the keywords
  - **:methods**, **:allMethods** and **:class** will set the target to :self.
  - **:methodsOverridin**g, **:allSubclassesMethods** and **:subclasses** will set the target to :children.
  - **:methodAndOverriders**, **:allHierarchyMethods** and **:hierarchy** will set the target to :hierarchy.
  - **:directSubclasses** will set the target to :directSub
- @result : Result of the contract verification (:pass, :fail) with :fail as default. It is the real result, not the expected one for the contract.
- @timestamp : Time when the last verification was applied with nil as default. Since the contract might be applied several times, we only remember the last one.
- @exceptions : List of methods|class for which the contract doesn't apply.
- @contract : Name of the contract applied (contract type such as :call, :override, ...) with nil as default.
- @args : parameters of the contract's verification as an array.
- @block : block passed to the contract.

4.3.3 UContract mechanism

As explained before, the *contract* DSL method creates a new blank UContract instance, *requires* and *suggests* set the type of usage contract and the target of the contract. The last call is the type of the usage contract to apply. Since the UContract class definition cannot contain the verification code for all the contract types, each type has a class where the code is put (for example, the call *assign* is linked to the class **AssignContract**). All these contract classes are put in the **Contracts** folder. Each class has an instance variable containing an array where we put the name of the call (**AssignContract** can be executed with the keywords *assign* and *assigns*).

When the tool is required, the UContract class will load all the classes contained in the Contracts folder and link the calls with the corresponding class where there is the verification code. With the *method_missing* method, the UContract instance will catch the call and execute the method for the corresponding class. The creation and requirements for new contract type classes are explained later.

To be complete, an usage contract can execute its check several times during the contracts checks (for example a class method which checks that a given method performs a given call). Since the checks are really fast, it doesn't impact the performance. The contracts of a class are all computed one last time with the *at_exit* method. The method *at_exit* is called at the end of the class declaration. It can be turned off but some contracts might not be executed.

Since the contracts can be declared anywhere in the code, contract might be declared before the targeted method is added to the class definition. When a new contract is declared in the code, the tool adds it in the array of usage contracts declared for the class and/or its children depending of the target. Then, the tool checks if the method is known by the class. If not, it waits until a new method is added to the class declaration and checks for all declared contracts if the method is targeted and if so, executes it.

Let's now look on how the classes of usage contract types are declared and how to add our own usage contract types.

4.3.4 Creating a new contract type

There are two ways of declaring a new type of Contract. The first subsection describes Contract types which will be used several times in the source code, while the second subsection will describe an other way of declaring new contracts, mostly used for one-time contract types.

Add a new child to Contract

In the folder *Contracts*, we can find the declared contract types. Each class describing a type must inherit from **Contract**, but we can have classes between if we want to create subclasses for a type. There are several rules to be respected if you want a contract to work :

1. put a value for the *@call_names* as an array which defines the names that will match the class for the verification (for example, the class CallContract has "call" and "calls" in its @call_names field)
2. override the *apply* method that takes two arguments, an array containing all arguments given when the method is called and a block, and returns true or false according to the verification result
3. the class must be a child of Contract
4. the class must be put in the Contracts folder

The first rule is mandatory, without it you cannot call your contract in a class. The second rule contains the code of the verification. The third is mandatory in order to inherit needed piece of code and the fourth if you want your contract to be added in the list of contract types known by the UContract instances.

Custom contract

The second way of declaring a new contract type is to create a custom contract. It's a contract with a custom block as verification to apply. Here is a syntactic example :

```
1 contract.requires(:method, :draw).respects_custom_contract $arg_0$,
    ..., $arg_k$ do
2       |args, ucontract|
3       ...
4 end
```

Note that this is the correct syntax without parenthesis and comma between last argument and the block. |args, ucontract| are the arguments passed to the custom method. They follow the following pattern : **args** is the arguments arg_0, ..., arg_k passed to the block as an array and **ucontract** is the UContract instance. Here is a concrete example where we check if the class name is in camelCase and not snake_case :

```
1 contract.requires(:hierarchy).respects_custom_contract do
2     |args, ucontract|
3     next !ucontract.class_name.include?("_")
4 end
```

Notice that the return in a block is the instruction **next**.

4.3.5 Contracts visualisation

Once we execute the contracts, we want to see the results. There are two possibles ways, depending on what the programmer wants.

