Tests de programme et inspection de code

I. Principes généraux des tests

A. Écrire du bon code

Définition 1 [BAR 2.1.1] **La spécification** (ou contrat) d'un programme consiste à définir 4 caractéristiques :

- Les **entrées** qui définissent une instance du problème
- Une **précondition** que les entrées doivent vérifier
- Les **sorties** qui décrivent une solution à l'instance du problème
- Une **postcondition** qui doit être vérifié après l'exécution du programme sur des entrées qui vérifient la précondition. Elle vérifie la validité des sorties.

Exemple 2 On peut définir une classe `CompteARebour` qui contient deux méthodes :

- une méthode `substract` qui soustrait une valeur au compteur actuel, on vérifie que la valeur donnée est positive.
- une méthode `lessThanZero` qui renvoie `True` si le compteur actuel est négatif et `False` sinon.

Le code Python suivant vérifie cette spécification et sera testé au cours de la leçon.

```
1  class CompteARebour :
2    def __init__(self,init_val) :
3      self.counter = init_val
4
5    def substract(self,val) :
6      assert val >= 0
7      self.counter -= val
8
9    def lessThanZero(self) :
10     return self.counter <= 0
```

Définition 3 **Programmation défensive** [TOR 5.3] Afin de rendre son code plus robuste, on peut s'assurer du bon déroulement de l'exécution en vérifiant notamment :

- le respect des préconditions sur les entrées
- la validité des pointeurs allouée en C
- l'état des fichiers ouverts par le programme

Exemple 4 La ligne 6 de la définition de la classe `CompteARebour` est un exemple de programmation défensive.

B. Premières notions de tests

Définition 5 [BAR 2.2.2] Afin de vérifier le bon fonctionnement d'un programme, on dispose de plusieurs méthodes qui se répartissent en deux principales catégories :

- la *vérification dynamique* qui se propose d'exécuter le code pour s'assurer d'un fonctionnement correct
- la *vérification statique* qui consiste en une revue du code (automatique ou manuelle), des lectures croisées et une analyse a posteriori d'anomalies.

Définition 6 [BAR 2.2.2] **Un test** d'un programme est un processus (manuel ou automatique) qui a pour objectif de détecter des différences entre les résultats fournis par le programme et ceux attendus par sa spécification et/ou d'établir la présence de certaines erreurs particulières.

Remarque 7 [BAR 2.2.2] Un test ne peut pas garantir l'absence de faute, ni les corriger, ils ne servent qu'à les trouver.

Remarque 8 [KOR 3.1.1] Un test classique se présente sous la forme Arrange, Act, Assert (AAA) :

- Arrange, on amène l'environnement à l'état voulu (initialisation des variables, ...).
- Act, on exécute le code à tester et on stocke les valeurs sortie s'il y en a.
- Assert, on vérifie que la propriété testé est vérifiée (sur les valeurs de sortie ou la postcondition par exemple)

Exemple 9 Le test de l'initialisation de notre classe `CompteARebour` :

```
1 def test_CAR_init():
2   val = random.random() # Arrange
3   compteur = CompteARebour(val) # Act
4   assert compteur.counter == val # Assert
```

Remarque 10 [TOR 5.3] En Python et en OCaml, on peut utiliser le mot clé `assert`, en C, on peut utiliser la fonction `assert()` en important `assert.h`.

II. Différents types de tests

A. Tests fonctionnel/non fonctionnel

Définition 11 Un test est dit **fonctionnel** s'il a pour objectif de vérifier que le code source correspond à la spécification donnée.

Définition 12 [BAR 2.2.2] **Un test non fonctionnel** vérifie d'autres propriétés que la correction du code comme :

- les performances (temps d'exécution, utilisation mémoire, ...)
- la sécurité du programme
- l'utilisabilité du programme

Exemple 13 **Asan** Pour détecter les problèmes de mémoires, on modifie les fonctions d'allocation et d'accès aux cases mémoires. DEV

Exemple 14 **Mesurer le temps d'exécution** [TOR 5.4] Pour mesurer le temps d'exécution d'un programme on peut utiliser la la commande `time` du shell Linux. Pour effectuer des mesures plus précises sur des portions du code, on peut utiliser la fonction `clock` de la bibliothèque `time` pour du code C et la fonction `time` du module `Sys` en Ocaml.

