Tester ses programmes

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 4 septembre 2006. Dernière mise à jour le 7 février 2008

https://www.bortzmeyer.org/test-logiciel.html

Peu de programmeurs exécutent des tests systématiques de leurs programmes. Le bon comportement d'un nouveau logiciel est en général déterminé par quelques essais manuels rapides. Pourtant, l'utilisation de tests systématiques permettrait une bien meilleur qualité du programme, surtout en cas de modification.

Ces programmeurs disent en général "Le programme fonctionne, je n'ai pas le temps de faire mieux". Pourtant, des tests systématiques et automatisables permettent de :

— Vérifier rapidement si le programme s'exécute correctement sur une nouvelle plate-forme, où il vient d'être porté,
— Vérifier rapidement si le programme est toujours correct, après une modification qui vient de lui être apporté par une personne qui n'est pas l'auteur,
— Documenter le comportement normal du programme : il est souvent plus facile de lire les jeux de tests que la documentation. **Un jeu de tests est une spécification exécutable.**

Ceci implique que les tests soient automatisables. On doit pouvoir les exécuter avec une seule commande et le résultat doit être binaire : le test a réussi ou pas. Pas question de demander aux testeurs de lire dix pages d'affichage verbeux dont la conclusion n'est pas évidente.

Ceci implique aussi que l'écriture des tests soit facile : comme le test n'apporte pas de fonction supplémentaire au programme, son écriture doit être le plus simple possible.

L'idée de bibliothèque facilitant le développement et l'exécution de jeux de tests a été popularisée par le système JUnit pour le langage Java. Mais d'autres systèmes existent, par exemple toute bibliothèque du dépôt CPAN (pour le langage Perl) dispose d'un jeu de tests et la personne qui installe ou modifie une bibliothèque de la CPAN peut, juste en tapant `make test` savoir si tout s'est bien passé :

```
% make test
PERL_DL_NONLAZY=1 /usr/bin/perl "-Iblib/lib" "-Iblib/arch" test.pl
1..14
ok 1
ok 2
ok 3
ok 4
ok 5
ok 6
ok 7
ok 8
ok 9
ok 10
ok 11
ok 12
ok 13
ok 14
```

Ici, tout s'est bien passé, les quatorze tests se sont déroulés sans problème. Si je fais une modification et que j'introduis une bogue :

```
% make test
PERL_DL_NONLAZY=1 /usr/bin/perl "-Iblib/lib" "-Iblib/arch" test.pl
1..14
ok 1
ok 2
ok 3
ok 4
ok 5
ok 6
ok 7
ok 8
ok 9
not ok 10
not ok 11
not ok 12
not ok 13
not ok 14
```

L'erreur est immédiatement visible.

D'autres langages ont un système analogue à JUnit. Par exemple, Python a PyUnit <`http://pyunit.sourceforge.net/`>. Un test écrit en PyUnit ressemble à :

```
    def testgetnameservers(self):
        for mydomain_name in existing_domains:
            mydomain = domain.Domain(mydomain_name)
            ns = mydomain.get_nameservers()
            self.assert_(len(ns) >= 2)
```

Les fonctions appelées par PyUnit ont un nom qui commence par "test". Elles comprennent une assertion, ici que le nombre (`len` = "*length*", longueur du tableau) de serveurs de noms est d'au moins deux. PyUnit va exécuter ce code, tester l'assertion et afficher un succès ou un échec en fonction du nombre de tests qui échouent.

PyUnit permet également de tester qu'une fonction qui doit échouer échoue réellement, par exemple en levant bien l'exception attendue si on lui passe certains paramètres (un bon jeu de tests ne teste pas que les cas qui doivent marcher mais aussi ceux qui doivent, d'après la spécification, échouer).

Il est recommandé de nomme ses tests d'une manière qui expliite ce qu'ils testent. Ici, ce code PyUnit fait référence à une bogue, enregistrée dans un système de gestion de bogues (comme Bugzilla) sous le numéro 113 :

```
    def test_bug113(self):
        # Code allow several admin contacts or allow to delete them all :-(
        mydomain_name = "nic.%s" % config.tld
        mydomain = domain.Domain(mydomain_name)
        mycontact = contact.Contact(existing_contact)
        myoldcontact = contact.Contact(existing_old_contact)
        self.assertRaises(domain.InvalidAction, mydomain.manage, mycontact,
                          "add", "admin")
        self.assertRaises(domain.NoSuchRecord, mydomain.manage, mycontact,
                          "del", "registrant")
        self.assertRaises(domain.InvalidAction, mydomain.manage, myoldcontact,
                          "del", "registrant")
```

La bogue indiquait que la fonction `mydomain.manage` ne levait jamais d'exceptions, même lorsqu'elle aurait dû. Avant même de résoudre la bogue, un jeu de tests a été écrit, permettant de documenter et d'expliciter la bogue (les rapports de bogue sont souvent incomplets et insuffisants), puis de vérifier qu'elle était bien résolue, et qu'elle ne réapparaitra pas (chose que les bogues font souvent).