First, the programmer can retrieve the list of contracts applied on a class. There are several calls :

```
1 AClass.contracts
2 AClass.children_contracts
3 AClass.get_contracts
4 AClass.show_contracts
```

- The first line will return an array containing the contracts applied on the class **AClass** and its methods.
- The second will return an array containing the contracts applied on its children.
- The third line will return an array containing the contracts applied on the class **AClass** and its methods but remove the contracts that are not relevant (For example a contract which is not checked because the targeted method is not implemented by the class).
- The last line prints the relevant contracts in the terminal.

The other way of visualizing the contract results is the **UContractHTML** class. Here is an example of an instruction added at the end of the class definition.

```
1 require_remative './UContract'
2 class AClass
3         ...
4 end
5 UContractHTML.show_contracts(classes: [:AClass], contracts: :all,
    results: :pass, sortBy: :method)
```

Let's explain the parameters :

- classes(default :all) : array of classes to display.
- contracts(default :all) : contract type to print (AssignContract, ...).
- results(default :all) : result to show (:pass or :fail).
- sortBy(default :class) : column used to sort (possibilities : **:class | :method | :contract | :type | :result**).

The tool will then open a new HTML page on the default browser and display the results in a table. The columns are the following : the name of class, the type (violation or suggestion) of the contract, the name of the method targeted or **-** if the contract targets the class, the type of the usage contract (call, implement, ...), and the result of the check.

Here is an example of the html produced by the HTML tool. As the tool is open source, the user can easily change the CSS if he wants :

Result at 2015-05-30 14:50:31 +0200

Number of contracts : 4

Contract	Type	Class	Method	Result
implements	violation	Person	-	pass
calls	violation	Person	greet	pass
assigns	violation	Person	initialize	fail
implements	violation	Person	-	fail

The html file contains a JavaScript code[12] which allows the programmer to sort the column if he wants to change it without changing the instruction in the code and regenerating the html file.

The HTML is produced with **ERB** which is a feature of Ruby. It enables to generate easily the html from templates. It is within the Ruby core, so the user doesn't need to install new gems.

We can also note that the html file produced can be transformed into a pdf with the following instruction :

```
1 require_rematives './UContract'
2 class AClass
3         ...
4 end
5 UContractPDF.show_contracts(classes: [:AClass], contracts: :all,
    results: :pass, sortBy: :method)
```

The parameters are the same as the HTML feature. The tool will also automatically open the file.

In order to produce such a document, we use the PDFKit[10] gem which transforms a html file into a pdf file. We simply generate the html file as above but instead of opening the browser, we create a pdf from the produced file. Let's note that the gem CodeRay[11] can be used to generate html file from a ruby file. The user could be interested in the CodeRay tool if he wants to print all the file from his project by combining CodeRay and PDFKit. The code combining both used during the tool implementation can be found in the appendix.

The files generated are put in the **result** folder at the root of the tool. Be careful that only the *result.html* and *result.pdf* are generated. If the other files are removed, the html file will lack CSS formatting, the template of the html file and JavaScript for column sorting. Do not remove them if you want to be able to use these features.

One important thing to take into account is the environment where the instruction is performed. The UContractHTML (and by extension the UContractPDF) class retrieves the usage contracts in the current environment. More specifically, the usage contracts must be interpreted if we want to visualize them. For instance, if we add the *UContractHTML.show_contracts* instruction at the end of a class, we will only see the contracts of the class, its upper hierarchy and the files in the require. If the programmer wants to see all the contracts, he must require all the files of his project with a code similar to the one below :

```
1 Dir['./Contracts/*.rb'].each {
2         |file|
3         require file
4 }
```

The line above will require all the files in the Contracts folder. A programmer can easily change the code to meet his needs.

4.3.6 Gems used

In order to gain time during development and avoid redundant work from others applications such as code parsing, we decided to use the following gems available on Internet to perform tasks not related to the UContract concepts and mechanics for our analysis.