B. Tests unitaires/d'intégration

Définition 15 **Test unitaire** [BAR 2.2.2] Un test unitaire teste des procédures ou des fonctions du code de manière isolé pour contrôler si elles correspondent à la spécification.

Exemple 16 On teste ici la méthode `lessThanZero` de notre classe `CompteARebour`:

```
1 def test_CAR_LTZ_unit(val):
2   compteur = CompteARebour(0)
3   compteur.counter = val
4   assert compteur.lessThanZero() == (val <= 0)
```

Définition 17 **Test d'intégration** [BAR 2.2.2] Au contraire un test d'intégration se concentre sur le bon comportement lors de la composition des procédures. L'objectif est de vérifier la stabilité et la cohérence des interfaces et interactions de l'application.

C. Tests boite noire/ boite blanche

Définition 18 **Test en boite noire** [CHA 4] Un test est en boite noire si on ne teste qu'en connaissance de la spécification du programme et pas de son implémentation.

Exemple 19 Un test en boite noire de la méthode `lessThanZero` :

```
1 def test_CAR_LTZ_black_box(val):
2   compteur = CompteARebour(val)
3   assert compteur.lessThanZero() == (val <= 0)
```

Définition 20 **Test en boite blanche** [CHA 5] Un test est en boite blanche si on considère l'implémentation du code testé.

Exemple 21 Pour tester la méthode `substract` de notre classe, on peut écrire un test unitaire en boite blanche :

```
1 def test_CAR_sub(val_ini,val_sub):
2   compteur = CompteARebour(val_ini)
3   compteur.substract(val_sub)
4   assert compteur.counter == val_ini-val_sub
```

III. Bien définir ses tests : notion de couverture

Définition 22 **Métrique de couverture** [KOR 1.3] Une métrique de couverture indique quelle proportion de code source une suite de test exécute, allant de 0 à 100%.

Remarque 23 [KOR 1.3] Un faible couverture indique une mauvaise suite de test, mais une bonne couverture n'implique pas nécessairement une bonne suite de test.

Remarque 24 Pour mesurer la couverture, on peut utiliser des programmes comme `gcov` ou `pytest-cov`

A. Couverture de ligne

Définition 25 **Couverture de ligne** [KOR 1.3.1] La couverture de code indique le ratio entre le nombre de lignes de code exécutés par au moins un test et le nombre de lignes de code total.

$$\text{Couverture de ligne} = \frac{\text{Lignes de code executées}}{\text{Lignes de code total}}$$

Exemple 26 [KOR Listing 1.1] Pour la fonction est_text_long :

```
1 def est_text_long(s: str) -> bool:
2   if (len(s) > 5):
3     return True
4   else:
5     return False
```

On a ici 5 lignes de code. Le test `assert(est_text_long("abc"))` passe par 4 lignes de code, soit une couverture de code de 80%

Remarque 27 [KOR 1.3.1] On peut changer le taux de couverture de code, simplement en réécrivant la fonction :

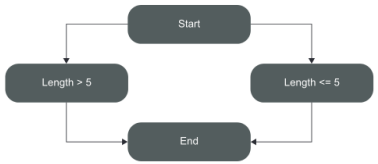
```
1 def est_text_long(s: str) -> bool:
2   return (True if len(s) > 5  else False)
```

On aura alors une couverture de 100%, avec le même test que précédemment.

B. Couverture de branches

Définition 28 Graphe de flot de contrôle [BAR 2.2.3] d'un programme est un graphe orienté dont les sommets sont les états du programme et les arêtes sont les transitions possibles d'un état à l'autre, correspondant à des branchements (`if`, `while`, ...).

Exemple 29 [KOR 1.3.2] Si on reprend l'exemple de la fonction `est_text_long`, on a un branchement lors du `if`, et donc deux branches p ossibles :



[KOR Figure 1.5]

Definition 30 [KOR 1.3.2] Plutôt qu'utiliser les lignes de codes comme métrique, on va se servir de la couverture de branche :

$$\text{Couverture de branche} = \frac{\text{Branches traversées}}{\text{Nombre total de branches}}$$

Méthode 31 Mutation d'entrée (ou *fuzzing*) est une méthode reposant sur une génération dynamique de cas de tests, cherchant par exemple à augmenter les branches du programme couvertes.

