Tests et débogage

INF304

Pourquoi tester un programme ?

(après tout, si les programmes sont corrects, il n'y a pas besoin de les tester, non ?)

Pourquoi tester un programme ?

(après tout, si les programmes sont corrects, il n'y a pas besoin de les tester, non ?)

- Taille des «vraies» applications
  Par exemple : un éditeur de texte, un compilateur, un navigateur web, un système d'exploitation, etc.

Pourquoi tester un programme ?

(après tout, si les programmes sont corrects, il n'y a pas besoin de les tester, non ?)

- Taille des «vraies» applications
  Par exemple : un éditeur de texte, un compilateur, un navigateur web, un système d'exploitation, etc.
- Développement en équipe : personne n'a l'ensemble du code «dans la tête»

Pourquoi tester un programme ?

(après tout, si les programmes sont corrects, il n'y a pas besoin de les tester, non ?)

- Taille des «vraies» applications
  Par exemple : un éditeur de texte, un compilateur, un navigateur web, un système d'exploitation, etc.
- Développement en équipe : personne n'a l'ensemble du code «dans la tête»
- Interaction avec le matériel, la plate-forme d'exécution
  Par exemple : logiciel embarqué (smartphone, pilote automatique, etc.)

Pourquoi tester un programme ?

(après tout, si les programmes sont corrects, il n'y a pas besoin de les tester, non ?)

- Taille des «vraies» applications
  Par exemple : un éditeur de texte, un compilateur, un navigateur web, un système d'exploitation, etc.
- Développement en équipe : personne n'a l'ensemble du code «dans la tête»
- Interaction avec le matériel, la plate-forme d'exécution
  Par exemple : logiciel embarqué (smartphone, pilote automatique, etc.)
- Il n'existe pas de méthode «sûre» de développement logiciel... et les erreurs logicielles peuvent coûter très cher ! (voire des vies humaines...)
  $\longrightarrow$ le test reste une des méthodes de validation les plus efficaces

Qu'est-ce qu'un «test» ?

À vous !...

Qu'est-ce qu'un «test» ?

- Activité de validation : on s'assure que le logiciel/programme/module a un comportement qui correspond à ce qui est attendu

- Activité de validation parmi d'autres...
  Autres méthodes de validations :
  - analyse statique [NB : le typage est une forme d'analyse statique]
  - calcul de complexité
  - preuve de programme ($\pm$ automatique)
  - model-checking
  - ...

- Ce qui caractérise le test : on va exécuter ce qu'on teste avec plusieurs entrées, choisies avec pertinence

Place du test dans le cycle de développement

Taille des applications/programmes, présence inévitables d'erreurs $\Longrightarrow$ **forte** importance du test dans le processus de développement de logiciel (environ 30 à 50% du temps de développement).

Différentes formes possibles :

- tester *a posteriori* (après la phase de codage)
  $\longrightarrow$ le plus classique, mais pas le plus efficace en coût et temps !
- tester au fur et à mesure
  $\longrightarrow$ meilleure solution, permet de détecter les pbs plus tôt (mais il faut pouvoir tester du code incomplet, nécessite de «simuler» les morceaux manquants ...)
- diriger le développement par le test (TDE, "test driven developemnt")
  - écrire les tests **avant** le code
  - répéter en permanence «coder-tester-coder-tester-...»

Méthodes de tests : généralités

En général, pour des gros projets logiciels, l'équipe de testeurs est distincte de l'équipe de développeurs.

Il existe de nombreux outils pour automatiser certaines phases du test :

- gain de temps
- moins d'erreurs possibles dans la mise en oeuvre des tests (on les verra au fur et à mesure)

On ne peut rien tester sans «spécification», c'est-à-dire une description du comportement attendu du programme

Remarque importante

Question : en pratique, à quoi sert le test ?

Remarque importante

Le test sert à rechercher des erreurs éventuelles, plutôt que de vérifier le bon fonctionnement du programme

- on ne trouvera des erreurs que là où on les a cherchées...
- surtout, l'absence d'erreurs ne permettra pas de «prouver» que l'application est «correcte»...

Différentes formes de test...

