Introduction aux tests

Vincent Danjean

18 juin 2019

Topic

Pourquoi faire des tests ?

pour trouver des bugs

- code non compilable
- code renvoyant des résultats incorrects

Pourquoi faire des tests ?

pour trouver des bugs

- code non compilable
- code renvoyant des résultats incorrects

pour vérifier que tout fonctionne bien

- ça ne correspond pas nécessairement à un bug (faute dans le code)
- peut être dû à des spécifications incomplètes
- code correct mais non sécurisé

pour évaluer/comparer les performances

- passage à l'échelle
- régressions de performances

Quand faire des tests ?

Cycle de développement classique

1. analyse des besoins
2. conception
3. codage
4. tests
5. déploiement
6. maintenance

Problèmes :

- les tests ne sont effectués que lorsque (presque) tout le code est écrit
- en cas de problème de conception, il faut revenir au début du cycle
  => très coûteux

Autres modèles de développement

Modèle en V

chaque étape développement est associé à des tests

- analyse des besoins -> test du système complet
- conception de haut niveau -> tests d'intégration
- conception de bas niveau -> tests unitaires
- codage du logiciel

Méthodes agiles

- développement par phases
- chaque phase :
  - apporte une fonctionnalité au produit
  - a sa propre série de tests associés

Modèles de développement et tests

En résumé

Tester un produit uniquement lorsqu'il est fini n'est pas efficace :

- pans entiers de développement à refaire/jeter
- déboggage trop complexe à mener (trop de choses en même temps)

Des tests en continu

- sur tous les aspects du produit
- éventuellement par une équipe complètement distincte de celle de développement

Limitations des tests (1/3)

Objectifs : détection des erreurs logicielles pour que les problèmes soient identifiés et corrigés

- loin d'être évident
- un test peut prouver la présence d'un problème
- mais les tests ne peuvent pas prouver l'absence de tout problème
- chaque test est mené (et est valide) dans un environnement particulier
- Ariane 5 : 800 kF d'économie en tests, $>$ 370 M$ de perte
  https://fr.wikipedia.org/wiki/Vol_501_d%27Ariane_5

Domaine de couverture des tests

Explosion combinatoire des entrées/actions possibles :

- couverture exhaustive généralement inenvisageable
- bug de la division du Pentium d'Intel
  https://fr.wikipedia.org/wiki/Bug_de_la_division_du_Pentium

Limitations des tests (2/3)

Tests non fonctionnels difficiles à évaluer

- performance et charge
- passage à l'échelle
- facilité d'utilisation
- sécurité
- internationalisation et localisation
- robustesse

Ces tests sont souvent difficiles à automatiser :

- demande d'un utilisateur réel
- mesure de ressentis
- etc.

Limitations des tests (3/3)

Tester n'est pas débogguer

Si un test échoue, plusieurs actions sont possibles :

- corriger un bug
- modifier la conception
- supprimer/adapter le test

Tests fonctionnels et non-fonctionnels

Tests fonctionnels

vérifie une fonction ou une action spécifique du code :

- souvent associé à des spécifications, à de la documentation
- parfois aussi associé à des cas d'utilisation (use case) ou à des scénario utilisateurs

Tests non-fonctionnels

- test sur des aspects non associés à des actions utilisateur ou des fonctions spécifiques
- exemples : passage à l'échelle, sécurité, ...

Méthodes de test

Boîte blanche

On connaît le code à tester

- tests portant sur API, ...
- rapport sur la couverture des tests (fonctions et lignes)

Boîte noire

On ne connaît que les spécifications

- les tests peuvent être élaborés indépendamment de l'écriture du code

Boîte grise

Le code n'est pas connu précisément, mais on peut connaître

- des algorithmes internes utilisés
- des structures internes

On peut aussi introduire des données

- exemple : BdD particulière injectée dans les tests

Niveaux de tests

tests unitaires pour tester des fonctions individuellement

- permet de tester les composants indépendamment les uns des autres

tests d'intégration pour tester les interactions entre composants

tests de système pour vérifier qu'un système complet répond bien aux attentes

tests d'intégration de système pour vérifier qu'un système complet interagit bien avec des systèmes tierces

tests de (non) régression pour vérifier qu'une fonctionnalité n'a pas disparu

- certains projets imposent un test de non régression avec chaque résolution de bug

