The REBOL Documentation Project

-- FR - Documentation REBOL - Divers --

Divers

Programmation Pragmatique : REBOL et eXtreme Programming - Extrait

REBOLtof

Première publication : 24 mars 2006, et mis en ligne le vendredi 24 mars 2006

Résumé :

J'ai le plaisir de vous présenter ici un extrait d'un ouvrage en cours de rédaction. Il s'agit d'études de cas pratiques d'application de la méthodologie d'ingénierie logicielle "eXtreme Programming" à la programmation avec REBOL.

L'extrait présenté porte sur le deuxième jour de développement d'une application de gestion d'une vidéothèque.

Parcourant d'anciens numéros de la revue en ligne anglophone "IEEE Software", mon attention est retenue par un article datant de quelques années, où il est question de la "compréhension de la sociologie de projets par la modélisation des acteurs" [Alexander]. Les auteurs décrivent les réactions de développeurs à l'égard de leurs clients. Une des - remarquables - réponses était "ils ne disposent pas des compétences nécessaires afin de participer à l'établissement des objectifs et des contraintes. Ils décrivent des solutions plutôt que des spécifications techniques, et *ils changent d'avis*."

Ce cas illustre une situation tellement présente dans le monde du développement logiciel : celle du développeur frustré par l'attitude du client. Et pourtant, dans le monde réel, le client est intéressé par des solutions à son problème, pas par des spécifications abstraites et centrées sur le développeur. Et le client a le droit de pouvoir changer d'avis. Et peut-être n'avait-il pas bien saisi le problème. Ou les conditions qui avaient mené à sa décision ont changé...

Tout développeur un peu expérimenté dans ce domaine a un jour été confronté à ce type de scenario...

N'hésitez pas à nous faire part de votre avis en utilisant le formulaire en-dessous de l'article

Jour 2

Toute l'équipe est réunie. Je rappelle les règles du jeu. Tous les jours à 10 heures, nous commençons par une réunion de coordination (appelée "Stand-up meeting" dans le jargon XP). Tous les emails et les téléphones doivent donc être traités avant cette heure. Chaque membre l'équipe décrit brièvement le travail effectué le jour précédent et celui planifié pour le reste de la journée. Différents petits problèmes techniques peuvent y être discutés, ainsi que toute information intéressant l'équipe. Cela aide chacun des développeurs à cadrer son travail dans celui du groupe, et chacun à l'occasion d'aider les autres. Le Stand-up meeting sert également de point de départ pour la journée de travail. Les paires de programmeurs se forment, et de nouvelles tâches sont choisies par les binômes...

Je rappelle à l'équipe que nous utilisons un autocollant vert sur une tâche quand elle est complète, et qu'un autocollant vert sur une story signifie que toutes ses tâches sont complètes. Sinon, il faut employer du rouge pour la tâche et sa story. Toutes les cartes, sauf les cartes de tâches utilisées par les binômes, sont affichées sur nos tableaux métalliques blanc qui occupent tout un pan de mur. Ainsi tout le monde, même le client, peut facilement suivre l'état d'avancement du projet.

- Pas de questions pour le moment ? Non ? Alors, choisissez votre partenaire... Nous n'avons pas encore eu de contact avec le client, donc nous n'avons pas de détail sur les priorités des stories... Alors, lisez ce document de specs que nous avons reçu, et tentez quelques "spikes" de code pour fixer les idées.

Les "spikes" sont des essais libres de petits bouts de code, destinés à vérifier la faisabilité d'une solution technique avant son intégration dans le code en développement.

Benoît fait un signe de main à Fabrice. Julie sourit à Martin... C'est parti.

- Quand auront-nous un contact avec le client ? s'inquiète Julie.

- Je dois encore aller aux nouvelles... J'y vais d'ailleurs de ce pas ! répond-je

Je me dirige vers le bureau de Bertrand, qui s'occupe des premiers contacts et de l'administration avec le client.

- Bertrand, qu'en est-il avec le client de Video4all ? Je n'arrive pas à le contacter...

- Euh... Oui... J'ai eu quelques conversations avec lui. Je peux dire qu'il a été fort déçu par la première firme à qui il avait d'abord confié le travail. Il se méfie maintenant, et depuis qu'il nous a livré les spécifications et signé le contrat, il ne nous a plus contacté.

- Bon, je vais continuer avec les éléments que j'ai. J'essayerai encore de le contacter.

De retour dans notre espace de travail, Fabrice me signale que Benoît a dû s'absenter d'urgence, pour régler un problème avec un autre projet. Pas de problème. Je me mets en binôme avec lui. Je vais en profiter pour lui rappeler notre utilisation du framework de tests unitaires.

