Programmation Ruby «*in-the-large*» OU Comment développer une application Ruby

Guy Tremblay
Professeur

Département d'informatique
UQAM
http://www.labunix.uqam.ca/~tremblay_gu

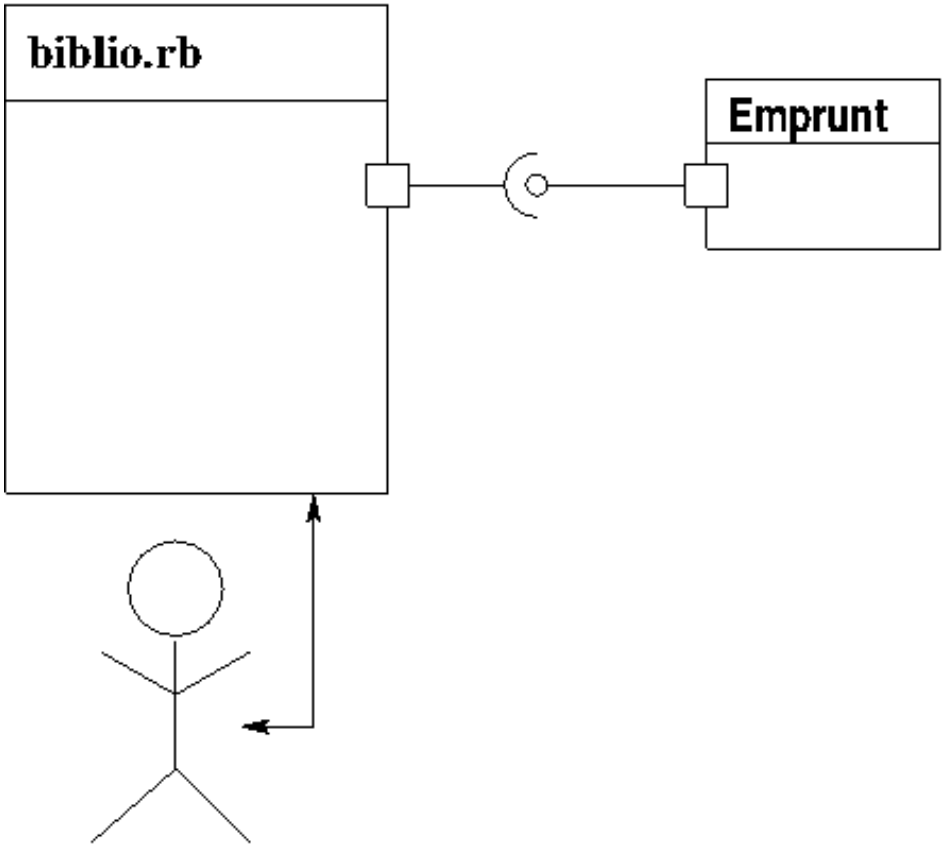
INF600A
6–13 novembre 2018

Contenu

1. Introduction/motivation

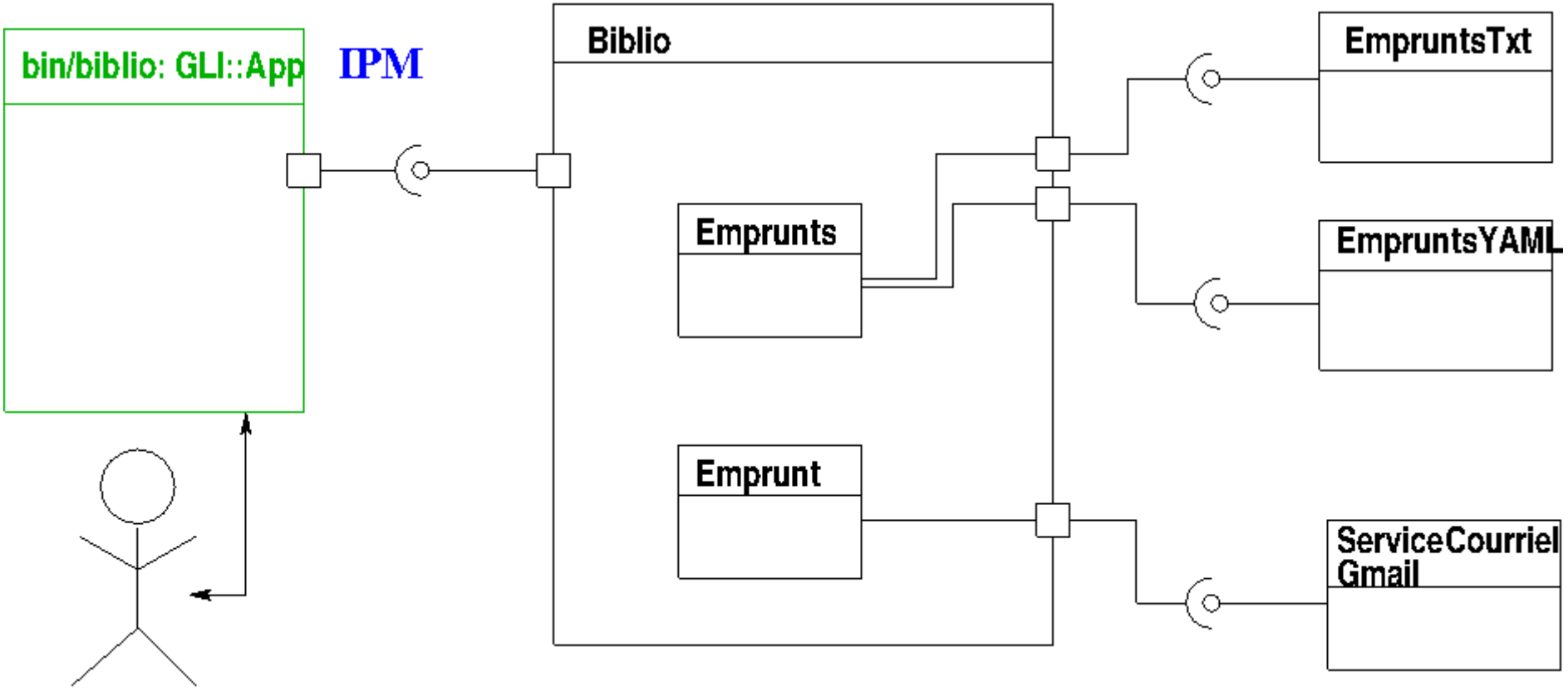
Dans ce qui suit, on va traiter de «programmation-*in-the-large*»