Le langage de programmation Haskell dispose quant à lui de HUnit <`http://hunit.sourceforge.net/`>. Par exemple, le programme suivant teste un analyseur de "*tags*" de langages (un moyen d'indiquer la langue, normalisé par l'IETF, dans le RFC 4646 [1]) :

```
shouldBeWellFormed tag =
   HUnit.TestCase (HUnit.assertBool (tag ++ " should be well-formed")
                        (Grammar.testTag tag == True))

shouldBeBroken tag =
   HUnit.TestCase (HUnit.assertBool (tag ++ " should *not* be well-formed")
                        (Grammar.testTag tag == False))

main = do
       brokenTags <- tagsFromFile brokenTagsFile
       wfTags <- tagsFromFile wfTagsFile
       let tests = HUnit.TestList (map shouldBeBroken (brokenTags) ++
                                   map shouldBeWellFormed (wfTags))
       HUnit.runTestTT tests
```

Ce programme, qui teste aussi bien des "*tags*" corrects que des "*tags*" erronés affiche le résulat suivant lorsque tout va bien :

```
% make test
./tests-from-files
Cases: 65  Tried: 65  Errors: 0  Failures: 0
```

Si une erreur est présente dans l'analyseur, on obtient :

```
% make test
./tests-from-files
### Failure in: 56
en-Latn-GB-boont-r-extended-sequence-x-private should be well-formed
Cases: 65  Tried: 65  Errors: 0  Failures: 1
```

1. Pour voir le RFC de numéro NNN, `https://www.ietf.org/rfc/rfcNNN.txt`, par exemple `https://www.ietf.org/rfc/rfc4646.txt`

et on voit tout de suite que l'analyseur est erroné.

HUnit, comme les autres modules présentés avant, exigeait que le programmeur écrive complètement les jeux de tests et donc les valeurs des paramètres passées aux fonctions testées. Le programmeur habitué à écrire des jeux de tests sait en général trouver les cas dignes d'être testés : ce sont souvent les cas situés aux limites, par exemple, pour une fonction qui cherche une sous-chaîne de caractères dans une chaîne, on testera les cas où la sous-chaîne cherchée est vide, où elle est plus longue que la chaîne, où la chaîne dans laquelle on cherche est vide, où la sous-chaîne ne se trouve pas dans la chaîne, où elle se trouve tout à la fin, etc. Cela ne suffit pas forcément et l'auteur du jeu de tests oublie facilement certains cas. En outre, écrire des jeux de tests peut être souvent fastidieux et il est préférable de les générer, au moins en partie, automatiquement.

C'est ce que permet le module QuickCheck <http://www.cs.chalmers.se/~rjmh/QuickCheck/> en Haskell. QuickCheck génère aléatoirement des valeurs pour les paramètres des fonctions testées et vérifie des **propriétés** sur ces fonctions, propriétés écrites par le programmeur, pour qui c'est une tâche moins mécanique et moins ennuyeuse que d'écrire les tests.

Comme exemple, prenons une fonction qui canonicalise des noms de domaine pour vérifier, avant réservation, que ces noms ne sont pas déjà pris. L'AFNIC a une règle particulière <http://www.afnic.fr/actu/nouvelles/nommage/CP20050517> pour les noms de communes françaises : celles-ci ont le privilège de voir leur nom réservé gratuitement, et cette réservation s'étend aux noms proches, c'est-à-dire ne s'en distinguant que par les caractères tiret, apostrophe ou espace. Le moyen le plus simple de mettre en œuvre cette réservation est de canonicaliser les noms et de comparer les formes canoniques. En Haskell, cela donne :

```
canonicalize cityname =
    map toLower
      (filter
           (\c -> if (c == '-') || (c == '\'') || (c == ' ') then
                   False
               else
                   True)
           cityname)
```

On veut maintenant vérifier la fonction `canonicalize`. Pour cela, on écrit des propriétés :

```
-- Canonicalization is idempotent
prop_idempotent s = (canonicalize . canonicalize) s == canonicalize s

-- Canonicalization removes some characters
prop_nomoredashes s = not (elem '-' (canonicalize s)) ||
                      not (elem ' ' (canonicalize s)) ||
                      not (elem '\'' (canonicalize s))
```

En emballant ces propriétés dans le programme QuickCheck complet, que voici, on va pouvoir tester :

```
import Char
import List
import Test.QuickCheck
import Text.Printf

import Canonicalize
```

```
main  = mapM_ (\(s,a) -> printf "%-25s: " s >> a) tests

-- We redefine the Char generator to get only ASCII characters
instance Arbitrary Char where
    arbitrary     = choose ('\0', '\128')
    coarbitrary c = variant (ord c `rem` 4)

prop_idempotent s = (canonicalize . canonicalize) s == canonicalize s

prop_nomoredashes s = not (elem '-' (canonicalize s)) ||
                      not (elem ' ' (canonicalize s)) ||
                      not (elem '\'' (canonicalize s))

tests  = [("Idempotence", test prop_idempotent),
          ("Delete all dashes", test prop_nomoredashes)]
```

Voici le résultat du test :

```
% runhaskell Tests.hs
Idempotence               : OK, passed 100 tests.
Delete all dashes         : OK, passed 100 tests.
```

QuickCheck dispose de très nombreuses options, permettant d'affiner ce résultat. L'exemple ci-dessus n'est donc que le plus simple (on a quand même utilisé un générateur à nous, pour que les chaînes de caractères ressemblent à de vrais noms de domaines).