- method_source : Gem used to retrieve the source code of a method. Since Ruby core doesn't allow it yet and the source code is mandatory for the verification (construction of S-EXP with Ripper), we decided to use this gem in our implementation. As we want our application to be used easily, the gem code is put in the application. The user does not need to install it, the code is already present in the tool.
  Found at `https://github.com/banister/method_source`

In order to assess the quality of our code and improve it as far as possible, we used three very useful gems :

- Reek : Gem used for static code analysis, helping us to detect bad names for variables, duplication code or too long methods.
  Found at `https://github.com/troessner/reek`

- Simplecov : Gem used for the test coverage. The main focus was to be sure all tests were done but also to detect unnecessary/forgotten code in our implementation.
  Found at `https://github.com/colszowka/simplecov`

- Minitest : Gem used for the unit tests related to the contracts' verification.
  Found at `https://github.com/seattlerb/minitest`

4.4 License

Since this subject is bound to research but also aims to increase the set of tools used by the Ruby programmers, the tool is under the **MIT license**. Here is the full statement of this license :

```
The MIT License (MIT)

Copyright (c) 2015 Sceraert Vincent

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions: \\

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
THE SOFTWARE.
```

Basically, this means that the tool is completely free, can be modified and used in other projects without any problems. However, the tool is used without any guaranties, and cannot be blamed for problems in programs that used it.

Chapter 5

Validation

Now that the tool has been explained, we want to assess its quality and correctness. We will first test the tool on small examples and check if the results are the ones that are expected. We used unit tests to create such examples and tried to cover as many possible cases as possible to find out if the tool reacts the way we want. Then, we will try use the tool on itself because we know the constraints that are hidden in it. Therefore it allows us to see if the tool itself respects its own constraints. We will also test the tool on a example from the Rails convention site and see if we can declare the conventions with our tools. At the end, we will consider a possible usage of the tool as some rules that can be added in the ActiveRecord class such all children class have a name which correspond to a table in the database.

5.1 Unit Testing

In order to validate a maximum of cases in each contract, we create a unit test case for each declared contract and verify the result. All these tests are put in the Units folder. In order to be easily visualized, we create a file named **TestSuite.rb** where we load all the files in the folder Units and also open a new page to print the coverage of the Tests. This enable us to and see if we missed some test cases or if some portions of code are unused.

In the last version of the tool, unit tests have been performed and all of them passed. Each test corresponds to a possible scenario for a contract type.

Here below is the result of the unit tests on the version of the 30th May 2015 :

```
56 runs, 102 assertions, 0 failures, 0 errors, 0 skips
```

5.2 Applying the tool on itself

In order to assess the quality and validity of our uContract tool, we decided to apply the tool on itself. However, it it impossible to directly write the contracts in the Contract classes declaration. This is due to the fact that when the first class in the Contract folder is interpreted, it does not know the other declared contracts because they are not yet interpreted. We cannot use them because the verification code of the contracts is not interpreted. To avoid this problem, we created a separate class, called **TestTool**, which requires all relevant classes of the tool and applies the contracts on them. Since they are first all required, we can add contracts to them after the require phase and analyse the added contracts.

In order to simulate a real usage of our tool, we will mix already predefined usage contracts and custom Contracts for the ones that are specific for our tool. These contracts can help the users of the tool to check if their modifications of the tool (new contracts, ...) are correct and meet the tool's internal requirements.

We executed our tests on version 0.1 of our usage contract tool pushed on BitBucket the 26th November 2014.

5.2.1 Contracts added

Contract classes

All classes declared in the Contracts folder must pass the following usage contracts :

- The file's name is the same as the class' name it declares. This must be satisfied because the UContract class considers that the name matches when it creates the list of predefined usage contract types. This type of contract was not taken into account when the initial list of predefined contracts was created. Since it is used only once in the verifications and we do not want to find new keywords for this type of contract, custom contracts perfectly fit our need.

```
1 TestTool.contract.requires(:class).respects_custom_contract do
2         inside = false
3         ObjectSpace.each_object(Class) {|cl| inside = true if
              name == cl.to_s }
4         if inside
5                 contracts_name.push(name) if
                      !contracts_name.include?(name)
6         else
7                 puts "#{name.to_s} class not found in file
                      #{name.to_s}.rb".red
```

```
8         end
9         next inside
10 end
```

- The class, with the same name as the file which declares it, is directly or indirectly a child of Contract. Again, this contract type is not likely to be used in practice because it seems odd to add such a contract in a class. This is why we used a custom contract.

```
1 class_inst.contract.requires(:class).respects_custom_contract do
2         parent = class_inst.superclass
3         while parent != Contract and parent != BasicObject
4                 parent = parent.superclass
5         end
6         puts "#{c_name} is not a child of Contract class".red if
             parent != Contract and class_inst != Contract
7         next (parent == Contract or class_inst == Contract)
8 end
```