... pour différents objectifs :

- Tests fonctionnels ou de conformité
- Tests de robustesse
- Tests unitaires
- Tests de non-régression
- Tests de performance
- Tests de sécurité
- Tests d'intégration

(liste non exhaustive...)

Tests fonctionnels

Parfois appelés «tests de conformité»

$\longrightarrow$ tester que le programme est «conforme à sa spécification» (sujet des exercices à venir...)

Exemple : un programme de tri doit effectivement trier les éléments (mais pas en inventer ni en supprimer...)

Tests de robustesse

$\longrightarrow$ tester que le programme résiste à un environnement d'exécution «dégradé»

- pannes matérielles
- entrées «hors-spécification»

Par exemple : un pilote automatique, une interface graphique...

On utilise pour cela un «modèle de fautes», qui décrit les problèmes possibles (on est obligé de faire certaines hypothèses sur l'environnement...)

Tests unitaires/tests d'intégration

Tests unitaires

$\longrightarrow$ tester les différents sous-programmes, paquetages indépendamment, par exemple :

- une fonction de tri d'un tableau
- un paquetage de lecture d'un fichier image

Tests d'intégration

$\longrightarrow$ tester une application complète

Tests de non-régression

$\longrightarrow$ tester qu'un changement de version, une mise à jour, une correction ne «détruit pas» les fonctionnalités existantes

Par exemple : changement de version de l'OS, correction d'un bug...

Tests de performance

$\longrightarrow$ tester que les performances du programme sont satisfaisantes

- temps de réponse
- résistance à la charge
- occupation mémoire (RAM, disque, etc.), bande passante utilisée...
- etc.

Par exemple : un serveur Web, une application smartphone, ...

Tests de sécurité

$\longrightarrow$ tester que le programme résiste à un environnement d'exécution «volontairement hostile»

- éviter que l'application soit rendu inopérante («déni de service»)
- éviter que des informations soit modifiées (intégrité), accédées (confidentialité) par des utilisateurs non autorisés à le faire

Exemples : un serveur Web ; un système d'exploitation

Dans cette UE...

... on s'intéressera en particulier aux tests fonctionnels et de robustesse

Test fonctionnel : principe

- produire une suite de tests (ou un «jeu de test»), selon un objectif de test (= on sait ce qu'on veut tester)
- exécuter chacun de ces tests : $\longrightarrow$ décider quel est le verdict (par exemple en utilisant un oracle)

  $\longrightarrow$ on est sensé connaître à l'avance le résultat attendu
- établir un verdict : quelle est la conclusion du test ?

Objectifs de tests

Un objectif de test peut-être par exemple :

- une fonctionnalité particulière que l'on veut tester

  (ex : ajouter un utilisateur déjà existant dans une application, empiler un premier élément dans une pile vide, ...)

- un scénario complet d'exécution

  (ex : lire une image de taille 100x100, appeler une fonction inverse vidéo, la ré-écrire dans le même fichier)

Verdicts

Le verdict est généralement de la forme :

- Pass (le test a réussi)
- Fail (une erreur a été trouvée)
- Inconclusive (le test n'a pas pu se dérouler correctement ou n'a pas atteint son objectif)

Verdicts

Le verdict est généralement de la forme :

- Pass (le test a réussi)
- Fail (une erreur a été trouvée)
- Inconclusive (le test n'a pas pu se dérouler correctement ou n'a pas atteint son objectif)

Mais... qui établit ce verdict ?

Oracles

Dans l'activité de test, un oracle désigne ce qui permet d'établir le verdict du test à l'issue de son exécution

l'oracle peut soit être le testeur lui-même (qui «voit» l'exécution du test), soit un programme ou une fonction qui établit le verdict automatiquement (indispensable dès que le jeu de test est conséquent : plusieurs dizaines/centaines/milliers de tests...)

Exemple : vérifier qu'un tableau de 100 ou 1000 éléments est correctement trié ...

Rq : le rôle de l'oracle est de vérifier que la spécification a bien été respectée

Le plus dur reste à faire...

Toutes ces étapes peuvent être plus ou moins automatisées, mais le plus difficile reste la production de la suite de tests (= choisir des tests à exécuter).