- Vois-tu Fabrice, XP part du principe "les tests en premier". Pour cela, on écrit d'abord le test unitaire destiner à contrôler une fonctionnalité, et ensuite, le code nécessaire pour faire passer le test.

- Comment écrire le test si je ne connais pas la fonctionnalité ? Je vais écrire n'importe quoi alors ?

- Justement : tu connais la fonctionnalité, ou au moins, tu connais les résultats attendus. Donc tu peux écrire le test ! Regarde, je vais prendre un exemple simple. Imaginons que tu doives implémenter la fonctionnalité de "rajouter 1 au nombre passé en argument". Tu sais à ce stade-ci que si tu passes "1" à la fonction, elle te retourne ...

- "2" bien sûr !

- Exactement. Et si tu passes "10" ?

- "11"...

- Voilà. Tu disposes de suffisamment d'éléments pour écrire tes premiers tests ! Vérifions que nous avons les utilitaires nécessaires sur la machine.

Je contrôle sur le PC la présence d'un éditeur de texte, et de l'évaluateur de REBOL, et du framework de test "RUn.r". Je lance l'éditeur, crée un nouveau fichier, que je sauve sous le nom "spike-ajoute.test". Nous avons pris la bonne habitude de faire figurer le mot "spike" dans le nom de tous les fichiers différents de ceux contenant le code de production, ce qui nous évite les erreurs quand nous "nettoyons" nos répertoires. Je tape le code suivant :

```
test-ajoute-1: does []
```

qui constituera l'ossature de notre premier test.

Je crée un second fichier, qui sera destiné au script contenant les fonctionnalités à tester. Je le sauve sous le nom de "spike-ajoute.r". Je me contente d'un rajouter un en-tête REBOL :

```
rebol []
```

Revenant à mon fichier de test, je place en début de fichier le chargement de mon script :

```
do %spike-ajoute.r
```

Et dans mon premier test, je place un appel à la fonction à créer :

```
test-ajoute-1: does [assert-equal ajoute 1 2]
```

- Voilà Fabrice. C'est ainsi que l'on crée son premier test. Maintenant, j'ai écrit littéralement : "j'affirme qu'en passant 1 comme argument à la fonction 'ajoute, la réponse sera 2".

- Je vois...

- Lançons maintenant notre test...

J'ouvre la console REBOL, et j'y charge le script de RUn :

```
>> do %RUn.r
Script: "(R)EBOL (Un)it" (none)
>>
```

Puis je lance mon test :

```
>> run-test %spike-ajoute.test
```

La réponse est immédiate :

```
==================
= (R)EBOL-(Un)it =
= Test Framework =
==================
== OUTPUT ================================================
> spike-ajoute.test
>> SETUP
Script: "Untitled" (none)
    --> Done
>> test-ajoute-1
    !-> ERROR generated:
        Script Error: ajoute has no value
        Near : assert-equal ajoute 1 2
        Where: test-ajoute-1
== TEST REPORT ===========================================
TOTAL : 1
   => Passed   : 0
```

```
    => Failures : 0
    => Errors   : 1
 ============================================================
 End of test
 Press [ENTER] to Quit
```

La réaction de Fabrice n'est pas moins rapide :

- Hé mais ça ne marche pas ton truc !

- L'erreur que tu vois prouve justement que mon "truc" fonctionne, lui répond-je en souriant devant sa réaction attendue. RUn me signale simplement que la fonction 'ajoute est inconnue. Ce qui est normal, puisque nous ne l'avons pas encore définie dans notre script.

- Ah... Alors il fallait d'abord la définir avant de lancer le test, non ?

- Et non... Car de cette manière nous pouvons une première fois vérifier le bon fonctionnement du framework de test dans notre contexte d'utilisation.

- Hu ... ?

- Oui : maintenant, une erreur s'est déclenchée. Ce qui est le comportement attendu, puisque nous n'avons pas encore définit la fonction. Imaginons maintenant que cette erreur ne se soit pas déclenchée... Quelle conclusion pouvons nous alors tirer sur la qualité de nos tests et donc de notre code ? Nous n'aurions aucune certitude...

- Okay. Mais dans quel cas cette erreur pourrait-elle ne pas se déclencher ?

- Il m'est par exemple déjà arrivé de créer une fonction et de la nommer identiquement qu'une autre fonction créée précédemment. Lors du test, je n'ai pas eu d'erreur, mais un test qui ne passait pas. Cela m'a permis d'éviter d'emblée un bug particulièrement vicieux : deux fonctions aux noms identiques...

- Compris !