Méthode 32 Mutation de code (ou *mutation testing*) est une méthode cherchant à vérifier la qualité d'une suite de test en modifiant légèrement le code du programme pour s'assurer que les tests ne passent plus.

DEV

IV. Inspection de code [CHA 6]

Remarque 33

- Les tests reposent sur des *spécifications*, qui ne sont pas toujours suffisamment précises
- Même avec une couverture parfaite, les tests ne peuvent pas assurer une absence de bugs.
- Les tests ne mesurent pas certaines propriétés du code, telle que sa lisibilité et sa réutilisabilité.

Définition 34 Inspection de code [CHA 6.1] L'inspection de code est un processus d'équipe, dans lequel chacun relis le code, les tests et la documentation, avant de mettre en commun les potentiels défauts lors d'une réunion. Ce processus doit suivre une structure bien défini, pour trouver le plus de problèmes possibles.

Exemple 35 Check-list pour les tests unitaires

- Toutes les erreurs lèvent des exceptions
- Les tests couvrent bien les cas limites
- Les tests couvrent bien les spécifications attendues

Exemple 36 Bonnes pratiques de programmation [TOR 5] Le relecteur peut vérifier que le code suit des bonnes pratiques de programmation :

- L'indentation du code est claire et cohérente avec le reste.
- Les variables ont des noms qui indiquent clairement leurs utilités.
- Le code est commenté pour en expliquer son fonctionnement.
- Toutes les exceptions sont gérées.

Remarque 37 Des outils automatiques comme `clang-tidy` ou `pylint` permettent de forcer certaines pratiques de programmation, pour garantir une cohérence entre le code de différents développeurs.

Tests de programme et inspection de code

I. Principes généraux des tests

A. Écrire du bon code

1 Def [BAR 2.1.1] La spécification

2 Ex [NAN]

3 Def Programmation défensive [TOR 5.3]

4 Ex

B. Premières notions de tests

5 Def [BAR 2.2.2]

6 Def [BAR 2.2.2] Un test

7 Rem [BAR 2.2.2]

8 Rem [KOR 3.1.1]

9 Ex [NAN]

10 Rem [TOR 5.3]

II. Différents types de tests

A. Tests fonctionnel/non fonctionnel

11 Def Test fonctionnel

12 Def [BAR 2.2.2] Un test non fonctionnel

13 Ex Asan

14 Ex Mesurer le temps d'exécution [TOR 5.4]

B. Tests unitaires/d'intégration

15 Def Test unitaire [BAR 2.2.2]

16 Ex [NAN]

17 Def Test d'intégration [BAR 2.2.2]

C. Tests boite noire/ boite blanche

18 Def Test en boite noire [CHA 4]

19 Ex [NAN]

20 Def Test en boite blanche [CHA 5]

21 Ex [NAN]

III. Bien définir ses tests : notion de couverture

22 Def Métrique de couverture[KOR 1.3]

23 Rem [KOR 1.3]

24 Rem [NAN]

A. Couverture de ligne

25 Def Couverture de ligne[KOR 1.3.1]

26 Ex [KOR Listing 1.1]

27 Rem [KOR 1.3.1]

B. Couverture de branches

28 Def Graphe de flot de contrôle [BAR 2.2.3]

29 Ex [KOR 1.3.2]

30 Def [KOR 1.3.2]

31 Métho Mutation d'entrée

32 Métho Mutation de code

IV. Inspection de code [CHA 6]

33 Rem

34 Def Inspection de code[CHA 6.1]

35 Ex Check-list pour les tests unitaires

36 Ex Bonnes pratiques de programmation [TOR 5]

37 Rem [NAN]

Bibliographie

[BAR] V. Barra, *Informatique MPI/MP2I.*

[TOR] T. Balabonski & S. Conchon & J. Filliâtre & K. Nguyen & L. Sartre, *MP2I MPI, Informatique Cours et exercices corrigés.*

[KOR] V. Khorikov, *Unit Testing : Principles, Practices and Patterns.*

[CHA] N. Chauhan, *Software Testing : Principles and Practices.*

Axel Stengel & Benjamin Voisin