$\longrightarrow$ on s'intéresse dans la suite à deux grandes techniques : le test «boîte noire» et le test «boîte blanche».

Test fonctionnel en «boîte noire»

«Boîte noire» = on ne se sert pas du programme source pour générer les tests, mais uniquement de sa spécification...

$\longrightarrow$ nécessité d'avoir des spécifications précises.

- Intérêt : le testeur n'a pas d'a priori, il peut imaginer des scénarios de test originaux
- Inconvénient : le testeur doit être créatif ; on ne sait pas quelle portion du code a été testée...

Approches pour la production de tests

Quelques approches possibles (non exclusives...) :

- test aux bornes $\longrightarrow$ tester les valeurs limites des entrées

  ex : trier un tableau vide, inverser une image entièrement noire

- partitionner l'ensemble d'entrée en fonction du résultat attendu

  $\longrightarrow$ énumérer les différentes classes de résultats possibles et choisir les entrées susceptibles de générer ces résultats

  ex : recherche du max de 3 entiers (max = 1^{er}, 2^{e} ou 3^{e} entier saisi)

- partitionner l'ensemble des entrées en fonction de sa structure (vis-à-vis du résultat attendu)

  ex : tester le tri de 3 entiers (séquence d'entrée croissante, décroissante, stationnaire, croissante puis décroissante, etc.)

- test aléatoire $\longrightarrow$ on choisit aléatoirement des entrées (correctes)

Deux remarques importantes...

- notion de «couverture du domaine des entrées» à avoir en tête ; mais comme dans le cas général on ne peut pas être exhaustif, on se base sur une partition (c'est une «hypothèse d'uniformité»)

Deux remarques importantes...

- notion de «couverture du domaine des entrées» à avoir en tête ; mais comme dans le cas général on ne peut pas être exhaustif, on se base sur une partition (c'est une «hypothèse d'uniformité»)

- sauf exception (i.e., si on choisit explicitement la méthode «test aléatoire»), on n'écrit pas de tests au hasard, mais on suit une méthode systématique qu'il faut définir au préalable

  NB : c'est cette méthode qu'on vous demande de décrire dans les exercices...

Test fonctionnel «boîte blanche»

«Boîte blanche» = le testeur utilise le code source du programme à tester pour produire les tests.

Deux approches complémentaires possibles :

- On peut (doit) appliquer la même technique que pour les tests en «boîte noire» (raisonnement à partir de la spécification)
- On peut EN PLUS utiliser le code source. Approche basée sur la notion de «couverture de code» :
  - couverture des fonctions/sous-programme
    $\longrightarrow$ chaque fonction doit être exécutée au moins une fois par au moins un test du jeu de tests
  - couverture des instructions
    $\longrightarrow$ chaque instruction doit être exécutée au moins une fois dans un des tests
  - plus complexe (voire parfois impossible) à mettre en œuvre : couverture des chemins d'exécution

Le débogage

- On ne «débuggue» pas à partir de rien : on part d'un test qui a «échoué»...
  (À répéter : **ON N'EFFACE JAMAIS UN TEST, QUEL QUE SOIT SON RÉSULTAT**)

Le débogage

- On ne «débuggue» pas à partir de rien : on part d'un test qui a «échoué»...
  (À répéter : **ON N'EFFACE JAMAIS UN TEST, QUEL QUE SOIT SON RÉSULTAT**)
- Deux approches possibles :
  - À partir du test : essayer de «caractériser» le bug en essayant de le reproduire avec différents tests (hypothèse : «quelle propriété du test a fait apparaître ce bug ?»)
  - À partir du programme : identifier la position/la source de l'erreur
    - partir d'hypothèses sur l'exécution du programme : pour une entrée donnée (le test qui a échoué...), on connaît *a priori* le déroulement attendu du programme
      Par ex. : valeur d'une variable à une position donnée, nombre d'itérations, chemin d'exécution emprunté, ...
    - on cherche ensuite à quel endroit l'exécution du programme diverge de ces hypothèses
      $\longrightarrow$ outils : assertions, points d'observations, ...
      Ou encore mieux : utilisation d'un débogueur !