Projet # 1
Développement d'une application Ruby en «ligne de commandes» avec tests

MGL7460
Automne 2016

1 Objectif

L'objectif de ce premier projet est de mettre en pratique certaines des pratiques professionelles de développement de logiciels vues dans les premières semaines du cours, notamment, contrôle du code source et tests automatiques. Un autre objectif est de vous familiariser avec l'utilisation d'un langage de script.

2 Ce qu'il faut faire

Vous devez développer, **en Ruby**,[1] une (petite) application en ligne de commandes, application pour laquelle vous devrez avoir des **tests automatiques**, principalement des tests unitaires, mais aussi *quelques* tests d'acceptation. De plus, cette application devra être développée et livrée (voir section «Code source») à l'aide d'un logiciel de contrôle de code source.

3 Caractéristiques de l'application et contraintes

C'est à vous de choisir l'application que vous allez développer. Toutefois, un certain nombre de contraintes devront être respectées :

- L'interface personne-machine de votre application devra être en mode «ligne de commandes».
- Plus spécifiquement, le programme principal — qui analyse les commandes et les options — devra être créé et mis en oeuvre à l'aide du *gem* `gli`.[2]

  Le programme principal devra contenir «le minimum», c'est-à-dire, essentiellement l'interface personne–machine, donc la partie qui effectue l'analyse des commandes et des options, puis qui appelle les méthodes qui font «**le vrai travail**». En d'autres mots, le programme principal (`bin/`*`votre-application`*) devra essentiellement jouer un rôle de répartiteur (*dispatcher*).

[1] Si tous les membres de votre équipe maîtrisent déjà Ruby, langage que nous utiliserons à diverses reprises au cours du trimestre (exemples dans les notes de cours, labos, etc.), alors vous pourrez utiliser un autre langage de script, e.g., Python, Groovy. Vous devrez toutefois en discuter avec moi au préalable.

[2] Donc, créé avec la commande `gli scaffold`, et mis en oeuvre à l'aide du DSL du *gem* `gli`.

- Votre application devra manipuler des **données persistantes**, c'est-à-dire que vous devrez avoir une forme de *base de données*. Toutefois, il pourra s'agir d'une simple base de données textuelles — une série de lignes où chaque ligne contient des valeurs séparées par des délimiteurs (fichier CSV). Vous pourrez aussi expérimenter d'autres formes de bases de donnés textuelles, par exemple :
  - `http://ruby-doc.org/stdlib-2.2.1/libdoc/yaml/rdoc/YAML/Store.html`
  - `http://ruby-doc.org/stdlib-2.2.1/libdoc/pstore/rdoc/PStore.html`

  Ou, si vous le désirez, vous pourrez utiliser toute autre forme de base de données (relationnelle, graphes, NoSQL, etc.).

- Votre application, pour effectuer «le vrai travail» (la logique d'affaire), devra définir **des classes et méthodes**, qui respecteront une hiérarchie de fichiers appropriée — dans un sous-répertoire `lib` — et seront définies à l'intérieur **d'un `module` indépendant portant le nom de votre application**.

  Pour un exemple, voir l'application `mini-sed` dans les notes de cours :[3]

  `http://www.labunix.uqam.ca/~tremblay/MGL7460/Materiel/projets-ruby.pdf`

- Vous devrez définir **des tests unitaires détaillés et complets** pour vos classes et méthodes. Vous pourrez utiliser avec `MiniTest` ou `RSpec`.

  Si vous utiliser `MiniTest`, vous pouvez utiliser le **style** que vous préférez : «à la JUnit» (i.e., en sous-classant la classe `MiniTest::Test` et avec des `assert_*`) ou «à la RSpec» (i.e., avec `describe`, `it` et `must_*`).

- Vous devrez aussi définir **quelques tests d'acceptation**. Ces tests n'ont pas à être aussi complets que vos tests unitaires : il suffira qu'il y ait au moins un (1) test pour chaque commande principale et chaque option principale. Ces tests pourront ne traiter que les «**cas normaux**». En d'autres mots, au niveau des tests d'acceptation, il ne sera pas nécessaire de tester tous les cas anormaux ou erronés, ni de tester toutes les combinaisons de variantes et options.

  Ces tests d'acceptation, comme pour les tests unitaires, devront être écrits avec `MiniTest` ou `RSpec`.[4]

Note : L'objectif premier de ce travail est de montrer que vous maitrisez les principaux éléments et aspects du développement d'une application en Ruby avec un DSL tel que celui de `gli`, y compris (et surtout!) l'aspect des **tests automatiques**. Donc, au niveau «quantitatif», *puisque certains se poseront certainement la question*, votre application devrait comporter (au moins) 4–5 commandes, et au moins une option globale et une commande avec une option locale. Toutefois, des points bonus pourront être attribués pour la difficulté ou l'originalité de l'application...

[3] Notez qu'en créant votre application avec «`scaffold gli`», c'est bien la structure que vous obtiendrez!