- Bon continuons...

J'implémente à présent le squelette le la fonction 'ajoute dans mon script, qui devient :

```
rebol []
ajoute: func [nombre] [true]
```

Et je montre à Fabrice :

- Pour la même raison que précédemment, je n'implémente pas encore la fonctionnalité. Je vais simplement tester si mon framework de test peut accéder à la définition de la fonction. Comme un résultat est attendu, je me contente de placer un 'true dans le corps de la fonction. Comment penses-tu que RUn va réagir ?

- Je suppose... Un test qui ne passe pas ?

- Correct : regarde...

Je repasse dans ma console. La touche "flèche vers le haut" me permet de remonter dans l'historique, et de retrouver ma dernière commande.

```
>> run-test %spike-ajoute.test
```

Comme prévu, le test échoue :

```
==================
= (R)EBOL-(Un)it =
= Test Framework =
==================
== OUTPUT ================================================
> spike-ajoute.test
>> SETUP
    --> Done
>> test-ajoute-1
    *-> Data expected: 2 of type: [integer!]
        but was : true of type: [logic!]
== TEST REPORT ===========================================
TOTAL : 1
   => Passed   : 0
   => Failures : 1
   => Errors   : 0
==========================================================
End of test
Press [ENTER] to Quit
```

- Maintenant que nous sommes certain que le framework de test fonctionne et est en mesure d'accéder à la fonction, nous avons déjà éliminé deux sources de biais potentielles ! Continuons notre travail... Nous allons maintenant implémenter la fonctionnalité minimum nécessaire afin de passer le test. Reprenons notre script et écrivons le corps de la fonction.

```
rebol []
ajoute: func [nombre] [nombre + 1]
```

- Et on repasse le test, dit Fabrice.

- Exactement ! lui répond-je. Et je relance le test :

```
==================
= (R)EBOL-(Un)it =
= Test Framework =
==================
== OUTPUT ================================================
> spike-ajoute-prerebol.test
>> SETUP
    --> Done
```

```
>> test-ajoute-1
    --> Passed in
== TEST REPORT ========================================
TOTAL : 1
   => Passed   : 1
   => Failures : 0
   => Errors   : 0
========================================================
End of test
Press [ENTER] to Quit
```

- Et voilà, s'exclame Fabrice. Travail terminé !

Je le corrige :

- Pas exactement terminé ! N'oublions pas le cycle du développement mené par les tests : créer le test, coder, tester, ... refactorer, retester, et continuer jusqu'à ce que le code donne satisfaction !

- Le code est simple, il n'y a quand même rien à refactorer ?

- Et si ! La documentation du code fait partie de la phase de refactoring ... Comme il est toujours possible d'involontairement changer le code lors de l'écriture des commentaires, il convient encore de retester le code. Rajoutons donc notre documentation, et une petite indentation des blocs :

```
rebol []

ajoute: func [
    "ajoute un au nombre passé en argument"
    nombre
] [
    nombre + 1
]
```

- Et retestons.

Le test passe sans problème...

- Que pouvons-nous encore améliorer dans notre fonction ?

Fabrice réfléchit quelques instants et répond.

- Que ce passerait-il si nous passions une string ! en argument ? Cela causerait une erreur ... Nous devrions préciser le type d'argument attendu !

- Correct ! Vas-y, prend le volant ...

- Le volant ?...

- Oui, c'est un idiome propre à XP. On se représente souvent le binôme comme le conducteur - celui qui est derrière le clavier - et le navigateur - celui qui tient la carte.

- Bon, je conduit, dit-il en souriant. Je limite donc le type d'argument valide aux integer !...

```
rebol []

ajoute: func [
    "ajoute un au nombre passé en argument"
    nombre [integer!]
] [
    nombre + 1
]
```

- Et maintenant, retestons...

Le test passe toujours.

- Nous avons rajouté une contrainte au code. Nous devons maintenant rajouter un test qui va contrôler cette contrainte, lui précise-je. Quel test pourrais-tu ajouter ?

- Eh bien... un test où je passe autre chose qu'un integer ! en argument...

- Vas-y !

Fabrice ajoute un test :

```
test-ajoute-2: does [assert-equal ajoute 1.1 2.1]
```

Et relance RUn...

```
==================
= (R)EBOL-(Un)it =
= Test Framework =
==================
== OUTPUT ================================================
> spike-ajoute.test
>> SETUP
Script: "Untitled" (none)
    --> Done
>> test-ajoute-1
    --> Passed in
>> test-ajoute-2
    !-> ERROR generated:
        Script Error: ajoute expected nombre argument of type: integer
        Near : assert-equal ajoute 1.1 2.1
        Where: test-ajoute-2
== TEST REPORT ===========================================
TOTAL : 2
   => Passed   : 1
   => Failures : 0
   => Errors   : 1
```

```
============================================================
End of test
Press [ENTER] to Quit
```

- L'erreur attendue est déclenchée ! C'est ce qu'il fallait faire, non ? s'inquiète Fabrice.