Automatiser les tests

Génération automatique de tests

- à partir des spécifications
- à partir de l'analyse du code source

Génération automatique de rapport

environnements pour n'avoir qu'à écrire les tests eux-mêmes

- lancement et analyse automatique des tests

exécution automatique des tests

- fréquence constante (ex : toutes les nuits)
- à chaque commit

Topic

Tests au cours de ce projet

Éviter le modèle classique :

- ne pas prévoir de phase de tests uniquement à la fin
- prévoir les tests (et les écrire) dès le début
- les exécuter très fréquemment
- les garder jusqu'à la fin

outils conseillés

- programmes Java dédiés (tests unitaires)
- script shell pour automatiser

outils existants mais trop complexes/lourds pour ce projet :

- JUnit
- Plateformes d'intégration continue : gitlab, Travis, Jenkins, ...

Topic

Généralités sur les shells

Les shells permettent d'enchaîner plusieurs commandes facilement

- assez puissants pour réaliser des programmes
  - avec variables, boucles, tests, sous-programmes, etc.
- très rapide à écrire
- permet de mélanger des outils écrits dans divers languages

La programmation shell peut devenir complexe

- seul l'usage basique sera présenté ici

De nombreux shells existent

Deux grandes familles (syntaxe) :

- sh : bash, zsh, ksh, dash, posh, etc.
- csh : tcsh, csh, etc.

Principes fondamentaux de bash

Pas de typage

- manipulation uniquement de chaînes
- séparation par les espaces (pas de ',', '()', etc.)
- en fait, bash connaît maintenant aussi les tableaux et peut faire de l'arithmétique

Instruction de base : la commande/processus

- le programme est exécuté avec ses arguments (séparés par des espaces)
  - `programme arg1 arg2 arg3`
- il renvoie un code de retour
  - par convention, 0 : ok, <>0 : erreur

Entrée/sorties standard, sorties standard et d'erreur

Chaque programme possède

- une entrée standard (par défaut le clavier)
- une sortie standard (par défaut l'écran)
- une sortie d'erreur (par défaut l'écran)

Redirection avec le shell

- entrée standard depuis un fichier : `prog < input-file`
- sortie standard vers un fichier (écrase) : `prog > fichier`
- sortie standard vers un fichier (ajoute) : `prog » fichier`
- sortie d'erreur vers un fichier : `prog 2> fichier_err`
- sortie d'un programme vers l'entrée d'un autre : `ls -1 | wc -l`

Substitution de commande

Remplacement d'une commande par son résultat

```
$(commande arg1 arg2)
```

Exemple

```
echo "Ce répertoire contient $(ls -1 | wc -l) fichiers"

Ce répertoire contient 45 fichiers
```

Enchaînements de commandes

Séquences

- commandes séparées par " ;" ou par un retour à la ligne
- Commentaires introduits par '#'

```
VAR=toto ; echo val ; VAR=tutu # commentaire
echo $VAR

val
tutu
```

Groupement de commandes dans un sous-shell : (séquence)

Attention aux effets de bord du sous-shell

```
VAR=toto ; ( echo val ; VAR=tutu ) ; echo $VAR

val
toto
```

Alternatives et boucles while

Syntaxe

- `if commandes1 ; then commandes2 ; else commandes3 ; fi`
- `while commande1 ; do commandes2 ; done`
- toutes les commandes peuvent être complexes (séquences, etc.)
- la valeur de retour de la dernière commande de `commandes1` décide si le test du if ou du while est vrai ou pas
- Attention : 0 => vrai, <>0 => faux

Exemple

```
if ls | grep -q "name" ; then
    echo "un fichier 'name' existe dans le répertoire"
    ls
else
    echo "aucun fichier 'name' dans le répertoire"
fi

aucun fichier 'name' dans le répertoire
```

Les variables

Nommées avec un identifiant

- assignation : `VAR=value`
- utilisation (interpolation) : `$VAR`
- par convention, les variables sont souvent en majuscules

Attention : les espaces sont importants. Que font les lignes suivantes :

```
ID= value
MESSAGE=Bonjour à tous
MA VARIABLE=2~
```