- If the **apply** method is implemented, the contract must be callable so @call_name must be assigned.

```
1 class_inst.contract.requires(:class).has_conditional_contracts do
2         implements(:apply)
3         assigns(:@call_names)
4 end
```

- If the apply method is implemented, the method must have a return.

```
1 class_inst.contract.requires(:class).has_conditional_contracts do
2         implements(:apply)
3         contract.requires(:method, :apply).calls(:return)
4 end
```

- Two contracts can't use the same name to be called. Since these names are used to define which type of contract must be executed, we cannot have two types with the same name or one of the contract will not be executed and the user won't understand why the contract always fails.

```
1 class_inst.contract.requires(:class).respects_custom_contract
     contracts_calls do
2         |args, ucontract|
3         c_calls = args[0]
```

```
4         result = true
5         if class_inst.call_names != nil
6                 class_inst.call_names.each{
7                         |command_name|
8                         if c_calls.keys.include?(
                             command_name.to_s) and
                             c_calls[command_name] !=
                             ucontract.class_name
9                                 puts command_name.to_s + " is
                                     already used by contract " +
                                     c_calls[command_name]
10                                result = false
11                        else
12                                c_calls[command_name.to_s] =
                                     c_name.to_s
13                        end
14                }
15                next result
16        end
17        next true
18 end
```

InstructionUtils classes

Some classes declared in the InstructionUtils folder must pass certain usage contracts :

- The same constraint regarding name of file-class is applied like in previous point.
- The class must implement the method check if its name ends with Checker and is not the InstructionChecker module.

```
1 class_inst.contract.requires(:class).implements(:check) if
     c_name.end_with?("Checker") and c_name != "InstructionChecker"
```

- The class must implement the method get_instructions or include the module InstructionSeeker if its name ends with Seeker.

```
1 if c_name.end_with?("Seeker") and c_name != "InstructionSeeker"
2         class_inst.contract.requires(:class)
             .has_disjunctional_contracts do
3                 implements(:get_instructions)
4                 includes_module(:InstructionSeeker)
```

```
5           end
6 end
```

Result

We decided to split the contracts in two groups : the ones which pass and the ones which fail. For each group, we first output the number of contracts in the group then enumerate, only for the failed ones, each contract in order to control the quality of our verification. We decided not to output in this report the ones which passed to avoid a too long document. This test was quite interesting : the test on the tool returned errors. However, the same contract failed for each class of the Contracts folder, and was the following :

```
1 class_inst.contract.requires(:class).has_conditional_contracts do
2         implements(:apply)
3         assigns(:@call_names)
4 end
```

Basically, the contract declaration lacks some precision regarding assignment done outside a method. While this was a little oversight, the tool allowed us to notice quickly the missing condition and to fix it.

Another interesting fact is that the custom contracts were powerful enough to express several specific usage contracts on a given code. This shows that the tool is quite generic and can be used for checking very specific constraints. Also, the code of the custom contracts can be easily moved to a new class, child of Contract. The only real advantage of the custom contracts is that you don't have to find a new keyword to define the new contract type (such as call for CallContract, implement for ImplementContract, ...), which can be difficult after a while if lots of contract types exist within the tool.

Here below you can find a little view of the HTML output produced by TestTool :

Result at 2015-05-30 15:15:19 +0200

Number of contracts : 133

Contract	Type	Class	Method	Result
has_conditional_contracts	violation	AssignContract	-	pass
respects_custom_contract	violation	AssignContract	-	pass
respects_custom_contract	violation	AssignContract	-	pass
has_conditional_contracts	violation	AssignContract	-	pass
has_disjunctional_contracts	violation	AssignSeeker	-	pass
respects_custom_contract	violation	BeginWithContract	-	pass
has_conditional_contracts	violation	BeginWithContract	-	pass
has_conditional_contracts	violation	BeginWithContract	-	pass
respects_custom_contract	violation	BeginWithContract	-	pass
respects_custom_contract	violation	BidirectionalConditionalContract	-	pass
has_conditional_contracts	violation	BidirectionalConditionalContract	-	pass
has_conditional_contracts	violation	BidirectionalConditionalContract	-	pass
respects_custom_contract	violation	BidirectionalConditionalContract	-	pass
respects_custom_contract	violation	CallContract	-	pass
respects_custom_contract	violation	CallContract	-	pass
has_conditional_contracts	violation	CallContract	-	pass
has_conditional_contracts	violation	CallContract	-	pass
has_conditional_contracts	violation	CompositeContract	-	pass
respects_custom_contract	violation	CompositeContract	-	pass

5.3 Rails

Lots of implicit constraints appear in the Rails framework, a good example being the ActiveRecord class which needs to be overridden but lacks verification of its constraints which are most the time unknown for the programmer.