- Tout à fait ! L'erreur attendue est là. Mais tu conviendras avec moi que RUn n'est pas parlant dans ce cas-ci... RUn dispose d'autres fonctions intéressantes, entre autres une qui permet d'intercepter l'erreur, et de la considérer comme un résultat attendu : 'assert-error. Modifions donc notre test de la manière suivante.

Reprenant le clavier, je modifie le test 2.

```
test-ajoute-2: does [assert-error [ajoute 1.1]]
```

- Remarque qu'il est ici nécessaire de placer le code à évaluer à l'intérieur d'un bloc. Cela sert à RUn afin d'intercepter et d'évaluer l'erreur.

Je lance le test...

```
>> run-test spike-ajoute.test
```

Et nous obtenons :

```
==================
= (R)EBOL-(Un)it =
= Test Framework =
==================
== OUTPUT ================================================
> spike-ajoute.test
>> SETUP
Script: "Untitled" (none)
    --> Done
>> test-ajoute-1
    --> Passed in
>> test-ajoute-2
    --> Passed in
== TEST REPORT ===========================================
TOTAL : 2
   => Passed   : 2
   => Failures : 0
   => Errors   : 0
============================================================
End of test
Press [ENTER] to Quit
```

- Et voilà, dis-je. L'erreur attendue est considérée comme le comportement normal de la fonction, et en résulte un test qui passe ...

- C'est chouette, di Fabrice. Et il ajoute avec un petit sourire : mais là tu ne contrôle pas le contenu

de l'erreur ! Elle pourrait être déclenchée par n'importe quoi, comme par exemple une division par 0 ...

- Finement observé ! Mais le concepteur de RUn y a pensé : tu peux préciser en raffinement (les *refinements* sont des spécialisation de fonction en REBOL) à la fonction 'assert-error, le type d'erreur et l'identifiant de celle-ci. Or nous savons que dans ce cas-ci, il s'agit d'une erreur... je regarde dans la documentation de REBOL... une erreur de type 'script et de id 'expect-arg ! Modifions notre deuxième test...

```
test-ajoute-2: does [assert-error/type-id [ajoute 1.1] [script expect-arg]]
```

- Je m'explique. Je rajoute un raffinement à la fonction : /type-id. Et je passe à la fonction un bloc contenant, dans l'ordre le type et l'identifiant (id) de l'erreur. Voyons le résultat en lançant le test :

```
==================
= (R)EBOL-(Un)it =
= Test Framework =
==================
== OUTPUT ================================================
> spike-ajoute.test
>> SETUP
Script: "Untitled" (none)
    --> Done
>> test-ajoute-1
    --> Passed in
>> test-ajoute-2
    --> Passed in
true
== TEST REPORT ===========================================
TOTAL : 2
   => Passed   : 2
   => Failures : 0
   => Errors   : 0
==========================================================
End of test
Press [ENTER] to Quit
```

- Le test passe ! Et au contraire, préciser un type ou un identifiant d'erreur différent fait échouer ton test. Modifions notre test :

```
test-ajoute-2: does [assert-error/type-id [ajoute 1.1] [script missing]]
```

Et le test échoue :

```
==================
= (R)EBOL-(Un)it =
= Test Framework =
==================
== OUTPUT ================================================
> spike-ajoute.test
```

```
>> SETUP
Script: "Untitled" (none)
    --> Done
>> test-ajoute-1
    --> Passed in
>> test-ajoute-2
    *-> Assertion Failed
false
== TEST REPORT ========================================
TOTAL : 2
   => Passed   : 1
   => Failures : 1
   => Errors   : 0
========================================================
End of test
Press [ENTER] to Quit
```

- Ouf, merci professeur, soupire Fabrice. Je comprends à présent beaucoup mieux l'utilité des tests...

Je souris.

- Et tu n'as eu qu'un aperçu d'une partie de leur puissance ! J'aurai encore l'occasion de te parler de tests de non régression, suite de tests ou déboguage... Mais il est déjà tard. Les autres sont déjà partis, et une des pratiques de XP est le "rythme soutenable" ! Donc... à demain.

- Oui, à demain !

Ce fut une journée enrichissante. J'aime expliquer XP à ceux qui disposent d'un esprit ouvert pour le comprendre.