In-the-small = un ou deux fichiers/classes/modules



Dans ce qui suit, on va traiter de «programmation-*in-the-large*»

In-the-large = plusieurs fichiers/classes/modules



Dans ce qui suit, on va traiter de «programmation-*in-the-large*»

Programming-in-the-small

[P]rogramming in the small describes the activity of writing a small program. Small programs are typified by being small in terms of their source code size, are easy to specify, quick to code and typically perform one task or a few very closely related tasks very well.

`https://en.wikipedia.org/wiki/Programming_in_the_large_and_programming_in_the_small`

Dans ce qui suit, on va traiter de «programmation-*in-the-large*»

Programming-in-the-large

By large programs we mean systems consisting of many small programs (modules), possibly written by different people. We need languages for programming-in-the-small, i.e. languages not unlike the common programming languages of today, for writing modules. We also need a «module interconnection language» for knitting those modules together into an integrated whole [. . .].

`https://en.wikipedia.org/wiki/Programming_in_the_large_and_programming_in_the_small`

Dans le présent cours et dans le prochain cours, on va traiter de «programmation-*in-the-large*»

On va le faire en développant. . . une petite application Ruby en mode «ligne de commandes» — la prochaine série de diapos

Dans le présent cours et dans le prochain cours, on va traiter de «programmation-*in-the-large*»

On va le faire en développant. . . une petite application Ruby en mode «ligne de commandes» — la prochaine série de diapos

On va aussi examiner certains outils clés de l'écosystème Ruby — la série présente de diapos

2. Le choix de la version de Ruby avec `rvm`

De nombreuses versions du compilateur/interpréteur `Ruby` sont disponibles


```
$ rvm list known
# MRI Rubies
[ruby-]1.8.6[-p420]
[ruby-]1.8.7[-head]
[ruby-]1.9.1[-p431]
[ruby-]1.9.2[-p330]
[ruby-]1.9.3[-p551]
[ruby-]2.0.0[-p643]
[ruby-]2.1.4
[ruby-]2.1[.5]
[ruby-]2.2[.1]
[ruby-]2.2[.10]
[ruby-]2.3[.7]
[ruby-]2.4[.4]
[ruby-]2.5[.1]
[ruby-]2.6[.0-preview2]
ruby-head

# JRuby
jruby-1.6[.8]
jruby-1.7[.27]
jruby-9.1[.17.0]
jruby[-9.2.0.0]
jruby-head

# Rubinius
rbx-1[.4.3]
rbx-2.3[.0]
rbx-2.4[.1]
rbx-2[.5.8]
rbx-3[.100]
rbx-head

# Minimalistic ruby implementation - ISO 30170:2012
mruby-1.0.0
mruby-1.1.0
mruby-1.2.0
mruby-1.3.0
mruby-1[.4.0]
mruby[-head]

# Ruby Enterprise Edition
ree-1.8.6
ree[-1.8.7][-2012.02]

# Topaz
topaz

# MagLev
maglev-1.0.0
maglev-1.1[RC1]
maglev[-1.2Alpha4]
maglev-head

# Mac OS X Snow Leopard Or Newer
macruby-0.10
macruby-0.11
macruby[-0.12]
macruby-nightly
macruby-head

# IronRuby
ironruby[-1.1.3]
ironruby-head
```

L'outil `rvm` permet d'avoir, sur une même machine ou dans un même compte, différentes versions de Ruby

La commande `rvm list` affiche les versions disponibles

```
$ rvm list

rvm rubies

   jruby-1.7.16.1 [ x86_64 ]
   ruby-1.8.7-head [ x86_64 ]
=> ruby-1.9.3-p550 [ x86_64 ]
 * ruby-2.1.4 [ x86_64 ]
   ruby-2.2.4 [ x86_64 ]

# => - current
# =* - current && default
#  * - default

$ ruby --version
ruby 1.9.3p550 (2014-10-27 revision 48165) [x86_64-linux]
```

L'outil `rvm` permet d'avoir, sur une même machine ou dans un même compte, différentes versions de Ruby

La commande `rvm use` permet de passer d'une version à une autre

```
$ rvm use ruby-2.2.4
Using /home/tremblay_gu/.rvm/gems/ruby-2.2.4

$ ruby --version
ruby 2.2.4p230 (2015-12-16 revision 53155) [x86_64-linux]
```

3. Les *gems* Ruby

Qu’est-ce qu’un *gem* Ruby ?

