Travail pratique #3 — INF600A (gr. 20)

Automne 2018

Date de remise: lundi 17 décembre, 13h00.
Tout travail remis *après* l'heure indiquée sera considéré en retard (pénalité = 10%/jour, partiel ou complet). **Aucun** travail ne sera accepté après mercredi 19 décembre, 13h00.

Ce travail est à faire seul ou avec une autre personne (donc max. 2).

Développement d'une application Ruby en «ligne de commandes» à l'aide du *gem* `gli`

Objectif

Dans le travail pratique #2, vous vous êtes familiarisés avec la programmation en Ruby. Toutefois, le programme développé était relativement simple, ne comportant qu'un script principal et quelques fichiers définissant une classe, des constantes et quelques méthodes auxiliaires.

Dans le présent travail, vous allez à nouveau utiliser Ruby, mais cette fois pour développer une application mieux structurée, qui pourrait éventuellement être distribuée sous forme d'un *gem*.

Choix du sujet

Pour ce travail, c'est vous qui choisissez l'application que vous allez développer. **Toutefois, vous devez faire approuver votre choix**. **Donc, au plus tard mardi 4 décembre**, vous devez m'indiquer le sujet sur lequel vous allez travailler, et ce en complétant le formulaire Web suivant :

`http://www.labunix.uqam.ca/~tremblay_gu/INF600A/Devoir3/choisir.cgi`

Lorsque vous soumettrez ce formulaire,[1] je recevrai un courriel m'indiquant votre choix. Si je ne vous réponds pas, alors c'est que votre choix est approuvé — «Qui ne dit mot, consent!» Par contre, si j'ai des questions ou suggestions quant à votre choix, je vous contacterai par courriel — à votre adresse UQAM!

[1] Bouton «Enregistrer le choix».

Caractéristiques de l'application et contraintes à respecter

Votre application devra respecter un certain nombre de contraintes :

- L'interface personne-machine doit être en «lignes de commandes».
- Le squelette de l'application doit être initialement créé à l'aide de la commande «`gli scaffold`» et le programme principal (`bin/votre-appli`), qui analyse les commandes et les options, doit être défini **à l'aide du DSL du *gem* `gli`**.

  Ce programme principal doit contenir essentiellement **la partie liée à l'interface personne–machine**, donc la partie qui effectue l'analyse des commandes et des options. L'action associée à une commande doit ensuite appeller une ou des méthodes qui font «**le vrai travail**» (la logique «métier»).
- Votre application doit manipuler des **données persistantes** — une *base de données*. Il peut s'agir, comme dans les devoirs 1 et 2, d'une base de données textuelles de style CSV[2], i.e., une série de lignes où chaque ligne contient des valeurs séparées par un délimiteur. Si vous le désirez, vous pouvez aussi expérimenter avec d'autres formes de données textuelles, par exemple, YAML ou PStore.
- Tel qu'indiqué plus haut, pour réaliser «le vrai travail», vous devez définir **des classes et méthodes auxiliaires**, lesquelles doivent respecter une hiérarchie de fichiers appropriée (sous-répertoire `lib`) et être définies à l'intérieur **d'un `module` indépendant portant le nom de votre application**.

  Pour des exemples, voir les applications `minised`[3] (diapos et labo #6) et `gv`[4] (diapos). En fait, **vous pouvez réutiliser des parties du code de ces applications**... en autant que vous indiquiez clairement la provenance.
- Vous devez définir **des exemples d'exécution** et **quelques tests d'acceptation**.

  Les exemples d'exécution doivent pouvoir se lancer à partir du fichier `Rakefile` — voir plus bas. Ces exemples doivent couvrir les diverses commandes et la grande majorité des options (globales ou locales). Par contre, pour simplifier, ces exemples peuvent être uniquement pour des cas normaux, sans erreur (*happy paths*).

