TD2 — Tests

Support enseignant

1. *Partie cours : test de programmes*

Pourquoi tester un programme ?

(... alors que tout étudiant en informatique est sensé passer des heures à apprendre à écrire des programmes corrects... 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.
  - *plusieurs milliers/millions de lignes de code, écrites par différentes équipes ou par ré-utilisation de code existant*
  - *contiennent nécessairement des erreurs de programmation*
  - *utilisation d'outils comme débogueur ou assertions fastidieuses pour de gros codes, et impossibles pour des boîtes noires.*
- *Interaction avec le matériel, la plate-forme d'exécution*
  Par exemple : logiciel embarqué (smartphone, pilote automatique, etc.)
  - *une application est en général la combinaison de logiciels et matériels dédiés ;*
  - *il n'existe pas (encore) de méthode de développement infaillible ;*
  - *il est nécessaire de tester le produit final dans son ensemble.*
- *Il n'existe pas de méthode «sûre» de développement logiciel ($\neq$ construction d'un bâtiment, formules de résistance des matériaux, ...) et les erreurs logicielles peuvent coûter très cher !*
  $\longrightarrow$ Le test reste une des méthodes de validation les plus efficaces.

*$\Longrightarrow$ **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*
  - *répeter en permanence «coder-tester-coder-tester-...»*

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)*

*Remarque importante : le test sert à **rechercher des erreurs éventuelles**, plutôt que de **vérifier le bon fonctionnement**.*

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

Remarque 2 : on peut pas tester si on a pas de «spécification» (c'est-à-dire, une description *du comportement attendu du programme).*

Différentes formes de test pour différents objectifs

On peut tester un programme dans différents buts (liste non exhaustive) :

Tests fonctionnels (ou tests de conformité)

⟶ tester que le programme est «conforme à sa spécification» (sujet des exercices à venir...)
Exemple : un programme de tri doit trier les éléments (mais pas en inventer ni en supprimer...).

Tests unitaires

⟶ 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 de non-régression

⟶ 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

⟶ tester que les performances du programme sont satisfaisante

- *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 robustesse

⟶ 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.
Remarque : lien ici avec la saisie défensive ?

Tests de sécurité

⟶ 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
Remarque : dans les 4 derniers cas on ne s'intéresse pas du tout aux fonctionnalités de l'application...

Tests d'intégration

⟶ tester une application complète

Le test fonctionnel

- *Principe :*
  - *produire une suite de tests (ou un jeu de test), selon un* objectif de test
  - *exécuter chacun de ces tests :* $\longrightarrow$ *décider quel est le verdict en utilisant un oracle*
  - *en fonction du résultat, décider de recommencer ou non d'autres 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)
  - *un scénario complet d'exécution*
    (ex : lire une image de taille 100x100, appeler une fonction inverse vidéo, la ré-écrire dans le même fichier)
- *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)*
- ***l'oracle** peut soit être le testeur lui-même (qui «voit» l'exécution du test), soit un programme qui établit le verdict automatiquement (éventuellement à partir d'un log)*
  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

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 blanche» et le test «boîte noire».*

Le test fonctionnel en «boîte blanche»

«Boîte blanche» = le testeur utilise le code source du programme a tester pour produire les tests.
L'approche la plus classique est basée sur la notion de «couverture de code» :

- *couverture des fonctions/sous-programme*
  $\longrightarrow$ *chaque fonction doit être exécutée au moins une fois*
- *couverture des instructions*
  $\longrightarrow$ *chaque instruction doit être exécutée au moins une fois dans un des tests*

Exercice : test en «boîte blanche»

Exercice 1. Soit le programme C suivant :

```
1  #include <stdio.h>
2
3  int main() {
4    int n;
5    int p,d,aux,ec;
6    int i;
7
8    printf("Nombre d'entrées : ");
9    scanf("%d", &n);
10
11   scanf("%d", &p);
12   scanf("%d", &d);
13   if (d > p) {
14     aux = p;
15     p = d;
16     d = aux;
17   }
18   for (i = 3; i <= n; i++) {
19     scanf("%d", &ec);
20     if (ec > p) {
21       d = p;
22       p = ec;
23     } else if (ec > d) {
24       d = ec;
25     }
26   }
27   printf("Valeur de p : %d\n", p);
28   printf("Valeur de d : %d\n", d);
29 }
```

1. Que fait ce programme ?

   Le programme affiche les deux plus grandes valeurs d'une séquence d'entiers. Le premier entier en entrée est la taille de la séquence.

2. Quel est le domaine de valeurs valides des entrées ?

   Entrées constituées :

   (a) d'un entier $N \geq 2$

   (b) d'une séquence de (ou «d'au moins») N entiers

3. Écrire un jeu de test complet pour ce programme, en justifiant sa construction.

   Deux approches possibles :

   — *Couverture des instructions : il faut au moins deux séquences pour couvrir les deux cas de la condition de la ligne 13.*

   Par exemple :

   — $[1, 2] \longrightarrow$ *couvre les instructions des lignes 13 à 17*

   — $[2, 1, 4, 3] \longrightarrow$ *couvre les instructions de la boucle* ***for***

   — *Partitionnement du domaine d'entrée : on peut s'appuyer sur le programme pour réaliser le partitionnement.*

   — *Les deux plus grandes valeurs peuvent être placées dans n'importe quel ordre dans la séquence (la première d'abord, ou la deuxième) $\longrightarrow$ 2 possibilités*

— *Les deux premières valeurs (A et B) sont traitées de manière particulière : on peut examiner tous les cas possibles :*
  — *A et B sont les deux plus grandes valeurs de la séquence*
  — *A est une des deux plus grandes valeurs, B non*
  — *B est une des deux plus grandes valeurs, A non*
  — *A et B ne sont aucune des deux plus grandes valeurs*

On combine les deux, cela donne 8 tests.

4. Que se passe-t-il si des données en-dehors du domaine de validité sont fournies ?
   — *Séquence de longueur < 2 : erreur d'entrées/sorties sur* `scanf("%d",&p)` *ou* `scanf("%d",&d)` *(en fait, «comportement non attendu» de scanf, puisqu'il n'y a pas d'exception en C).*
   — *Nombre de valeurs $< N$: erreur d'entrées/sorties sur* `scanf("%d",&ec)`

Le 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...*

Quelques approches possibles :

— *test aux bornes*
  $\longrightarrow$ tester les valeurs limites des entrées
  ex : trier un tableau vide, inverser une image ne contenant que des pixels noirs
— *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 : tester la recherche du max de 3 entiers (max = 1er entier, 2ème entier ou 3ème 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, décroissante puis croissante)
— *test aléatoire*
  *$\longrightarrow$ on choisit aléatoirement des entrées (**correctes**) (la fonction de génération aléatoire peut éventuellement privilégier certaines entrées)*

Remarque : notion de «couverture du domaine des entrées», mais comme dans le cas général on ne peut pas être exhaustif, on se base sur une partition (c'est une «hypothèse d'uniformité»).