3.1 Qu’est-ce qu’un *gem* Ruby ?

Qu’est-ce qu’un *gem* ?

Un *gem* est un composant logiciel

Dans l’écosystème Ruby, un *gem* est un composant logiciel — une **bibliothèque**, un paquetage de code — pouvant être utilisé par un autre programme.

Qu'est-ce qu'un *gem* ?

Un *gem* est un composant logiciel

Dans l'écosystème Ruby, un *gem* est un composant logiciel — une **bibliothèque**, un paquetage de code — pouvant être utilisé par un autre programme.

Un *gem* peut aussi être une application

Un *gem* peut aussi fournir une application, par exemple, une application utilisable par l'intermédiaire de la ligne de commandes.

Un *gem* est un composant logiciel — pouvant être distribué et installé

La commande `gem list` affiche les `gems` installés avec leur numéro de version

```
$ gem list

*** LOCAL GEMS ***

aruba (0.13.0)
bigdecimal (1.2.6)
builder (3.2.2)
bundle (0.0.1)
bundler (1.11.2)
bundler-unload (1.0.2)
childprocess (0.5.9)
contracts (0.13.0)
cucumber (2.3.2)
cucumber-core (1.4.0)
cucumber-wire (0.0.1)
diff-lcs (1.2.5)
executable-hooks (1.3.2)
ffi (1.9.10)
gem-wrappers (1.2.7)
gherkin (3.2.0)
io-console (0.4.3)
json (1.8.1)
multi_json (1.11.2)
multi_test (0.1.2)
psych (2.0.8)
rake (10.4.2)
rdoc (4.2.0)
rspec-expectations (3.4.0)
rspec-support (3.4.1)
rubygems-bundler (1.4.4)
rvm (1.11.3.9)
thor (0.19.1)
```

Un *gem* est un composant logiciel — pouvant être distribué et installé

La commande `gem install` permet d'obtenir et d'installer, «à bras», un *gem*

Installation de la dernière version d'un gem

```
$ gem install gli
Fetching: gli-2.13.4.gem (100%)
Successfully installed gli-2.13.4
Parsing documentation for gli-2.13.4
Installing ri documentation for gli-2.13.4
Done installing documentation for gli after 1 seconds
1 gem installed
```

Un *gem* est un composant logiciel — pouvant être distribué et installé

La commande `gem install` permet d'obtenir et d'installer, «à bras», un *gem*

Installation de la dernière version d'un gem

```
$ gem install gli
Fetching: gli-2.13.4.gem (100%)
Successfully installed gli-2.13.4
Parsing documentation for gli-2.13.4
Installing ri documentation for gli-2.13.4
Done installing documentation for gli after 1 seconds
1 gem installed
```

Installation d'une version spécifique d'un gem

```
$ gem install gli -v 2.11.0
Fetching: gli-2.11.0.gem (100%)
Successfully installed gli-2.11.0
Parsing documentation for gli-2.11.0
Installing ri documentation for gli-2.11.0
Done installing documentation for gli after 0 seconds
1 gem installed
```

3.2 Comment utiliser un *gem*

Certains gems sont des applications


```
$ gem list | grep rake
rake (11.1.1, 10.4.2, 10.3.2, 10.1.0, 0.9.6)

$ rake --version
rake, version 11.1.1
```

Certains gems sont des composants logiciels qu'on peut utiliser dans un programme avec `require`


```
$ gem list | grep pruby
pruby (0.1.1)

$ pruby
bash: pruby: commande inconnue...
Commande similaire: 'ruby'

$ irb
>> require 'pruby'
=> true

>> PRuby::VERSION
=> "0.1.1"
```

Certains gems sont des composants logiciels qu'on peut utiliser dans un programme avec `require`


```
$ gem list | grep pruby
pruby (0.1.1)

$ pruby
bash: pruby: commande inconnue...
Commande similaire: 'ruby'

$ irb
>> require 'pruby'
=> true

>> PRuby::VERSION
=> "0.1.1"
```

Note : Et certains *gems* sont à la fois composants (utilisables via `require`) et applications (utilisables comme programmes).

L'instruction `require` charge le code d'un *gem* . . . et n'effectue ce chargement qu'une seule fois


```
$ irb
>> require 'pruby'
=> true

>> PRuby::VERSION
=> "0.1.1"

>> require 'pruby'
=> false

>> PRuby::VERSION
=> "0.1.1"
```

Note : Il existe aussi une instruction `load`, qu'on utilise rarement, et qui charge le fichier même si déjà chargé

L'instruction `require` utilise par défaut la version (installée) la plus récente du *gem*


```
$ gem list | grep rake
rake (11.1.1, 10.4.2, 10.3.2, 10.1.0, 0.9.6)

$ irb
>> require 'rake'
=> true

>> Rake::VERSION
=> "11.1.1"
```

