Tests unitaires

TP Boost/C++ et Python

Tests unitaires

Eric Ramat
ramat@lil.univ-littoral.fr

26 février 2007

Durée : 6 heures

---

1 Objectif

L'objectif de ce TP est d'écrire un ensemble de fonctions Python et C++ en adoptant la démarche eXtrem Programming. En voici les grandes lignes. Dans ce TP, nous nous intéresserons qu'aux éléments en gras.

1.1 Les pratiques de développement

- produire du code simple à modifier
  - **coder et concevoir avec simplicité**
  - **répondre aux besoins du client (et rien d'autre !)**
  - concevoir peu (l'essentiel) avant de coder
- pratiquer le refactoring
  - restructurer régulièrement le code
  - faire le ménage dans le code (nommage de variables bizarres, fonctions obscures, ...)
  - diviser les longues fonctions en fonctions plus petites
  - décomposer la fonctionnalités en sous-fonctionnalités (de plus en plus simple)
  - factoriser le code commun
  - tendre vers des pattern designs
  - **attention, les tests ne doivent pas être touchés et valider la restructuration par les tests.**
- développer des standards de développement
  - adopter un coding style et rendre public au groupe les règles
  - présenter les règles sous forme d'exemples
  - adopter un système de gestion des sources qui permet de vérifier la conformité du code au style
- développer un vocabulaire commun
  - définir un vocabulaire "imagé" pour parler des composants du logiciel
  - penser à réviser aussi votre vocabulaire

1.2 Les pratiques du développeur

- **adopter le développement par les tests**

  - **écrire des tests qui échouent (pour le code non écrit) avant d'écrire le code ! Ecrire les entêtes des fonctions à tester avant**
  - **écrire le code pour que les tests passent**
  - **lancer les tests et automatiser le**
  - **important : une fonctionnalité est terminée lorsque tous les tests passent !**
  - **choisir un framework de tests (boost : :test en C++ et unittest en Python)**
  - **avant la correction d'un bug, écrire un test qui teste le bug**
  - **écrire des tests d'acceptation avec le client (si tous ces tests passent alors le travail est fini)**
- programmer en binôme
  - les binômes sont définis pour une tâche ou une courte période :
    - un pilote, celui qui réalise la tâche,
    - un navigateur, celui qui s'assure que la tâche est accomplie dans le respect des règles du projet (vérification du code, ...)
  - les rôles peuvent changer
  - collaborer pour trouver la meilleure solution
  - diffuser vos connaissances dans l'équipe et apprendre des autres
- adopter la propriété collective du code
  - tout développeur a le droit de modifier toute partie du code
  - si une partie du code peut être amélioré, améliorer le
  - le code doit être lisible pour tous
- intégrer continuellement
  - maintenir un dépôt du logiciel stable (toutes les parties ont passés les test avec succès)
  - mettre à jour fréquemment votre version locale du code
  - intégrer souvent votre code (à chaque petite modification, ajout,..., liés à une et une seule fonctionnalité)
  - attention à la gestion de l'intégration, dédier une machine pour cela
  - le code inachevé (non testé) doit être jeté en fin de scéance

2 Travail

Le développement doit se faire au fur et à mesure des besoins du client et on doit se limiter **strictement** à ces besoins. Pour le premier scénario, Python sera le langage de développement. Pour le deuxième scénario, le code est à écrire en C++ et on utilisera la STL et boost.

Comme tout le monde le sait les besoins du client évoluent constamment, nous allons donc présenter ces besoins sous forme d'un scénario en plusieurs étapes. Chaque étape doit donner naissance à des tests et au code répondant aux tests.

2.1 Scénario $n^o 1$

Le client vient faire des calculs simples et en langage naturel. Attention, on ne se contentera tout de même pas à un simple switch. Le client (c'est à dire le prof) se garde le droit d'allonger la liste des tests.

2.1.1 Première histoire

Le client aimerait un programme qui donne "deux" lorsque l'on met en entrée du programme "un plus un".

2.1.2 Deuxième histoire

Le client étant de plus en plus exigent, il aimerait que “un plus deux” donne “trois” et que “trois plus quatre” donne “sept”.

2.1.3 Troisième histoire

L'addition étant disponible, le client demande la multiplication et aimerait que les 2 exemples suivants fonctionnent :

- “deux fois trois” donne “six” ;
- “trois fois trois” donne “neuf”.

2.1.4 Quatrième histoire

Après l'addition et la multiplication, le client désire combiner l'ensemble des opérateurs dans des expressions plus longues. En voici un exemple : “deux fois trois plus un” donne “sept”.