[4] Le projet #2 portera plus spécifiquement sur les tests d'acceptation à l'aide d'outils tels que `cucumber`, `jBehave`, `Fit`, etc. Donc, on reviendra sur ce sujet d'ici quelques semaines.

4 Ce qu'il faut remettre

Rapport écrit

Vous devrez remettre un (1) rapport écrit par équipe (≈ 10–12 pages, fichier PDF), rapport contenant les éléments suivants :

1. Une description des principales fonctionnalités **mises en oeuvre par** votre application — donc celles qui ont **effectivement** été réalisées et complétées — et non pas celles que vous **auriez voulu ou aimé** développer...
2. Une description de votre environnement de développement — type de machine et d'OS, éditeur de texte ou IDE, version de Ruby utilisée, etc.
3. Une description de l'architecture de votre application : quelles sont les principales classes et les dépendances entre ces classes.
4. Une brève description de ce que vous avez testé dans vos tests unitaires — donc le niveau de couverture des options, des cas d'erreurs, etc.
5. Une conclusion où vous discutez brièvement de votre expérience de programmation en Ruby : problèmes et difficultés rencontrés avec le langage, l'environnement, choses intéressantes apprises, etc.

La remise du rapport se fait par courriel.

La date limite pour la remise du rapport (**fichier PDF**) : **jeudi 20 octobre, 18h00**.

Code source

Pour le code source, la contrainte est que je puisse obtenir une copie de votre code source à partir d'un dépôt `git`, initialement en clonant votre dépôt, puis en le mettant à jour (`pull`) — **et ce même avant la remise finale**.

J'examinerai l'historique des *commits* pour m'assurer que *vous avez utilisé l'outil de contrôle du code source de façon appropriée* et aussi pour m'assurer que tout n'a pas été fait par une seule personne. En d'autres mots, il ne suffira pas qu'un coéquipier fasse un seul gros `commit`... le jour de la remise ☹

Ma suggestion, pour me simplifier l'accès : Utilisez `GitHub`[5] ou `bitbucket`,[6] deux systèmes pour lesquels je possède un compte usager.

Évidemment, j'essaierai aussi d'exécuter votre application et de lancer les tests. Votre code source doit donc inclure un fichier `Rakefile` approprié, définissant minimalement les deux cibles suivantes :

1. `test` : Exécute les tests unitaires.
2. `test_acceptation` : Exécute les tests d'acceptation.

Si vous avez utilisé des *gems* autres que `gli`, il est important que vous les spécifiez dans le fichier `*.gemspec` — voir plus bas.

[5] `tremblay-guy`
[6] `tremblay_g`

Rapports de participation

- Chaque personne devra compléter et remettre un «Rapport de participation» :

  `http://www.labunix.uqam.ca/~tremblay/MGL7460/Projets/rapport-participation-1.docx`

- Les rapports de participation devront être remis **sous forme papier**.
- Date de remise : jeudi 27 octobre, 18h00 (au début du cours).

5 Utilisation de *gems* additionnels et choix de l'environnement de développement et de tests

Voici certaines contraintes à respecter quant à l'environnement de développement et de tests et quant à la version de Ruby :

a. **Développement et tests sur** `malt.labunix.uqam.ca`

   À moins d'indication contraire de votre part, je testerai le bon fonctionnement de votre application sur la machine `malt.labunix.uqam.ca`, **avec Ruby 2.2.1**.

   Si votre application requiert des *gems* **additionnels**, qui ne sont pas déjà installés sur `malt`, alors vous devrez les indiquer explicitement dans le fichier `.gemspec` de votre application.

   Dans ce cas, vous devrez installer ces *gems* additionnels de façon locale, **dans votre espace personnel** — ce que je ferai aussi pour tester votre application :

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

b. **Développement et tests sur une autre machine que** `malt`

   Si vous choisissez de développer sur une autre machine que `malt`, alors :

   - Il faut **l'indiquer explicitement** dans votre document et dans votre code, en indiquant dans quel environnement vous avez développé (Linux ou Mac OS).
   - **Vous devrez utiliser Ruby MRI** (et non `JRuby`), version ≥ 2.1.
   - Votre fichier `.gemspec` devra être complet quant aux *gems* requis. Pour tester votre application, je commencerai par exécuter `bundle install`, puis je lancerai les tests... si ça ne fonctionne pas, vous aurez un gros problème ☹

Travail remis à Guy Tremblay

MGL7460-40 : Projet # 1
À remettre jeudi 20 octobre 18:00

Nom	
Prénom	
Code permanent	
Nom	
Prénom	
Code permanent	
Nom	
Prénom	
Code permanent	

Descriptions fonctionnalités, architecture, portée des tests, conclusion		10 pts
Qualité de l'architecture — séparation IPM (`gli`) et classes de mise en oeuvre		10 pts
Qualité du code — KISS & DRY + respect des conventions Ruby		10 pts
Preuve de bon fonctionnement — tests unitaires		10 pts
Preuve de bon fonctionnement — tests d'acceptation		5 pts
Qualité du français		5 pts
(Bonus) Difficulté, originalité de l'application		5 pts
Total		50 pts

Note globale		/ 10