L'instruction `gem` utilisée avant `require` permet de spécifier une version spécifique à utiliser


```
$ gem list | grep rake
rake (11.1.1, 10.4.2, 10.3.2, 10.1.0, 0.9.6)

$ irb
>> gem 'rake', '=10.4.2'
=> true

>> require 'rake'
=> true

>> Rake::VERSION
=> "10.4.2"
```

Si un *gem* n'est pas installé on ne peut évidemment pas l'utiliser avec `require`


```
$ gem list | grep pruby

$ pruby
bash: pruby : commande introuvable

$ irb
>> require 'pruby'
LoadError: cannot load such file -- pruby
  from /home/tremblay_gu/.rvm/rubies/ruby-2.1.4/lib/
       ruby/site_ruby/2.1.0/rubygems/core_ext/
       kernel_require.rb:54:in 'require'
  from /home/tremblay_gu/.rvm/rubies/ruby-2.1.4/lib/
       ruby/site_ruby/2.1.0/rubygems/core_ext/
       kernel_require.rb:54:in 'require'
  from (irb):1
  from /home/tremblay_gu/.rvm/rubies/ruby-2.1.4/bin/
       irb:11:in '<main>'
```

3.3 Comment spécifier les *gems* dont on a besoin

Lorsqu'on crée un *gem* pour un projet `foo`, on utilise (typiquement) une structure standard de répertoires


```
$ tree foo
foo
|-- bin
|   `-- foo
|-- features
|   |-- foo.feature
|   |-- step_definitions
|   |   `-- foo_steps.rb
|   `-- support
|       `-- env.rb
|-- foo.gemspec
|-- foo.rdoc
|-- Gemfile
|-- lib
|   |-- foo
|   |   `-- version.rb
|   `-- foo.rb
|-- Rakefile
|-- README.rdoc
`-- test
    |-- default_test.rb
    `-- test_helper.rb

7 directories, 13 files
```

Le fichier `foo.gemspec` décrit les méta-informations du *gem* ainsi que ses dépendances

Dépendances = quels gems sont nécessaires pour que `foo` fonctionne

```
$ cat foo.gemspec
require File.join([File.dirname(__FILE__),'lib','foo','version
spec = Gem::Specification.new do |s|
  s.name = 'foo'
  s.version = Foo::VERSION
  s.author = 'Guy Tremblay'
  s.email = tremblay.guy@uqam.ca
  s.homepage = 'http://www.labunix.uqam.ca/~tremblay_gu'
  s.platform = Gem::Platform::RUBY
  s.summary = 'Mon projet foo'
  s.files = `git ls-files`.split("")
  s.require_paths << 'lib'
  s.bindir = 'bin'
  s.executables << 'foo'

  s.add_development_dependency('rake')
  s.add_development_dependency('rdoc')
  s.add_development_dependency('aruba')
  s.add_runtime_dependency('gli','2.12.0')
end
```

De nombreux outils génèrent automatiquement la structure de répertoires et le fichier gemspec requis

Exemple : L'outil `gli`, pour définir des interfaces personne-machine «à la `git`»

```
$ gli --help
NAME
    gli - create scaffolding for a GLI-powered application

SYNOPSIS
    gli [global options] command [command options] [arguments.

VERSION
    2.12.0

GLOBAL OPTIONS
    --help             - Show this message
    -n                 - Dry run; dont change the disk
    -r, --root=arg     - Root dir of project (default: .)
    -v                 - Be verbose
    --version          - Display the program version

COMMANDS
    help               - Shows a list of commands or help for one
    init, scaffold     - Create a new GLI-based project
```

De nombreux outils génèrent automatiquement la structure de répertoires et le fichier gemspec requis

Exemple : L'outil `gli`, pour définir des interfaces personne-machine «à la `git`»

```
$ gli init bar
Creating dir ./bar/lib...
Creating dir ./bar/bin...
Creating dir ./bar/test...
Created ./bar/bin/bar
Created ./bar/README.rdoc
Created ./bar/bar.rdoc
Created ./bar/bar.gemspec
Created ./bar/test/default_test.rb
Created ./bar/test/test_helper.rb
Created ./bar/Rakefile
Created ./bar/Gemfile
Created ./bar/features
Created ./bar/lib/bar/version.rb
Created ./bar/lib/bar.rb
```

De nombreux outils génèrent automatiquement la structure de répertoires et le fichier gemspec requis

Exemple : L'outil `gli`, pour définir des interfaces personne-machine «à la `git`»

```
$ cat bar/bar.gemspec
# Ensure we require the local version and not one we might have installed already
require File.join([File.dirname(__FILE__),'lib','bar','version
spec = Gem::Specification.new do |s|
  s.name = 'bar'
  s.version = Bar::VERSION
  s.author = 'Your Name Here'
  s.email = 'your@email.address.com'
  s.homepage = 'http://your.website.com'
  s.platform = Gem::Platform::RUBY
  s.summary = 'A description of your project'
  s.files = `git ls-files`.split("")
  s.require_paths << 'lib'
  s.has_rdoc = true
  s.extra_rdoc_files = ['README.rdoc','bar.rdoc']
  s.bindir = 'bin'
  s.executables << 'bar'
  s.add_development_dependency('rake')
  ...
  s.add_runtime_dependency('gli','2.12.0')
end
```

La spécification des dépendances dans un `.gemspec`… ne rend pas les *gems* disponibles

- Le fichier `.gemspec` indique, de façon claire et explicite, les dépendances, i.e., les *gems* requis.