2.1.5 Cinquième histoire

Le client donne trois nouveaux exemples :

- “deux plus trois fois trois” donne “onze” ;
- “cinq fois trois fois deux” donne “trente”.
- “quatre plus six fois cinq” donne “trente quatre” ;

2.2 Scénario $n^o 2$

Cette fois, le client est un autre développeur de la même équipe que vous et l'objectif de l'équipe est d'écrire une application de dessins. Vous allez vous intéresser à la partie stockage des objets géométriques et l'ensemble des fonctions à développer se situe dans la classe kernel : :storage (kernel étant le nom du namespace).

2.2.1 Première histoire

Le client désire une fonction add_object dont les paramètres sont :

- l'identifiant de l'objet ;
- le type de l'objet ;
- les caractéristiques de l'objet.

Les trois paramètres sont des chaînes de caractères. Dans un premier temps, le client propose deux exemples afin de valider la fonction :

- add_object(“id1”,”circle”,”x=10 ;y=10 ;r=3”) ;
- add_object(“id2”,”rectangle”,”x=3 ;y=5 ;h=5 ;w=8”).

Le client désire tester si les objets dont les identifiants sont “id1” et “id2” sont bien présents dans l'instance de storage. Il va de soi qu'il faut développer une nouvelle fonction du type exist_object(“id1”) pour réaliser les tests.

2.2.2 Deuxième histoire

Comme dans toute application graphique, on peut déplacer les objets. Le client aimerait directement modifier les caractéristiques des objets à l'aide d'une nouvelle fonction move_object. En voici un exemple d'utilisation : move_object(“id2”,5,10).

Le client indique que la valeur 5 correspond au déplacement en x et 10 en y.

Le client désire tester si les nouvelles coordonnées de l'objet d'indentifiant "id1" sont bien 8 et 15.

2.2.3 Troisième histoire

Afin de faciliter la manipulation d'un ensemble d'objets, la fonction group serait la bienvenue. Cette fonction a pour objectif de créer des groupes d'objets géométriques.

Voici un exemple : group("id3","id1 :id2")

Le premier paramètre est l'identifiant du groupe et le deuxième la liste des objets constituants le groupe.

2.2.4 Quatrième histoire

A ce stade du développement, le client se demande ce qu'il se passe dans les situations suivantes :

- première situation : add_object("id1","circle","x=0 ;y=0 ;r=-5") ; on s'attend à ce que l'objet n'appartienne pas à l'instance de storage ;
- deuxième situation : s'il fait cette séquence d'appels :

```
add_object("id1","circle","x=10;y=10;r=3");
add_object("id2","rectangle","x=3;y=5;h=5;w=8");
group("id2","id1:id2");
```

  le groupe n'est pas crée ;
- troisième situation : move_object("id1",5,8) ; une exception est levée, l'objet n'existe pas.

2.2.5 Cinquième histoire

Le client aimerait déplacer un groupe donc tous les objets géométriques appartenant au groupe.

Voici un exemple :

```
add_object("id1","circle","x=10;y=10;r=3");
add_object("id2","rectangle","x=3;y=5;h=5;w=8");
add_object("id3","circle","x=-5,y=-10,r=7");
group("id4","id1:id2:id3");
move_object("id4",-3,0)
```

Le client désire tester les coordonnées de tous les objets.

2.2.6 Sixième histoire

Le client aimerait créer des groupes dont les éléments le constituants peuvent être des groupes. Voici un exemple :

```
add_object("id1","circle","x=10;y=10;r=3");
add_object("id2","rectangle","x=3;y=5;h=5;w=8");
group("id3","id1:id2");
add_object("id4","circle","x=-5,y=-10,r=7");
group("id5","id3:id4");
```

Le client désire tester si le groupe d'identifiant id5 est crée et s'il possède bien 2 objets. Le client veut en profiter pour tester si le nombre d'objets est bien égal à 5.

2.2.7 Septième histoire

Le client désire pouvoir supprimer des objets avec la fonction delete_object. Voici un exemple :

```
add_object("id1","circle","x=10;y=10;r=3");
add_object("id2","rectangle","x=3;y=5;h=5;w=8");
delete_object("id1");
```

Le client teste si le nombre d'objets est égal à 1 et si l'objet d'identifiant id1 n'existe plus.

2.2.8 Huitième histoire

Le client ajoute l'exemple suivant :

```
add_object("id1","circle","x=10;y=10;r=3");
add_object("id2","rectangle","x=3;y=5;h=5;w=8");
group("id3","id1:id2");
add_object("id4","circle","x=-5,y=-10,r=7");
group("id5","id3:id4");
delete_object("id2");
```

Le client fait les tests suivants :
- le nombre d'objets du groupe id3 est égal à 1 ;
- le nombre total d'objets est 4 ;
- l'objet id2 n'existe plus.

3 Références

- Boost : http://www.boost.org
- STL : http://www.sgi.com/tech/stl/
- Tests unitaires avec PyUnit : http ://pyunit.sourceforge.net/pyunit.html