Protection des espaces : ' et "

```
VAR=toto ; V1="$VAR $VAR" ; V2='$VAR $VAR'
echo "V1=$V1" ; echo "V2=$V2"

V1=toto toto
V2=$VAR $VAR
```

Variables et caractères spéciaux

Variables spéciales

`$0` le nom du script en cours d'exécution

`$1, $2, ...` les arguments du script en cours d'exécution

`$*` ou `$@` tous les arguments du script en cours d'exécution

`$?` le code de retour de la dernière commande exécutée

Caractères spéciaux

Non protégés, les caractères '*' et '?' sont remplacés par bash avant l'exécution d'une commande en recherchant le motif parmi les fichiers.

`*` remplace 0, 1 ou plusieurs caractères

`?` remplace 1 caractère

Exemples : afficher les fichiers terminant par '.tex'

- dont la première lettre est o : `ls o*.tex`
- dont la seconde lettre est r : `ls ?r*.tex`

Les variables d'environnement

Variables accessibles par les programmes lancés

la commande `env` sans argument permet de voir l'environnement courant

Quelques variables d'environnement classiques

- PATH : répertoires de recherche des commandes
- CLASSPATH : répertoires de recherche des classes JAVA
- DISPLAY : l'écran à utiliser pour ouvrir des fenêtres
- et beaucoup d'autres

Définition d'une nouvelle variable d'environnement

- en gardant la valeur actuelle de la variable : `export VAR`
- avec une nouvelle valeur : `export VAR=value`

Boucles for

Itération sur des chaînes

```
for variable in val1 val2 val3 ; do
   echo "ma variable vaut $variable"
done

ma variable vaut val1
ma variable vaut val2
ma variable vaut val3
```

Quelques commandes utiles pour les tests

Test (ou '[')

permet de tester l'égalité de chaînes ou la présences de fichiers

```
test "$VAR" = toto
if test -f name ; then echo "fichier 'name' présent"; fi
if [ -f name ] ; then echo "fichier 'name' présent"; fi
```

Diff

permet de comparer le contenu de deux fichiers

Grep

permet de filtrer les lignes en entrée

Sort

permet de trier les lignes en entrée

Quelques commandes utiles pour les tests (suite)

Stat

permet de récupérer des informations sur un fichier

```
# plein d'information sur 'mon_fichier'
stat mon_fichier
# juste la taille du fichier (stockée dans 'taille_fichier')
taille_fichier="$(stat -c "%s" mon_fichier)"
```

Exécution de scripts

interpréteur explicite `bash monscript`

interpréteur implicite :

- première ligne du script

  `#!/bin/bash`
- `chmod +x monscript`
- `./monscript`

Exemple de tests en shell

```
# test basique
java MonProgram 2 4 > sortie
if ! diff sortie sortie.reference > /dev/null ; then
    echo "ERREUR : MonProgram a produit une sortie non prévue"
fi

# enchaînement de 3 tests
for test in 1 2 3; do
    java MonProgram 2 4 < test-entree-$test > sortie
    if ! diff sortie sortie-$test.reference > /dev/null ; then
      echo "ERREUR : MonProgram a produit une sortie non" \
 "prévue pour le test $test"
    fi
done
```

Exemple de tests en shell

```
# récupération automatique des tests
PRG="Prog1 Prog2"
# Itération sur les programmes
for p in $PRG; do
  # Itération sur les fichiers d'entrée
  # de la forme test-Prog1-1, test-Prog1-2, ...
  for e in test-$p-* ; do
    # Récupération du numéro à la fin du nom
    n=$(echo $e | sed -e "s/test-$p-//")
    if ! java $p < $e > sortie ; then
      echo "test $p / $n : ERREUR : code retour $?"
    elif diff sortie sortie-$p-$n.reference > /dev/null ; then
      echo "test $p / $n : PASS"
    else
      echo "test $p / $n : ERREUR : sortie incorrecte"
    fi
  done
done
```

Pour aller plus loin

Avec bash

- consulter le manuel de bash (`man bash`)
- consulter le manuel des différentes commandes (`man commande`)
- regarder les nombreux tutoriaux de bash sur le web
  - attention : bash a de nombreuses autres fonctionnalités

Avec les tests

- observer et essayer des systèmes de tests plus performants
- mais à faire après le projet