  Mais… il faut ensuite s'assurer que ces *gems* sont effectivement installés !

- Deux approches possibles :
  - Manuellement — «à bras» ☹
  - Automatiquement — «à l'aide d'un outil» ☺

La spécification des dépendances dans un `.gemspec`... ne rend pas les *gems* disponibles

- Le fichier `.gemspec` indique, de façon claire et explicite, les dépendances, i.e., les *gems* requis.

  Mais... il faut ensuite s'assurer que ces *gems* sont effectivement installés !

- Deux approches possibles :
  - Manuellement — «à bras» ☹
  - Automatiquement — «à l'aide d'un outil» ☺

  `bundler`

3.4 L'outil `bundler` pour gérer et installer des *gems*

L'outil `bundler` permet d'installer automatiquement des *gems* à partir d'un `Gemfile`

Les gems requis sont spécifiés dans le fichier `Gemfile`

```
$ cat Gemfile
source 'https://rubygems.org'
gemspec

$ cat Gemfile.lock
cat: Gemfile.lock: Aucun fichier ou dossier de ce type

$ bundle install
Fetching gem metadata from https://rubygems.org/.........
Resolving dependencies...
Using rake 11.1.1
Using ffi 1.9.10
...
Using bundler 1.7.4
Your bundle is complete!
Use `bundle show [gemname]` to see where a bundled gem is inst
```

Note : `bundle install` examine le fichier `Gemfile`.
Ici, ce fichier fait simplement référence au fichier `gemspec` courant.

L'outil `bundler` permet d'installer automatiquement des *gems* à partir d'un `Gemfile`

Les gems requis/utilisés sont enregistrés dans le fichier `Gemfile.lock`

```
$ cat Gemfile.lock
PATH
  remote: .
  specs:
    foo (0.0.1)
      gli (= 2.12.0)

GEM
  remote: https://rubygems.org/
  specs:
    aruba (0.14.1)
      childprocess (~> 0.5.6)
      contracts (~> 0.9)
      cucumber (>= 1.3.19)
      ffi (~> 1.9.10)
      rspec-expectations (>= 2.99)
    ...
    rdoc (4.2.2)
      json (~> 1.4)
    rspec-expectations (3.4.0)
      diff-lcs (>= 1.2.0, < 2.
      rspec-support (~> 3.4.0)
    rspec-support (3.4.1)
    thor (0.19.1)

PLATFORMS
  ruby

DEPENDENCIES
  aruba
  rake
  rdoc
```

Note : Si on développe une application — et non un *gem* — alors le fichier `Gemfile.lock` devrait être mis sous contrôle des versions.

Remarque importante concernant `gem install` et `bundle install`

- Dans certaines configurations, par défaut, ces commandes installent les *gems* dans un répertoire global à tous les usagers

  $\Rightarrow$ Il faut être `root` ou exécuter en `sudo` ☹

Remarque importante concernant `gem install` et `bundle install`

- Dans certaines configurations, par défaut, ces commandes installent les *gems* dans un répertoire global à tous les usagers

  $\Rightarrow$ Il faut être `root` ou exécuter en `sudo` ☹

- Par exemple, sur `java`, **vous ne pouvez pas installer de *gems* de cette façon !**

Remarque importante concernant `gem install` et `bundle install`

- Dans certaines configurations, par défaut, ces commandes installent les *gems* dans un répertoire global à tous les usagers

  $\Rightarrow$ Il faut être `root` ou exécuter en `sudo` ☹

- Par exemple, sur `java`, **vous ne pouvez pas installer de *gems* de cette façon !**

- Par contre, si nécessaire, vous pouvez installer des *gems* dans votre espace personnel :

```
$ bundle install --path vendor/bundle
```

3.5 Comment créer un *gem* qu'on pourra ensuite distribuer

Pour créer un *gem* `foo` qu'on pourra distribuer, on doit tout d'abord finaliser le fichier `foo.gemspec`

On ajoute les informations sur l'auteur, la description détaillée du *gem*, etc.

```
$ emacs foo.gemspec
  ...
  s.author = 'Guy Tremblay'
  s.email = 'tremblay.guy@uqam.ca'
  s.homepage = 'http://www.labunix.uqam.ca'
  s.platform = Gem::Platform::RUBY
  s.summary = 'Une description sommaire de foo ...'
  s.description = 'Une description plus detaillee
                   de foo ...
                   ...'

  ...
  s.add_development_dependency('...')
  s.add_runime_dependency('...')
  ...
```

Note : Le fichier `foo.gemspec`, utilisé avec `gem build`, va permettre de créer un *gem* distribuable

On crée le *gem* avec la commande `gem build`

Il manque la licence d'utilisation... donc à compléter, mais autrement, le *gem* peut être distribué

```
$ ls -o foo-0.0.1.gem
ls: foo-0.0.1.gem: No such file or directory

$ gem build foo.gemspec
WARNING:  licenses is empty, but is recommended. Use a license
WARNING:  See http://guides.rubygems.org/specification-referen
  Successfully built RubyGem
  Name: foo
  Version: 0.0.1
  File: foo-0.0.1.gem

$ ls -o foo-0.0.1.gem
-rw-r--r-- 1 tremblay_gu 13824 [...] foo-0.0.1.gem
```

Note : «`ls -o`» ≈ «`ls -l`»

Le *gem* peut alors être distribué, installé, etc.