  Quant aux tests d'acceptation, eux aussi doivent pouvoir être exécutés via le `Rakefile`. Par contre, ils n'ont pas besoin d'être complets ou exhaustifs : il suffit qu'il y ait **un (1) test d'acceptation pour chaque commande** (avec options intéressantes), test qui traitera un «**cas normal**» («*happy path*»). En d'autres mots, pas besoin de tester ni les divers cas anormaux ou erronés, ni toutes les combinaisons d'options.

  Ces tests d'acceptation devraient être écrits avec `MiniTest`, comme dans les devoirs 1 et 2 ou dans les applications `minised` et `gv` — et vous pouvez utiliser (directement ou en les modifiant/adaptant) les méthodes de `test_helper.rb`.[5]

[2] *Comma-Separated Values*.
[3] `git clone http://www.labunix.uqam.ca/~tremblay_gu/git/minised.git`
[4] `git clone http://www.labunix.uqam.ca/~tremblay_gu/git/gv.git`
[5] Et vous pouvez utiliser le **style** d'assertions que vous préférez, i.e., avec `assert_*` ou avec `must_*`.

Combien de commandes et de tests?

L'objectif du présent travail est de montrer que vous comprenez comment développer un *gem* Ruby structuré correctement, que vous êtes capable d'utiliser un DSL tel que celui de `gli`, et que vous pouvez définir des tests d'acceptation simples.

Donc, au niveau «quantitatif» — puisque certains se poseront sûrement la question — votre application pourrait ne comporter que 3–4 commandes, mais doit comporter au moins **une (1) option globale** (i.e., applicable à toutes les commmandes, comme `--depot` dans les devoirs 1/2), et au moins une (1) commande avec **une option locale** (comme les commandes `init`, `ajouter`, `selectionner`, `trier` ou `lister` dans les devoirs 1/2).

Note : Tel qu'indiqué dans le barème, des points **bonus** pourront être attribués pour la difficulté ou l'originalité de votre application...

Choix de l'environnement de développement/tests et spécification des *gems*

Contrairement aux deux premiers devoirs, j'accepterai des applications n'ayant pas été développées et testées sur `java.labunix.uqam.ca`. Toutefois, **voici certaines contraintes à respecter** :

a. Vous devez **indiquer clairement et explicitement** dans votre document (voir plus bas) dans quel environnement vous avez développé votre application (`java.labunix`, machine personnelle Linux/MacOS/Cygwin).

b. **Vous devez utiliser Ruby 2.5.1 (MRI)** — donc «`rvm use ruby-2.5.1`»!

c. Votre fichier `.gemspec` doit être complet quant aux *gems* requis. Pour tester votre application, je commencerai par exécuter «`bundle install --path vendor/bundle`», puis je lancerai l'exécution des exemples et ensuite des tests (voir plus bas)... si ça ne fonctionne pas, vous aurez alors un gros problème ☹

Ce que vous devez remettre — de façon électronique

a. Document **PDF**, **remis de façon électronique avec Oto** (voir plus bas), contenant les éléments suivants :

   (1) Une brève introduction donnant une vue d'ensemble de ce que fait votre application.

   (2) Une description plus détaillée des principales fonctionnalités **mises en œuvre par** votre application — donc celles qui ont **effectivement** été réalisées et complétées, et non pas celles que vous **auriez voulu ou aimé** développer...

   Notamment, pour cette partie, vous devez fournir la sortie produite par votre application pour la commande `help`.

   (3) Une description (brève) de votre environnement de développement — type de machine et d'OS, éditeur de texte ou IDE, etc.

   (4) Une description de l'architecture de votre application : quelles sont les principales classes et les dépendances entre ces classes. Notamment, un diagramme de composants serait utile — voir les exemples vus en cours.

   (5) Une description (brève) de ce que vous avez testé dans vos tests d'acceptation.[6]

   Si vous avez aussi des tests unitaires — ce que je vous suggère! — indiquez-le.

   (6) Une conclusion, où vous discutez de votre expérience de programmation en Ruby (labos, devoirs 2 et 3) : les problèmes et difficultés rencontrés avec le langage, l'environnement, les choses intéressantes ou nouvelles que vous apprises, etc.

b. Tout le code source que vous aurez développé, y compris les tests.