After a few searches[8], we found some rules on the children of **ActiveRecord::Base** that must be respected :

1. New active record class name exists as table in DB.
2. Class field names match table field name.
3. Foreign keys name : following the pattern singularized_table_name_id (e.g., item_id, order_id).
4. call 'save' or 'update' to save in DB.

As another validation, we will create a small database like the example from the guide of ruby on rails and try to apply our usage contracts to see if we can use them to other domain than pure structural verifications.

We first create the database which will be called **Magazine** and the tables. We will write a wrong name on purpose to assess the tool's behaviour. We will test our example with PostgreSQL[9].

Table name	Class name	Test case	Expected result
Person	Person	Correct name	pass
Article	ARTICLES	Check if case sensitive	pass (two tables with same name but different capitalized letters are unlikely to exist in practice)
Subscription	Registration	Wrong name	fail

Rule 4 above is easily checked with the *call* contract. However, the first rule cannot be checked with our basic tool. That is why we create a new contract called **MatchTableContract** and put it in the Contracts folder.
The behaviour is simple : The contract match_table takes an arbitrary number of arguments. These arguments are the name of the table we want to check. If no argument is given, the name of the class is used where the contract is applied.

We then create one class named **TableClass** which contains the contract inherited by the three classes mentioned above. Since we only check the name, the classes are empty for our example. We add some unit tests to capture the result. Here is the full code of the contracts verification :

```
1
2  require_relative "./UContract"
3  require "minitest/autorun"
4
5  class TableClass
6    contract.requires(:subclasses).matches_table
7  end
8
9  class Person < TableClass ; end
10
11 class ARTICLES < TableClass ; end
12
13 class Registration < TableClass ; end
14
15 class TestTableClassChildren < MiniTest::Test
16
17   def test_person
18     Person.get_contracts.each{
19       |contract|
20       assert_equal(:pass, contract.result)
21     }
```

```
22    end
23
24    def test_articles
25      ARTICLES.get_contracts.each{
26        |contract|
27        assert_equal(:pass, contract.result)
28      }
29    end
30
31    def test_registration
32      Registration.get_contracts.each{
33        |contract|
34        assert_equal(:pass, contract.result)
35      }
36    end
37
38  end
```

Here are the results of the unit tests :

```
  1) Failure:
TestTableClassChildren#test_registration [D:/Ruby Workspace/UContractGIT/t.rb:34]:
Expected: :pass
  Actual: :fail

3 runs, 3 assertions, 1 failures, 0 errors, 0 skips
```

The results are simple to analyse. As expected, the class Person passes the contract. Also, as expected, Registration does not pass the contract because the table Registration does not exists in the Database. As expected, the class ARTICLES passes. This is due to to fact that the contract performs a lower-case on the table names because, as mentioned in our assumption above, it is unlikely possible to have two tables with the same name but different capitalized letters in practice. However, this case may happen. That is why both checks (case sensitive and not) appeared in the code. The one not needed is commented. This choice has to be made by the user of the tool.

We also added the contract have_fields where we can check that a table has the fields added in the class.

```
1 class Person
2         contract.requires(:class).has_field(:Person, :name, :country)
3 end
```

5.4 Conclusion

We obverse that the tool can handle lots of constraints checking such as name matching between a file and a class. We also saw that we can easily express new contract types if these are not in the predefined list of usage contract types. We can also very easily change a custom contract into a predefined contract type if needed.

We also saw that the tool can easily handle other domains such as databases and we can imagine contracts that apply on other data such as XML files. The tool is thus quite generic and easy to extend if needed.

Chapter 6

Discussion of our solution

In this section, we will discuss about the tool in general. We will first talk about the code quality and the processes we used to enforce it. Afterwards, we will look at the tool in a dual view, what are its qualities and what are its limitations.