Pas de documentation `ri` pour ce *gem*, donc on utilise l'option «`--no-ri`»

```
$ gem list foo

*** LOCAL GEMS ***


$ gem install --no-ri foo-0.0.1.gem
Successfully installed foo-0.0.1
1 gem installed


$ gem list foo

*** LOCAL GEMS ***

foo (0.0.1)
```

3.6 En résumé pour développer une application qui utilise d'autres *gems*

En résumé, pour développer un *gem* `mon-gem` qui utilise divers autres *gems* I

1. On définit un fichier `Gemfile` et un fichier `.gemspec` :

```
$ pwd
/home/tremblay_gu/INF600A/Programmes/Ruby/MonGem

$ tree -a
.
|-- Gemfile
`-- mon-gem.gemspec

0 directories, 2 files

$ cat Gemfile
source 'https://rubygems.org'
gemspec
```

En résumé, pour développer un *gem* `mon-gem` qui utilise divers autres *gems* II


```
$ cat mon-gem.gemspec
spec = Gem::Specification.new do |s|
  s.name = 'mon-gem'
  s.version = '0.0.1'
  s.author = 'Guy Tremblay'
  ...
  s.email = 'tremblay.guy@uqam.ca'
  s.summary = 'Un exemple simple de gem'
  s.description = 'Un exemple [...]'

  s.add_development_dependency('rake')
  s.add_development_dependency('minitest')
  s.add_runtime_dependency('gli')
end
```

En résumé, pour développer un *gem* `mon-gem` qui utilise divers autres *gems* III

2. On exécute `bundle` :

```
$ bundle install --path vendor/bundle
Fetching gem metadata from https://rubygems.org/.........
Resolving dependencies...
Fetching rake 12.3.1
Installing rake 12.3.1
Using bundler 1.16.2
Fetching gli 2.18.0
Installing gli 2.18.0
Fetching minitest 5.11.3
Installing minitest 5.11.3
Using mon-gem 0.0.1 from source at `.`
Bundle complete! 3 Gemfile dependencies, 5 gems now instal
Bundled gems are installed into `./vendor/bundle`
```

En résumé, pour développer un *gem* `mon-gem` qui utilise divers autres *gems* IV

3. L'exécution de `bundle install` a alors créé un fichier `.bundle/config` (qui indique où sont les *gems*), un fichier `Gemfile.lock` (qui indique les *gems* téléchargés) et a chargé les *gems* requis dans `vendor/bundle` :

En résumé, pour développer un *gem* `mon-gem` qui utilise divers autres *gems* V


```
$ tree -a -L 4
.
|-- .bundle
|   `-- config
|-- Gemfile
|-- Gemfile.lock
|-- mon-gem.gemspec
`-- vendor
    `-- bundle
        `-- ruby
            `-- 2.5.0

4 directories, 3 files
```

En résumé, pour développer un *gem* `mon-gem` qui utilise divers autres *gems* VI


```
$ cat .bundle/config
---
BUNDLE_PATH: vendor/bundle
BUNDLE_DISABLE_SHARED_GEMS: '1'
```

En résumé, pour développer un *gem* `mon-gem` qui utilise divers autres *gems* VII


```
$ cat Gemfile.lock
PATH
  remote: .
  specs:
    mon-gem (0.0.1)
      gli

GEM
  remote: https://rubygems.org/
  specs:
    gli (2.18.0)
    minitest (5.11.3)
    rake (12.3.1)

PLATFORMS
  ruby

DEPENDENCIES
  minitest
  mon-gem!
  rake

BUNDLED WITH
   1.16.2
```

En résumé, pour développer un *gem* `mon-gem` qui utilise divers autres *gems* VIII


```
$ tree -L 2 vendor/bundle/ruby/2.5.0
vendor/bundle/ruby/2.5.0
|-- bin
|   |-- gli
|   `-- rake
|-- build_info
|-- cache
|   |-- gli-2.18.0.gem
|   |-- minitest-5.11.3.gem
|   `-- rake-12.3.1.gem
|-- doc
|-- extensions
|-- gems
|   |-- gli-2.18.0
|   |-- minitest-5.11.3
|   `-- rake-12.3.1
`-- specifications
    |-- gli-2.18.0.gemspec
    |-- minitest-5.11.3.gemspec
    `-- rake-12.3.1.gemspec

10 directories, 8 files
```

4. Le *gem* `rake` pour l'automatisation des tâches

Qu'est-ce que `rake` ?

Rake is a build language, similar in purpose to make and ant. Like make and ant it's a Domain Specific Language, unlike those two it's an internal DSL programmed in the Ruby language.

Source: *http ://martinfowler.com/articles/rake.html*

Qu'est-ce que `rake` ?

Rake is a build language, similar in purpose to make and ant. Like make and ant it's a Domain Specific Language, unlike those two it's an internal DSL programmed in the Ruby language.

Source: *http ://martinfowler.com/articles/rake.html*

[Rake] came about as an experiment to see if Make's functionality could be easily reproduced by creating an internal DSL in Ruby. The answer was "yes," and Rake was born.

