

TP synthèse

Python/Gtk, design patterns, tests unitaires et XML

Eric Ramat

ramat@lil.univ-littoral.fr

12 septembre 2007

Durée : 7,5 heures

1 Objectifs

L'objectif de ce TP est d'écrire un ensemble de classes et méthodes Python 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 séance

2 Le sujet

Nous allons nous intéresser à Invers. Vous pouvez jeter un coup d'oeil sur les pages suivantes pour connaître les règles du jeu : <http://jeuxsoc.free.fr/index2.php?principal=/i/inver.htm¶m=> et http://jeuxdeplateau.com/article.php?id_article=22.

Un programme Python, `server.py`, a été écrit et permet de synchroniser deux joueurs virtuels ou non. Dans la suite du sujet, nous appellerons ce programme l'arbitre.

L'arbitre a pour objectif de recevoir, par socket, les coups joués et d'envoyer les messages de synchronisation aux joueurs. Le protocole est le suivant :

- le serveur ouvre deux sockets, l'une sur le port 8000 pour le joueur vert et l'autre sur le port 8001 pour le joueur rouge ;
- le serveur se mets en attente d'une connexion sur chacune des sockets ;
- les clients se connectent (création d'une socket et connexion sur les ports 8000 et 8001) ;
- les clients doivent ensuite envoyé la trame XML `<start />` pour indiquer qu'ils sont prêts à démarrer ;
- quand les deux joueurs sont prêts (connexion + envoi de `<start />`) alors le serveur envoie alors `<go />` au joueur vert (celui connecté sur le port 8000) ;
- le client élabore son coup et créé une trame XML formatée de la manière suivante : `<play number='x' position='y' colour='/'>` où `number` est le numéro de la ligne ou de la colonne où l'on

veut placer sa tuile et position a pour valeur Left, Right, Bottom ou Top, l'endroit où est placé la tuile et colour est la couleur de la tuile placée (Green ou Red) ; lorsque la tuile est placée à gauche (Left) alors les tuiles sont décalées vers la droite et la tuile la plus à droite est retirée du jeu ;

- le serveur répond au joueur qui vient de jouer par une trame : * <win /> ; * <lost /> ; * <invalid /> pour dire Coup invalide ; * <green /> pour dire Une tuile verte (Green) est sortie du jeu ; * <red /> pour dire Une tuile rouge (Red) est sortie du jeu ;
- le serveur envoie une trame XML au joueur qui n'a pas joué ; cette trame est composée de la trame envoyée par le joueur qui a joué à laquelle on ajoute un attribut qui indique la couleur de la tuile sortie (out='Green|Red') ; si le coup n'était pas valide, l'autre joueur reçoit <invalid /> ; s'il a gagné ou perdu, il reçoit <win /> ou <lost /> comme l'autre joueur ;
- le serveur envoie ensuite le caractère <go /> à l'autre joueur pour qu'il propose son coup ;

L'arbitre vérifie à chaque coup, si l'un des joueurs n'a pas triché. En effet, on ne peut jouer qu'avec les tuiles sorties. Il faut donc que le programme 'joueur' gère la couleur des tuiles hors du jeu. Autre indication : les lignes et les colonnes sont numérotées de 0 à 5.

3 Travail

Le développement doit se faire au fur et à mesure des besoins du client et on doit se limiter **strictement** à ces besoins.

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. Attention, le client n'est pas très sûr de ce qu'il vous livre pour les tests.

3.1 Première histoire

Une première version de l'arbitre a été écrite. Lancez le, connectez vous dessus avec deux programmes Python, les deux programmes envoient la trame <start /> et le programme du joueur vert attend la trame <go />. Si le programme du joueur vert reçoit la trame <go />, le premier test est validé.

Indications :

- pour la programmation des sockets en Python, voir section références ;
- le flux de communication des joueurs doit être encapsulé et un Proxy serait le bienvenu. Le client pourrait avoir envie de changer d'avis sur le mode de communication.

3.2 Deuxième histoire

Le joueur vert envoie la trame <play number='3' position='Top' colour='Green' /> et attend que l'arbitre lui réponde par la trame <green />.

3.3 Troisième histoire

Le joueur rouge reçoit la trame <play number='3' position='Top' colour='Green' out='Green' /> et la trame <go />.

3.4 Quatrième histoire

Le joueur rouge envoie la trame <play number='2' position='Top' colour='Red' /> et il reçoit en réponse la trame </invalid >.

3.5 Cinquième histoire

Le client désire aller plus loin. Pour tester son arbitre, il veut lui envoyer des trames “aléatoires” même invalides. Dans un temps fini, les joueurs doivent recevoir respectivement la trame <win /> et <lost />. Attention, pour un joueur, le génération aléatoire doit être unique (pensez aux patterns).

3.6 Sixième histoire

Un joueur aléatoire c'est bien mais le client aimerait disposer d'une petite interface graphique comme l'arbitre, pour désigner ces coups.

Cette interface doit permettre de :

- visualiser l'état du jeu ;
- indiquer la ligne ou la colonne et le côté où la tuile doit être insérée ;
- connaître la couleur de la tuile hors du jeu ;
- visualiser l'historique des coups.

Une mise en facteur du code de l'arbitre et de l'interface graphique serait le bienvenu.

Indications :

- un petit MVC est à envisager ;
- pour la programmation de l'interface graphique, voir le code de l'arbitre et les références.

3.7 Septième histoire

Le client désire changer sa gestion du flux entre les programmes. Il désire utiliser : XML-RPC. Ce protocole permet d'attaquer directement les modèles d'un programme distant sous forme d'un proxy. Les trames XML définies précédemment disparaissent et sont remplacées par l'invocation de méthodes distantes.

Adaptez votre code. L'ensemble des tests doivent être aussi adapté.

4 Références

- Python : <http://www.python.org>
- Les sockets en Python : <http://docs.python.org/lib/socket-example.html>
- XML-RPC et Python : <http://docs.python.org/lib/module-xmlrpclib.html> et <http://www.xmlrpc.com/>
- Tests unitaires avec PyUnit : <http://pyunit.sourceforge.net/pyunit.html>
- PyGTK : <http://www.pygtk.org>
- Design Patterns : <http://home.earthlink.net/~huston2/dp/patterns.html>