6.1 Code quality

To achieve readability, the code follows several naming conventions, such as snake_case for symbols, methods and variables, and CamelCase for classes and modules. Besides, to check the code quality we used some tools previously introduced : Reek for the code quality by using metrics, and **SimpleCov** to show the coverage of the code with a simple GUI and to allow us to see portions of the code which are not executed by unit tests. Reek allowed us to detect variable names which were too short, and methods which were called a lot of times in the same method.

The tool is generic enough to be applied on different kind of regularities (either in code or database used by the code or other kind of structural information). For example, we can perform checks on the source code but also match between classes and database tables names. This means we could probably also use the tool to check XML, HTML, ... structural constraints.

Since the usage contracts are executed quickly in order to have immediate feedback, the tool is efficient and can be easily improved by adding new usage contracts specific to certain applications. This means that the normal expressibility of the tool can be extended to build more complex and specific checks for a particular program. For example, the database regularities were added as a specific kind of contract.

6.2 Strengths

6.2.1 Immediate feedback

One of the main strengths of the tool is the fact that results are computed really fast and can be viewed almost instantaneously. This is due to the fact that the tool focuses on static checks that don't require a lot of computation.

Besides, we can easily view the results on a web page and manipulate the array of usage contracts (cfr. chapter 4). Programmers can easily list the results and store them in a file for logs and versioning.

6.2.2 Customisation

The programmer can create new contract types by creating a new child of the **Contract** class or by creating a new custom contract where he passes the block of code for the verification as a special parameter. This means that we can easily tackle new application domains by creating new contract types (such as database contracts for instance).

One can thus easily increase the set of contract types and enhance the tool's capabilities. Moreover, we can easily look at the verification code of the predefined contracts and inspire new contracts for the targeted project. The limit of contract types is bound to the programmer imagination.

6.2.3 Ease of use

The DSL was built to be as easy to understand as possible. The DSL tries be as close as possible to a real language : *the contract requires that the methods overriding draw call the method refresh* is transformed into *contract.requires(:methodsOverriding, :draw).call(:resfresh)*, which is really close to the original English sentence.

Besides, it is really easy to create an exception to the contract's verification. For example, the sentence *the contract requires that the methods overriding draw call the method refresh except for the class called Figure* is transformed into *contract.requires(:methodsOverriding, :draw).call(:resfresh.exception(:method, :called, :Figure)* which, again, is really close to the sentence is was derived from.

When a new contract type is declared (not a custom one), the keyword(s) used is(are) set by the programmer. He can choose the most understandable keyword and can easily use the **TestTool** to check that the keyword(s) chosen for the new contract type are not already used by an already declared usage contract type.

6.2.4 Modularity

The contracts can be added to the source code by declaring the contracts in the class definition. As soon as the class is interpreted, the contract will be executed and produce a result if it can be applied on its liable entities (a contract on a method which doesn't exist will not be executed and will produce a fail result).

If the developers want to reuse their contract declarations or don't want to make the code less readable by adding lots of contracts, they can also add the contracts in a Module and simply include the Module in the targeted class. This means we don't have to repeat contracts declaration and we can separate, if wanted, the source code and the contracts. The advantages of using an internal over an external tool is that we can name the file and Modules to easily track the links between the declared contracts and their targets. Besides, they can be put in the same folder, being easy to retrieve.

6.3 Limitations

Now that we have talked about some strengths, we can now talk about some limitations of the tool in order to assess its global quality. We tried to be as objective as possible and talk about real issues of the tool usage and drawbacks.

6.3.1 No support for singleton methods

A first limitation is the use of singleton method. Let's quickly explain with an example of a class with one method and one contract which applies to all methods of the class.

```
1  class AClass
2    contract.requires(:allMethods).maxLength(20)
3    def aMethod
4    end
5  end
6
7  # Add a new method to the instance
8  inst = AClass.new
9  def inst.putsL
10   puts "l"
11 end
12 # Add a second method to the instance
13 class << inst;send(:define_method, :notTaken){puts :notTaken}; end
```

In our example, we added two new methods to the instance **inst**. However, the contract is not

applied on the methods *putsL* because it is not added in the class definition. This means the usage contracts can only be applied on methods added to the class definition. Even the second method *notTaken* is not targeted by the contract.

