Spécifier & Tester Classes et Objets

Virginia Aponte

CNAM-Paris

14 septembre 2023

Que teste-t-on pour une classe ?

1. On teste *la cohérence de l'état interne* $\Rightarrow$
   - ex : une cabine de métro autonome dans l'état *en déplacement* ne peut pas avoir *ses portes ouvertes*.
2. ET, on teste les *contrats des méthodes* $\Rightarrow$
   - faire comment pour les méthodes statiques . . .
   - . . . sauf que, certaines méthodes ne renvoient pas de résultat, mais modifient l'état de l'objet $\Rightarrow$
     - tester que ces modifications correspondent au contrat
     - tester que l'état interne *après modifications* reste cohérent

Quelle forme prend une spécification de classe ?

- soit une interface au sens Java + contrats javadoc
- soit un squelette de classe + contrats javadoc

$\Rightarrow$ décrit des contrats pour **les 2 parties** d'un objet :

1. contrat javadoc pour l'état interne de l'objet :
   $\Rightarrow$ qu'est-ce « qu'un état interne cohérent » ?
2. contrats javadoc pour les méthodes de la classe/interface ;

Interface `CompteCrediteur`

Exemple : les comptes bancaires « toujours en positif ».

```
   /* Comptes bancaires de solde toujours positif */
    interface CompteCrediteur {
          public double getSolde();
          public void retrait(double m);
          public void depot(double m);
}
```

Nous devons y ajouter des commentaires javadoc avec :

- contrat pour la cohérence de l'état de l'objet (invariant)
- des contrats pour ses méthodes

Le contrat de cohérence (ou invariant) d'état interne

A tout moment, le solde courant du compte est positif.

- Doit rester VRAI à tout moment (on dit : « invariant »)

$\Rightarrow$ à la création de l'objet, avant et après chaque méthode.

- Appellé Invariant de cohérence ou (« contrat de cohérence »)
- placé (commentaire javadoc) au début de l'interface/classe.

```
/**
 * Invariant: this.getSolde() > 0 (toujours positif)
 */
public interface CompteCrediteur {
```

Classe Compte qui implante CompteCrediteur

```
interface Compte implements CompteCrediteur {
    private double solde;  // solde courant du compte
    /**
     * Retourne le solde courant du compte.
     */
    public double getSolde(){
       return this.solde;
    }
```

- Ce code retourne bien le solde courant du compte ;
- mais, pouvons nous garantir qu'après son appel, le solde courant est toujours positif ? *Uniquement si le solde était positif avant l'appel*
- autrement dit, la méthode ne peut pas garantir la cohérence de l'état *après appel*, si l'état était incohérent avant appel.

Invariants +contrats de méthodes

- comportement d'une méthode : on ne peut rien garantir si les données de l'objet sont incohérentes avant l'appel.
- Conclusion : pour qu'une méthode se comporte selon son contrat, et aussi pour la tester il faut :
  1. l'état est cohérent *avant appel*,
  2. *après appel* : la postcondition est vrai ET AUSSI l'invariant.

Quels sont les tests à réaliser ?

Notre but

Nous assurer que tout objet est cohérent à sa création ET que toute méthode après appel, ne change pas cet état en incohérent.

On teste

1. les contructeurs : ne construisent pas d'objets incohérents ;
2. les méthodes se comportent selon leur contrat (comme avant), ET EN PLUS, on veuille à ce que :
   1. l'objet sur lequel on fera l'appel est cohérent *avant appel*,
   2. *après appel* : l'état de l'objet est cohérent.
   3. *après appel* : l'état de l'objet correspond au contrat de la méthode.

Comment tester un objet ?

Préalable pour tester un objet : il faut . . . en avoir céee un !

Un cas de test

Comme avant, un cas de test est mis dans une méthode de test, où

1. on crée un nouvel objet, initialisé avec les valeurs pertinentes pour ce cas de test ;
2. si on veut tester l'état interne de cet objet, on invoque les bons *getteurs* dessus ;
3. si on veut tester le comportement d'une méthode :
    1. l'état de l'objet avant appel doit être cohérent ;
    2. on invoque sur cet objet la méthode à tester ;
    3. on teste que le comportement correspond à son contrat ;
    4. et si ce n'était pas un cas d'échec, on vérifie que l'état après appel est toujous cohérent (*getteurs*).

Contrat du constructeur

L'invariant doit être vérifié à tout moment :

$\Rightarrow$ pas de création d'objets incohérents.

```
/*
 * Creation d'un compte  de solde initial n.
 * @param n :  montant initial
 * post : this.getSolde = n  &&  this.getSolde() >0
 * @throws IllegalArgumentException si n <= 0
 */
Compte(double n)
```

Tester le constructeur

```
/* Test du constructeur, impossible de construire
 * un objet incoherent */
@Test(expected = IllegalArgumentException.class)
  public void testFailConstruction() {
      System.out.println("failToConstruct");
      Compte c = new Compte(-10.0);
}
```

Teste l'échec du constructeur à construire un objet incohérent.

Tester le constructeur : contrat

Teste si le constructeur initialise correctement l'objet.

```
/* Test du constructeur, postcondition */
@Test
  public void testPostConstruction() {
      System.out.println("Postcondition Construct");
      double init = 30.0;
      Compte c = new Compte(nit);
      assertTrue(c.getSolde()==init);
}
```

Contrat pour `retrait`

```
/**
 *  Retire m du solde courant si m < this.getSolde().
 *  @param m montant a retirer
 *  pre :  m > 0  &&  invariantOK
 *  post:  this.getSolde() = oldSolde - m
 *                 &&  invariantOK
 * @throws IlegalArgumentException si oldSolde <= m
 */
void retrait(double m);
```

retrait(n) : cas avec échec

```
/* Test de retrait, echec solde insuffisant. */

@Test(expected = IllegalArgumentException.class)
   public void testRetraitFail()  {
       Compte c = new Compte(80);
       c.retrait(100);
 }
```

Test pour `retrait(n)` : cas sans échec

```
/* Test de retrait: postcondition */
@Test
  public void testRetraitOK()  {
      Compte c = new Compte(80);  // invariant OK.
      double oldsolde = c.getSolde();
      c.retrait(10);
      assertEquals(c.getSolde(),oldsolde-10,0 );
      assertTrue(c.getSolde() > 0 ); // invariant OK.
 }
```

Tester l'invariant via une méthode dans la classe Compte

Nous déclarons une méthode qui teste si l'invariant est valide.

Elle est définie dans la classe Compte,
car elle doit accéder aux variables privées de l'objet !

```
public invariantOK(){
        return this.solde >0;
}
```

Encapsulation et tests

- L' état interne des comptes est abstrait : toutes les variables d'instance sont protégées.
- Conséquence : les méthodes de tests n'ont pas accès à l'état interne ;
  - pour tester contrats de méthodes + invariant d'état :
  - il faut disposer de méthodes "observateurs" de l'état (getSolde(), getNum(), etc).
  - sans ces méthodes, impossible de faire les tests !