Source: *«Continuous Delivery», Humble & Farley*

Caractéristiques de `rake`

- Comme `Make` : `rake` se fonde uniquement sur les notions de tâches et de dépendances.
- Comme `Make` : `rake`, bien qu'écrit en Ruby, peut être utilisé avec n'importe quel langage
- Contrairement à `Make` : un script `rake` est un programme Ruby comme n'importe quel autre (DSL interne) !
- Contrairement à `Make` : les commandes (e.g., manipulation de fichiers) d'un script `rake` peuvent être écrites de façon indépendante de la plateforme

Exemple = compilation d'un fichier `.c` : Avec `Make`


```
default: run

# Execution du programme.
run: hello
	./hello

# Generation de l'executable.
hello: hello.c
	gcc -o hello hello.c

# Nettoyage.
clean:
	rm -f hello hello.o
```

Exemple = compilation d'un fichier `.c` : Avec `rake`


```
task :default => :run

desc 'Execution du programme'
task :run => 'hello' do
  sh %{./hello}
end

desc 'Generation de l\'executable'
file 'hello' => ['hello.c'] do
  sh %{gcc -o hello hello.c}
end

desc 'Nettoyage'
task :clean do
  rm_f 'hello hello.o'
end
```

Quelques exemples d'exécution

Construction de la cible par défaut

```
$ rake
gcc -o hello hello.c
./hello
Hello, World!
```

Quelques exemples d'exécution

Construction de la cible par défaut

```
$ rake
gcc -o hello hello.c
./hello
Hello, World!
```

Construction d'une cible déjà dérivée ⇒ aucune action

```
$ rm hello

$ rake hello
gcc -o hello hello.c

$ rake hello

$
```

Quelques exemples d'exécution

Liste des tâches disponibles, avec description

```
$ rake -T
rake clean  # Nettoyage
rake hello  # Generation de l'executable
rake run    # Execution du programme
```

Liste des tâches, avec ou sans description

```
$ rake -T -A
rake clean    # Nettoyage
rake default  #
rake hello    # Generation de l'executable
rake run      # Execution du programme
```

Quelques exemples d'exécution

Dry run — indique ce qui serait exécuté, mais sans exécuter

```
$ rake -n
** Invoke default (first_time)
** Invoke run (first_time)
** Invoke hello (first_time, not_needed)
** Invoke hello.c (first_time, not_needed)
** Execute (dry run) run
** Execute (dry run) default
```

L'utilisation de règles implicites permet de réduire le nombre de tâches explicitement spécifiées

Règle style `Make`

```
rule '.o' => '.c' do |task|
  sh %{gcc -o #{task.name} #{task.source}}
end
```

L'utilisation de règles implicites permet de réduire le nombre de tâches explicitement spécifiées

Règle avec *pattern-matching*

```
rule /\.o$/ => '.c' do |task|
  sh %{gcc -o #{task.name} #{task.source}}
end
```

L'utilisation de règles implicites permet de réduire le nombre de tâches explicitement spécifiées

Règle avec détection dynamique

```
articles = Rake::FileList.new( '**/*.md' ) do |files|
  files.exclude( '~*' )
  files.exclude( /^temp.+\// )
  files.exclude { |file| File.zero? file }
  ...
end

detect_files = lambda do |task|
  articles.detect { |article| article.ext == task.ext }
end

rule '.html' => detect_file do |task|
   sh %{...}
end
```

Source: Adapté/simplifié de «*Rake Task Management Essentials*», Koleshko

Les avantages/désavantages de `rake`

Avantages

- DSL Interne $\Rightarrow$ Puissant et expressif ☺
- Disponible sur toute plateforme où Ruby est disponible

  Notamment, `jruby` fonctionne sur toute plateforme avec une JVM (machine virtuelle Java)

Les avantages/désavantages de `rake`

Avantages

- DSL Interne ⇒ Puissant et expressif ☺
- Disponible sur toute plateforme où Ruby est disponible

  Notamment, `jruby` fonctionne sur toute plateforme avec une JVM (machine virtuelle Java)

Désavantages

- DSL Interne ⇒ Il faut connaitre (un peu) Ruby ☹
- Il faut installer Ruby et divers *gems*
- `jruby` est parfois un peu lent à démarrer ⇒ un appel à `rake` avec `jruby` est plus lent qu'un appel à `make`