Par exemple, une question essentielle des systèmes de tests à données aléatoires, comme QuickCheck, est la « qualité » des données de test. Dans l'exemple ci-dessus, si le nom de domaine ne comporte pas de caractères « spéciaux » comme le tiret ou l'espace, il a peu de chances de déclencher une bogue. Même chose si ce nom est très court. QuickCheck permet de compter si une donnée du test est « triviale », la trivialité étant définie par le programmeur du test. Réécrivons notre programme :

```
-- Tests if there are "special" characters in the string. Special are defined
-- by the AFNIC registration rules
no_special s = not (elem '-' s) &&
               not (elem ' ' s) &&
               not (elem '\'' s)

-- Trivial strings are strings without special characters or null strings
is_trivial s = null s || no_special s

-- Canonicalization is idempotent
prop_idempotent s = classify (is_trivial s) "trivial"
                    ((canonicalize . canonicalize) s == canonicalize s)
...
```

Ainsi, avec la nouvelle fonction `classify`, on va pouvoir déterminer que beaucoup de tests étaient en fait peu sévères :

```
Idempotence               : OK, passed 100 tests (84% trivial).
Delete all dashes         : OK, passed 100 tests (84% trivial).
```

Une autre façon de déterminer la force de nos tests est de récolter des statistiques sur leur distribution et de les afficher. Par exemple, si je mets un `collect` au lieu du `classify` :

```
prop_idempotent s = collect (round (fromIntegral (length s) / 10))
                    ((canonicalize . canonicalize) s == canonicalize s)
```

Le programme va alors afficher la répartition de la longueur des chaînes de test (arrondie à la dizaine la plus proche) :

```
Idempotence              : OK, passed 100 tests.
56% 0.
26% 1.
11% 2.
5% 3.
2% 4.
```

On voit que la grande majorité des chaînes sont courtes, trop courtes. Cela vaudrait donc le coup de travailler les générateurs pour augmenter la taille moyenne. Le mieux est d'augmenter le nombre de tests, QuickCheck augmentant automatiquement la taille de ceux-ci (les tests courts sont faits au début, pour des raisons de performance) :

```
detailed :: Config
detailed = Config
  { configMaxTest = 1000
  , configMaxFail = 10000
  , configSize    = (+ 3) . (`div` 2)
  , configEvery   = \n args -> let s = show n in s ++ [ '\b' | _ <- s ]
  }
...
tests  = [("Idempotence", check detailed prop_idempotent),
...
```

Cela nous donne (cette fois, on arrondit à la centaine la plus proche) :

```
Idempotence              : OK, passed 1000 tests.
59% 0.
28% 1.
10% 2.
2% 3.
0% 4.
```

Ce qui est bien meilleur.

Comme indiqué, d'autres langages ont un système équivalent. Les programmeurs C semblent apprécier Check <http://check.sourceforge.net/> ou libtap <http://www.onlamp.com/pub/a/onlamp/2006/01/19/libtap.html>.

Parmi les ressources disponibles sur les tests, notons le site Web <http://www.testing.com> et l'intéressante liste de diffusion Agile Testing <http://groups.yahoo.com/group/agile-testing/>.

Une fois les jeux de tests développés, on peut faire en sorte qu'ils soient automatiquement exécutés à intervalles réguliers (c'est l'idée d'**intégration continue <http://www.martinfowler.com/articles/continuousIntegration.html>** de Martin Fowler). Par exemple, un script shell, lancé par cron toutes les nuits, peut produire une page Web indiquant l'état actuel du programme en cours de développement, ce qui est une aide considérable pour les développeurs. C'est un système de ce genre qu'utilise NetBSD <http://blog.netbsd.org/tnf/entry/testing_netbsd_easy_does_it> (qui dispose de mécanismes spéciaux <http://blog.netbsd.org/tnf/entry/revolutionizing_kernel_development_testing_with> pour pouvoir appliquer cette méthode au noyau).

Cédric Beust, sur son blog <http://beust.com/weblog/archives/000404.html>, argumente que les tests ne sont pas toujours prioritaires et qu'ils ne doivent pas faire oublier le développement. L'argument ne me convainc guère : on voit beaucoup plus de programmes qui n'ont jamais été testés que de programmes dont les auteurs auraient passés trop de temps à tester.