   Votre «code source» **doit inclure un fichier `Rakefile` approprié**, définissant minimalement les deux cibles suivantes :

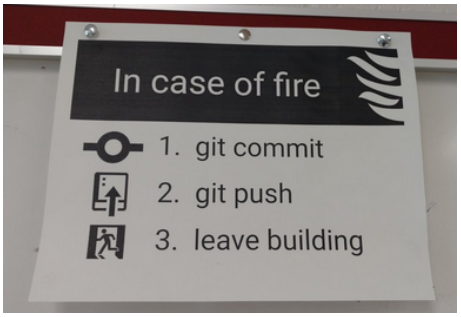
   (1) `exemples` : Lance des exemples d'exécution de votre application — pour «illustrer», de façon *informelle*, le fonctionnement des principales commandes.

   (2) `test_acceptation` : Exécute les «vrais» tests d'acceptation.

   Tel qu'indiqué plus haut, un fichier `.gemspec` doit aussi être présent.

[6] Donc, je veux comprendre, **avant d'aller voir votre code**, ce que vous avez testé comme fonctionnalités (QUOI?), et non pas les détails de comment vous avez procédé pour chacun des tests.

Remise électronique avec Oto et utilisation de `git`



La remise **électronique**, comme pour les devoirs 1 et 2, doit se faire à l'aide d'Oto.

Par exemple, supposons que `Devoir3` est le répertoire contenant les éléments à remettre et `ABCD11111101` est votre code permanent. Alors vous devrez exécuter la commande suivante **sur `java` à partir du répertoire parent de `Devoir3`** :

```
$ ssh oto.labunix.uqam.ca oto rendre_tp tremblay_gu INF600A ABC11111101 $PWD/Devoir3
```

Comme dans les deux premiers devoirs, une (petite) partie des points sera associée à une utilisation appropriée de `git` — donc vous devrez aussi remettre votre répertoire de code sous contrôle de `git`.

Par contre, si vous avez créé un dépôt `git` externe (par ex., `GitLab`, `GitHub`, `bitbucket`), **vous n'avez pas besoin de remettre votre code par l'intermédiaire d'Oto**. Vous devez plutôt : *i*) indiquer dans votre document comment je peux cloner votre dépôt (en m'attribuant si nécessaire les permissions d'accès : voir tableau) et *ii*) me transmettre par courriel (`tremblay.guy@uqam.ca`) une invitation ou l'URL de votre dépôt.

Fournisseur	**Nom d'usager**
`GitHub`	`tremblay-guy`
`GitLab`	`tremblay_guy`
`bitbucket`	`tremblay_g`

Critères de correction

Pour la correction, j'utiliserai **la grille à la dernière page** du présent document.

Je rappelle que tant la qualité de votre architecture que la qualité (style) de votre code — présentation, clarté, respect des principes DRY et KISS[7], structure, choix des identificateurs, **respect du style Ruby**, etc. — seront évaluées. Donc, profitez de vos tests pour faire un peu de nettoyage/*refactoring* de votre code avant de le remettre!

[7] DRY = «*Don't Repeat Yourself*» ; KISS = «*Keep It Simple, Stupid*».

Rappels, remarques et suggestions

Options des commandes : *switch* vs. *flag*

Pour les options, n'oubliez pas la distinction entre *switch* et *flag* :

- Une *switch* est une option sans valeur, donc activée (*on*) ou pas (*off*) ;
- Un *flag* est une option avec une valeur associée, donc avec un argument.

Tant les *switches* que les *flags* peuvent avoir une forme courte — un unique caractère précédé d'un tiret «-» — ou une forme longue — un identificateur complet long (et significatif) précédé de deux tirets «--».