Even if it is a limitation of an interesting feature, we may consider this kind of method as out of scope. The tool focuses on structural regularities while methods like in the previous example, can be evaluated as a dynamic behaviour.

However, singleton method such as

```
AClass.define_singleton_method(:test){puts "test"}
```

will be targeted by the contract. This means that we can dynamically added new static methods to a class and they will be checked by the contracts.

6.3.2 Use of hook methods

As previously explained, the tool relies on the usage of hook methods. However, if a class uses the same hook method, the tool or the class will not run correctly. For example, if we override the hook method **method_added**, the contract will no longer be applied on new methods :

```
1  class Person
2    contract.requires(:allMethods).maxLength(1)
3    def self.method_added(name)
4      puts "missing"
5      puts name
6    end
7
8    # No contract verification
9    def new_m
10         puts :new_m
11   end
12 end
```

This means that the programmer needs to be careful when applying the tool. Since the goal of the tool is not to perform dynamic checks, the programmer could consider changing the name of its hook method (add a character in the name of the hook method in the class declaration for instance) just for the time needed for contract verification. As previously explained, the hook methods used are :

- method_added
- inherited

- included

In order to avoid the problem, the method overriding one of the hook methods must perform a call *super* with the same argument as the overriding method.

One could think of a contract to be sure that *super* is called in the hook methods overridden. However, we cannot add a contract in the Module declaration because it raises errors due to the fact that Ruby is interpreted, and that not all the needed code is interpreted when the contract is added. We cannot add one of our contracts to enforce this mandatory super call.

Let's also state that the tool uses the *at_exit* method but is less prone to bugs. Overriding this method will only remove a last contract check when the class definition is completely interpreted.

6.3.3 Interpretation vs Compilation

As stated before, Ruby is an interpreted language. This means that we must interpret the code in order to check the contracts. This means we cannot add contracts in the core of the tool because not all mandatory code is interpreted when needed (cfr. chapter 5 on the tool checking itself). Also, this means that each time the tool is used, it must be reinterpreted.

Smalltalk, being compiled, didn't have this limitation. Each time the class is modified, the compiler automatically checks the usage contracts applied on the class.

Chapter 7

Improvements

This section explains some features which are not yet implemented. Some of the points may not be feasible but it's still interesting to mention them.

7.1 New keywords for quantification

The purpose of this feature is to add new keywords in requires/suggests which will allow the user to restrict the scope of the verification of the contract. For example, instead of declaring the contract :

```
1 contract.requires(:allMethods).assign(:@timestamp)
```

which will require all methods of the class to assign the instance variable timestamp, the programmer could use :

```
1 contract.requires(:oneOf, :allMethods).assign(:@timestamp)
```

which would require that at least one method assigns the variable. The new keywords could be :

- **:oneOf** : at least one liable entity must pass the verification
- **:all** : all liable entities must pass the verification
- **:exactlyOneOf** : one and only one liable entity must pass the verification
- **:atMostOneOf** : zero or one liable entity must pass the verification
- **:fewOf** : at least 25% must pass (thresholds could be configured by the user)
- **:someOf** : at least 50% must pass
- **:mostOf** : at least 75% must pass

In some cases, such a keyword could simplify a contract check (:allMethod with :oneOf instead of huge disjunctional contract with all contracts declared). In other cases, it could reduce the scope of the verification. For example, achieve at least 25% of the contracts for a given date, then 50% three months after. This could be useful during a refactoring process for instance.

7.2 Dynamic uContracts

This feature concerns the constraints that cannot be checked with static checks. This type of constraints must be executed on information that can be retrieved by the source code, such as value of variables or number of iterations for a loop. It has to be defined if the syntax will be the same as the static ones or if a new syntax will be created.

We already though of some possible dynamic constraints :

- assignment type check : the type of assignment must be of a given Class.
- boundaries : a given variable must be set as one of wanted value (example : the return code can be negative).
- condition on threads to easily detect a deadlock source or why a given variable has not a correct value.
- ...

The type returned check (the type of the return must be of a given Class) is not considered, because it is already handled by **contract.ruby**. It could be possible to integrate this gem within the tool, but it was not tested.

7.3 More predefined uContracts

Since the tool is still a prototype, it certainly misses useful contracts. New contracts may be discovered with tests on real applications and discussions with their authors, as well as known but not checked coding idioms more specific to applications such as Rails conventions.

