Analyse et Programmation Orientées Objets / C++

Modules de tests unitaires

Charte de réalisation

Réalisation des tests unitaires

➡ Principes

Le développement d'une classe C++ doit être accompagné en parallèle et avec le même soin de la réalisation de ses modules de tests unitaires.

Usage de la fonction *main*

Deux classes distinctes ne doivent pas partager un même module de tests unitaires

Rôle du module de tests unitaires

Un module de tests unitaires doit permettre de vérifier la conformité de la classe vis à vis de ses spécifications fonctionnelles.

Il doit exécuter au moins une fois (cas nominal) :

- chacun des constructeurs
- chacune des méthodes publiques

Limites des tests unitaires (1)

Un module de tests unitaires ne prouve pas que la classe cible fonctionne parfaitement.

Seuls les langages autorisant la preuve formelle de programme permettent d'atteindre l'exhaustivité :

- langage Prolog
- ...

Limites des tests unitaires (2)

Un module de tests unitaires permet seulement de vérifier que les résultats fournis par les méthodes invoquées dans les tests sont conformes aux résultats attendus.

Le programmeur a la charge de définir de manière exhaustive tous les résultats attendus.

Conditions d'exécution et de contrôle

La maîtrise des exigences fonctionnelles de la classe cible doit être l'unique pré-requis pour exécuter son module de tests unitaires et en contrôler les résultats.

Aucune explication ou connaissance particulière ne doit être nécessaire à l'opérateur, et notamment celle des codes sources.

Organisation des tests unitaires (1)

Un module de tests unitaires est une succession de "***test design*** » parfaitement identifiés.

Chaque "***test design***" permet de vérifier une fonctionnalité majeure de la classe.

Chaque "***test design***" produit un compte-rendu global de contrôle de la fonctionnalité cible.

Les "***test design***" sont structurellement indépendants.

Organisation des tests unitaires (2)

Un "***test design***" est une succession de "***test case***" parfaitement identifiés.

Chaque "***test case***" permet de vérifier un sous-ensemble cohérent de comportements attachés à la fonctionnalité cible.

Chaque "***test case***" produit un compte-rendu local de contrôle des comportements cibles.

Les "***test case***" sont structurellement indépendants.

Organisation des tests unitaires (3)

Un "***test case***" est une succession de "***test unit***".

Chaque "***test unit***" permet de vérifier le résultat obtenu par l'exécution d'une méthode cible par rapport à un résultat attendu et fourni par le programmeur du test.

Niveaux "***test design***" et "***test case***" obligatoires.

La classe prédéfinie *Tests*

➡ Implémente les outils facilitant le respect de la charte décrite ci-avant :

- Usage fortement recommandé en TD&TP
- Non instanciable
- Mise en oeuvre très intuitive

Les méthodes de la classe *Tests* (1)

➡ Begin

- Usage obligatoire avant la description du premier *test design*
- Nom de la classe cible et version en paramètres

➡ End

- Usage obligatoire après la description du dernier test case du dernier test design
- Aucun paramètre

Les méthodes de la classe *Tests* (2)

➡ Design

- Appel obligatoire avant description premier *test case*
- Identification du *test design* et niveau de détail de la trace en visualisation en paramètres

➡ Case

- Appel obligatoire avant description premier *test unit*
- Identification du *test case* en paramètre

Les méthodes de la classe *Tests* (3)

- Unit

  - Valeur attendue (type prédéfini) et valeur constatée (type prédéfini) en paramètres

Création du contexte d'exécution (1)

Le contexte d'exécution d'un test unitaire est constitué de l'ensemble des objets supports exploités par le test.

Le contexte d'exécution comprend :

- le ou les objets qui définissent **le résultat attendu**
- le ou les objets nécessaires à l'élaboration (par la méthode cible) du **résultat constaté**

Création du contexte d'exécution (2)

La création du contexte d'exécution d'un test est entièrement à la charge du programmeur

Les méthodes de la classe *Tests* n'automatisent que :

- la comparaison entre chaque résultat attendu et le résultat constaté correspondant
- la propagation du compte rendu vers les niveaux de tests supérieurs (*test case* et *test design*)
- visualisation de tous les comptes rendus

Création du contexte d'exécution (3)

- Les blocs d'instruction qui créent le contexte d'exécution de chaque *test unit* ne doivent avoir aucune dépendance structurelle ou fonctionnelle entre eux.
- Des blocs d'instruction de niveau supérieur (*test case* ou *test design*) peuvent être utiles pour créer une partie commune (factorisation) à des contextes de niveau inférieur.

Création du contexte d'exécution (4)

- Dans l'hypothèse de blocs d'instructions de niveaux supérieurs, les instructions de niveaux inférieurs ne doivent pas modifier les objets communs (principe d'indépendance des *test_unit*)

Problème :

Comment concilier la dernière règle avec le test des méthodes qui modifient l'objet cible (accesseurs de modification par exemple) ?

Exemple de mise en oeuvre (1)

```
void main () {
   Ville* pNice    = new Ville ("Nice", 375000);
   Ville* pNantes = new Ville ("Nantes", 250000);


   Tests::Begin ("Ville", "1.0.0");
      Tests::Design ("Toutes les fonctions", 3);
         Tests::Case (”Accesseurs de consultation");
            Tests::Unit ("NICE", pNice->getNom());
            Tests::Unit (345000, pNice->getPopulation());
```

Exemple de mise en oeuvre (2)

```
---
        Tests::Unit ("NANTES", pNantes->getNom());
        Tests::Unit (250000, pNantes->getPopulation());

    Tests::Case (”Accesseurs de modification");
         pNice->setPopulation(375000);
        Tests::Unit (375000, pNice->getPopulation());

 Tests::End();
}
```

Bibliographie

Méthodologie ECSS – E.S.A

Note technique sur la charte des tests unitaires