La façon dont la valeur est spécifiée dépend alors de la forme du *flag* :

- Forme courte : avec « » (blanc)
- Forme longue : avec «=»

Voici un exemple avec la commande `grep`, qui sélectionne les lignes d'un fichier qui satisfont un(des) patron(s) :

```
# Sélectionne les lignes qui contiennent "foo"
$ grep foo fich1.txt

# switch: Compte le nombre de lignes qui contiennent la chaine "foo"
$ grep --count foo fich1.txt
$ grep -c foo fich1.txt

# flag: Sélectionne les lignes qui matchent les patrons dans le fichier pat
$ grep --file=pat fich1.txt
$ grep -f pat fich1.txt
```

Traitement de la base de données

Puisque votre application doit traiter des données persistantes, le traitement de la BD devrait ressembler à ce qui est fait dans l'application de gestion des vins du devoir #2. Donc, en gros, le traitement effectué dans le programme principal (`bin/votre-appli`) **pour chaque commande** devrait avoir l'allure suivante :

- On charge la base de données, ce qui a pour effet de créer une structure de données interne, i.e., une collection qui représente le modèle des données — appelée l'entrepôt de données (*repository*).
- L'analyse des options et arguments et l'identification de la commande est faite par l'application (`GLI::App`), qui appelle l'action appropriée que vous avez spécifiée pour la commande.
- L'action effectue les vérifications appropriées sur les options et arguments. Si le traitement est vraiment **simple**, il peut être effectué directement dans le programme principal. Autrement, l'action pour la commande dans le programme principal appelle plutôt une méthode du modèle qui met en œuvre la logique métier requise — méthode définie dans un module approprié dans le répertoire `lib`.
- La méthode appelée retourne un ou des résultats, qui sont possiblement émis par l'application.
- L'exécution du programme se termine en sauvegardant les données de l'entrepôt de données (*repository*) dans la base de données.

Un tel comportement est requis car chaque commande est exécutée, au niveau du *shell*, de façon indépendante. En d'autres mots, il n'y a pas de notion de «session» — on lance le programme en spécifiant la commande et ses arguments (et options)... et c'est terminé (donc ni menu, ni sélection interactive!).

Notez qu'une façon de réaliser le comportement de chargement puis de sauvegarde de la base de données est d'utiliser les méthodes `pre` et `post` dans le fichier définissant les commandes avec `gli` — voir l'application `gv` :

```
pre do |global,command,options,args| ...
  ...
end

post do |global,command,options,args|
  ...
end
```

Traitement des erreurs

En cas d'erreur d'exécution dans votre programme, pour obtenir des informations plus précises et détaillées sur le contexte dans lequel est survenu l'erreur, il faut modifier l'appel à la méthode `on_error` — voir l'application `gv` :

```
on_error do |exception|
  ...
end
```

Qualité du code et style de programmation

Tel qu'indiqué dans la grille de correction, une partie de la note sera consacrée à la qualité du code et une partie **au respect des conventions de style de Ruby** :

- Diapositives sur «Qualité du code et Refactoring»:

  http://www.labunix.uqam.ca/~tremblay_gu/INF600A/Materiel/qualite_et_refactoring.pdf

- Annexe C «Règles de style Ruby» des notes de cours sur Ruby :

  http://www.labunix.uqam.ca/~tremblay_gu/INF600A/Materiel/ruby.pdf

Notamment, je rappelle la convention Ruby quant au choix des identificateurs, à savoir l'utilisation du `CamelCase` vs. `snake_case` :

- `NomDeClasse`
- `NOM_DE_CONSTANTE`
- `nom_de_methode`
- `nom_de_parametre_ou_variable`

Accents

Remarque **importante** concernant le contenu de vos fichiers : **N'utilisez pas d'accents... ni dans les identificateurs, ni dans les chaines, ni dans les commentaires!**

L'encodage des accents crée souvent (!) des problèmes de portabilité. C'est spécialement le cas lorsqu'on passe d'une machine Linux à une machine Mac ou vice-versa — et idem pour Windows! Donc, comme je compte exécuter vos applications sur `java` (Linux CentOS) et/ou Mac...