We could, for example, write new contracts specific to Ruby on Rails 4 and check if a project, developed in Ruby on Rails 3, respects all constraints after a migration. The limit of the contracts list is bound to the Ruby programmer's imagination and needs.

Chapter 8

Conclusion

We managed to create a tool which help new developers to avoid waste of time due to unknown conventions of a framework. This is due to the fact that the tool is efficient (giving immediate feedback), and the user simply needs to interpret his code and he can obtain an output of the result.

Besides, the DSL approach allows the programmer to use a syntax similar to the source code, thus minimizing the effort to understand our tool. Our tool contains lots of keywords to specify the target of a contract which can be a class, a method or an entire hierarchy, even removing some entities from the contract check. It also allows us to declare new usage contract types that can be easily reused or to define one-shot contracts for very specific cases. The tool has enough power to tackle lots of constraints on different type of informations (Ruby classes, databases, ...).

However, the tools has also some weaknesses. Some of them, such as impossibility to add contracts in the tool itself, derived from the fact that Ruby is interpreted and not compiled. Others, such as interfering with the mechanisms of the tool, derived from the usage of hook methods.

Besides, the tool can be very useful in practice for Ruby programmers to check constraints on their code when they modify it or when they want to apply a refactoring on their project. Besides, applications, such as migration verification tool between Rails 3 and Rails 4, emerged when discussing with Ruby programmers, showing that the tool could tackle new problems in the Ruby world.

The tool also gives information about the concept of usage contracts. Lots of contracts being the same as Smalltalk (most of them were retrieved from the Smalltalk prototype), the concept could very likely be reused for other object-oriented languages. However, some contracts such as the usage of packages in Smalltalk are only feasible and relevant in Smalltalk while some contracts such as module inclusion are relevant to other languages using such concepts. Thus, not all usage contract types are not feasible for all object-oriented languages.

The primary concept of contracts heavily uses classes as liable entities but the community seems to prioritize the use of mixins instead of inheritance. It simply changes the focus but the concept of contract remains relevant. However, it shows that the concept of usage contracts can be applied on liable entities that were not described when the first version of usage contract was designed showing it is generic enough to tackle liable entities of other languages than the Smalltalk ones.

Nevertheless, the tool can be used to other domains such as database regularities. Other information can be used with usage contract as long as they have structural regularities (XML, JSON, ...) proving the concept of usage contracts can be applied to other data that the ones described by the Smalltalk prototype.

Chapter 9

Bibliography

[1]**Angela Lozano, Kim Mens, Andy Kellens. Usage Contracts: Offering Immediate Feedback on Violations of Structural Source-code Regularities. Science of Computer Programming. Elsevier, 2015. DOI: 10.1016/j.scico.2015.01.004**

[2]**Ruby documentation website** : `http://ruby-doc.org/` (Last visit the 10th May 2015)

[3]**M. Fowler. Language workbenches: The killer-app for domain specific languages?, 2005.** : `http://martinfowler.com/articles/languageWorkbench.html#InternalDsl` (Last visit the 25th April 2015)

[4]**Eiffel language website** : `https://www.eiffel.com/values/design-by-contract/`

[5]**RSpec documentation** `http://rspec.info/` (Last visit the 25th April 2015)

[6]**Discussion on behaviour driven testing** : http://blog.bughuntress.com/automated-testing/automated-testing-with-behavior-driven-testing (Last visit the 27th April 2015)

[7]**Co-evolving code and design with intensional views A case study** by Kim Mens, Andy Kellens, Frédéric Pluquet, Roel Wuyts (Version of the 12 September 2005 available on `http://www.sciencedirect.com/science/article/pii/S1477842405000394`)

[8]**Guide and conventions of Ruby on rails** : `http://guides.rubyonrails.org/active_record_basics.html` (Last visit the 5th May 2015)

[9]**PostGreSQL website** : `http://www.postgresql.org/` (Last visit the 4th April 2015)

[10]**PDFKit Git repository** : `https://github.com/pdfkit/pdfkit` (Last visit the 20th May 2015)

[11]**CodeRay Git repository** : `https://github.com/rubychan/coderay` (Last visit the 20th May 2015)

[12]**Javascript to sort a <table>** : `https://yoast.com/articles/sortable-table/` (Last visit the 19th March 2015)