5. Le *gem* `gli` pour la spécification de suites de commandes

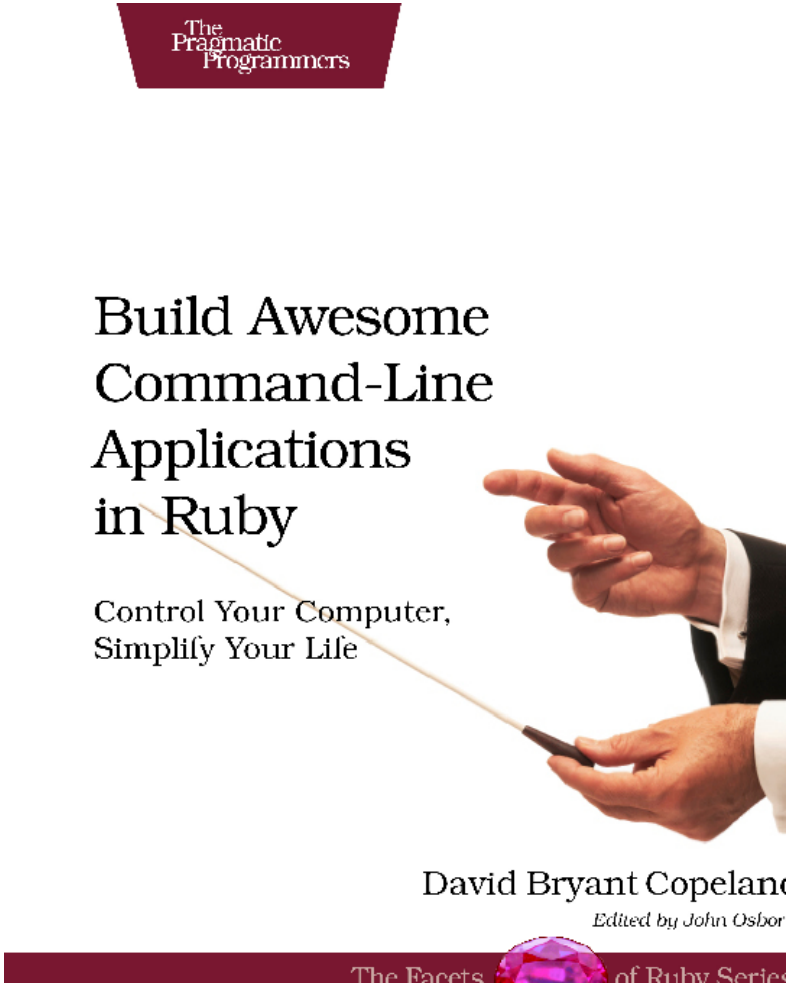
Le *gem* `gli` est utilisé pour spécificier des suites de commandes «à la `git`»

GLI lets you create command-suite (i.e. git-like) style applications in Ruby very easily.

Source: `https://github.com/davetron5000/gli/wiki`

`gli` est décrit dans le livre de Copeland (2012)

`gli` = *git like interface*
command line parser



Le *gem* `gli` fournit un DSL pour la spécification de suite de commandes

DSL = *Domain-Specific Language*

A [programming] language offering expressive power focused on a particular problem domain, such as a specific class of applications or application aspect.

Source: *K. Czarnecki, 2005*

Parfois aussi appelé un «petit langage» — «*a little language*».

Note : On traitera plus en détail de DSL dans un prochain cours.

Le *gem* `gli` définit des méthodes qui permettent de définir les options et commandes d'une application

Description d'options

```
desc "Description d'une switch"
switch [:o, :option]

desc "Description d'un flag"
default_value "Valeur par defaut"
arg_name "Nom de l'argument"
flag [:f, :flag]
```

Le *gem* `gli` définit des méthodes qui permettent de définir les options et commandes d'une application

Description d'options

```
desc "Description d'une switch"
switch [:o, :option]

desc "Description d'un flag"
default_value "Valeur par defaut"
arg_name "Nom de l'argument"
flag [:f, :flag]
```

Éléments du DSL = méthodes fournies par `gli`

- `desc`
- `switch`
- `default_value`
- `arg_name`
- `flag`

Le *gem* `gli` définit des méthodes qui permettent de définir les options et commandes d'une application

Description de commandes

```
desc 'Description de la commande'
arg_name 'Liste des arguments'
command :nom_de_la_commande do |c|
  c.desc 'Option pour commande'
  c.switch :s

  c.action do |global_options,options,args|
    # Mise en oeuvre de la commande
    ...
  end
end
```

Le *gem* `gli` définit des méthodes qui permettent de définir les options et commandes d'une application

Description de commandes

```
desc 'Description de la commande'
arg_name 'Liste des arguments'
command :nom_de_la_commande do |c|
  c.desc 'Option pour commande'
  c.switch :s

  c.action do |global_options,options,args|
    # Mise en oeuvre de la commande
    ...
  end
end
```

Autres éléments du DSL = méthodes fournies par `gli`

- `command`
- `action`

Le *gem* `gli` fournit aussi une commande qui permet de créer le squelette d'une application

The simplest way to get started is to create a scaffold project. For example, a command-suite app named "todo" that has the commands "list", "add" and "complete" is created with:

```
> gli init todo list add complete
```

A new `./todo` directory is created containing the app. View the basic output of the scaffold with:

```
> cd todo
> bundle exec bin/todo help
```

Source: `https://github.com/davetron5000/gli/wiki`

Un exemple plus détaillé d'utilisation de `gli` : `minised`

À venir au prochain cours !

Références

G.T. Brown.

Ruby Best Practices.

O'Reilly, 2009.

D. Chelimsky, D. Astels, Z. Dennis, A. Hellesoy, B. Helmkamp, and D. North.

The RSpec Book : Behaviour Driven Development with RSpec, Cucumber, and Friends.

The Pragmatic Bookshelf, 2010.

D.B. Copeland.

Build Awesome Command-Line Applications in Ruby : Control Your Computer, Simplify Your Life.

The Pragmatic Bookshelf, 2012.

M. Wynne and A. Hellesoy.

The Cucumber Book : Behaviour-Driven Development for Testers and Developers.

The Pragmatic Bookshelf, 2012.