Architecture de votre application : décomposition en composants (couches)

L'architecture de votre application devrait comporter au moins trois (3) composants (couches) :

- L'interface personne-machine (IPM), définie dans le fichier `bin/votre-appli`. Cette partie effectue l'analyse des commandes et des options, appelle une/des méthodes de la couche métier (du modèle d'affaire), puis affiche (sur `stdout`) les résultats appropriés.

  C'est cette partie qui, notamment, choisit et formate les messages à afficher. (Par exemple, si on décidait d'afficher des messages en anglais plutôt qu'en français, ou d'émettre du HTML plutôt que du texte simple, c'est **uniquement** cette partie qui aurait besoin d'être modifiée.)

- La couche métier (le modèle d'affaires), qui représente et manipule les objets métiers, typiquement, ceux mis en œuvre par une ou des classes appropriées pour modéliser la solution.

  Ces classes et méthodes seront définies dans des fichiers du répertoire `lib/votre-appli`, notamment dans le fichier `lib/votre-appli/votre-appli.rb`.

- La couche qui encapsule les aspects liés à l'accès aux fichiers de données — donc la couche persistance. Typiquement, ceci sera réalisé par un module ou une classe défini dans le répertoire `lib/votre-appli`. Ce sont les méthodes de ce module ou classe qui «connaissent» (encapsulent, dissimulent) les détails d'accès aux fichiers de données — cf. `BDTexte` dans l'exemple `gv`.

Par exemple, si on décidait de rendre disponible une IPM via une application Web plutôt qu'une IPM en lignes de commandes, c'est uniquement la première couche qui aurait besoin d'être modifiée. En outre, cette nouvelle IPM contiendrait **très peu de** code en commun avec l'IPM en ligne de commandes, puisque tous les traitements effectifs sont en fait délégués à la couche métier.

Utilisation de `MiniTest` pour les tests

Par défaut, le code généré par «`gli scaffold`» utilise `Test::Unit`, l'ancien cadre de tests pour Ruby. Pour utiliser plutôt `MiniTest`, vous devez modifier le fichier `test/test_helper.rb` pour qu'il contienne ce qui suit — voir aussi `minised` et `gv` :

```
gem 'minitest'
require 'minitest/autorun'
require 'minitest/spec'
require 'minitest/mock'
require 'open3'
```

Vous devez aussi ajouter la ligne suivante dans le fichier `*.gemspec` :

```
  s.add_development_dependency('minitest')
```

Initialement, un seul test (bidon) est défini : `test/default_test.rb`. Si vous désirez utiliser le style avec assertions, ce fichier peut être modifié comme suit — c'est simplement le nom de la super-classe qu'il faut modifier :

```
require 'test_helper'

class DefaultTest < MiniTest::Test
  ...
end
```

Si vous désirez utiliser le style avec *expectations*, ce fichier doit être modifié comme suit :

```
require 'test_helper'

describe "TestBidon" do
  describe "methode a tester bidon" do
    it "test bidon" do
      (1 + 1).must_equal 2
    end
  end
end
```

Remarque : Comme vous n'utiliserez pas `cucumber` pour les tests d'acceptation, vous pouvez aussi modifier le `Rakefile` et le fichier `*.gemspec` pour **supprimer toutes les références à `cucumber` et à `features`**, de même que vous pouvez supprimer le répertoire `features`.

INF600A-20 : Grille de correction du Devoir 3

Nom	
Prénom	
Code permanent	
Courriel	
Nom	
Prénom	
Code permanent	
Courriel	

Descriptions fonctionnalités, env., architecture, portée des tests, conclusion		10 pts
Qualité de l'architecture — séparation IPM (`gli`) vs. classes de mise en œuvre		10 pts
Qualité générale du code (KISS, DRY)		5 pts
Respect du style Ruby		5 pts
Utilisation appropriée de `git`		5 pts
Preuve de bon fonctionnement : exemples d'exécution (cible `exemples`)		10 pts
Preuve de bon fonctionnement : tests `MiniTest` (cible `test_acceptation`)		5 pts
Qualité du français		5 pts
(Bonus) Difficulté, originalité de l'application		5 pts
Total		55 pts

Note globale	/ 10