

Les tests fonctionnels

- 1 Principes généraux
- 2 Les tests statiques
- 3 Introduction aux tests dynamiques
- 4 Les tests fonctionnels
- 5 Les tests structurels
- 6 Les tests et les langages objets
- 7 Le TDD

Le principe

4

Les tests fonctionnels

- Le principe
- Les tests exhaustifs
- Les tests aléatoires
- Les tests aux limites
- Les tests partitionnels
- Un exemple de test unitaire
- Test fonctionnel d'une classe
- Combinaisons logiques

Les tests fonctionnels

Principe

- Tests dynamiques
- Reposent sur la spécification
- Tests boîte noire : pas besoin de connaître le code

Quelques tests fonctionnels

- Test exhaustif
- Test aléatoire
- Test aux limites
- Test partitionnel

Les tests exhaustifs

4

Les tests fonctionnels

- Le principe
- **Les tests exhaustifs**
- Les tests aléatoires
- Les tests aux limites
- Les tests partitionnels
- Un exemple de test unitaire
- Test fonctionnel d'une classe
- Combinaisons logiques

Les tests exhaustifs

Principe

- Consistent à tester toutes les valeurs possibles en entrée
- Le plus complet des tests
- Rarement applicable

Les tests exhaustifs

Exemple

Soit une fonction à tester acceptant deux entiers sur 32 bits en paramètres.

- 2^{32} possibilités par entiers, soit

2^{64} possibilités

- Si un test prend une milliseconde, cela nécessitera plus de

100 millions d'années

Les tests exhaustifs

Applicables si la spécification ne fait apparaître que :

- Des booléens
- Des types énumérés (non implémentés à l'aide d'entiers)
- Quelques octets.

Les tests exhaustifs

Exemple

Spécification de la fonction `carre` :

pour tout entier x , `carre(x)` renvoie $x * x$.

Test

On suppose ici que les entiers sont codés sur 2 octets.

```
void main() {  
    const int min = -32768;  
    const int max = +32767;  
    for (int i = min ; i <= max ; i++)  
        assert(carre(i) == i * i );  
}
```

Que se passe-t-il lors de l'exécution de la boucle si les entiers sont effectivement codés sur deux octets ?

Les tests exhaustifs

Test (bis)

```
#include <limits.h>
#include <assert.h>

void main() {
    const int min = SHRT_MIN;
    const int max = SHRT_MAX;

    for (int i = min ; i < max ; i++)
        assert(carre(i) == i * i );
    assert(carre(max) == max * max );
}
```

exhaustif : exhaustif.c :23 : main : Assertion 'carre(i) == i * i'
failed. Quel est le type de retour de la fonction carre ?

Les tests exhaustifs

Test (ter)

```
#include <limits.h>
#include <assert.h>

int main() {
    const int min = INT_MIN;
    const int max = INT_MAX;

    for (int i = min ; i < max ; i++) {
        assert(carre(i) == (long)i * i );
    }
    assert(carre(max) == (long)max * max);

    return EXIT_SUCCESS;
}
```

Sur cet ordinateur, il faut 16 secondes pour tester tous les entiers (sur 32 bits, avec l'option -O1).

Les tests exhaustifs

Test (le même en Java)

```
class Exhaustif {
    public static void main (String [] args){
        int min = Integer.MIN_VALUE;
        int max = Integer.MAX_VALUE;

        for (int i = min; i < max; i++) {
            assert Exhaustif.carre(i) == (long) i * i + 2L * i + 1;
        }
        assert Exhaustif.carre(max) == (long) max * max + 2L * max + 1;
    }

    public static long carre(int i) {
        return (i + 1L) * (i + 1L);
    }
}
```

On ne calcule pas le carré car alors l'optimiseur Java "omet" les tests.

Les tests exhaustifs

Test (le même en Java)

```
class Exhaustif {
    public static void main (String [] args){
        int min = Integer.MIN_VALUE;
        int max = Integer.MAX_VALUE;

        for (int i = min; i < max; i++) {
            assert Exhaustif.carre(i) == (long) i * i + 2L * i + 1;
        }
        assert Exhaustif.carre(max) == (long) max * max + 2L * max + 1;
    }

    public static long carre(int i) {
        return (i + 1L) * (i + 1L);
    }
}
```

L'exécution du programme requière environ 10s, pour tester deux paramètres entiers cela prendrait donc $2^{32} \times 10s$, comme il y a un peu moins de 2^{35} secondes dans une année, cela représente plus de 1280 années.

Les tests exhaustifs

Test d'une énumération

```
public enum Couleur {
    ROUGE("1\u20220\u20220"), VERT("0\u20221\u20220"), BLEU("0\u20220\u20221");

    private String rgb;

    Couleur(String rgbValues) {
        this.rgb = rgbValues;
    }

    public String getRGB() {
        return this.rgb;
    }
}
```

Les tests exhaustifs

Test d'une énumération

```
class TestExhaustif {  
    public static void main (String [] args){  
        assert Couleur.ROUGE.getRGB() == "1\u00000";  
        assert Couleur.VERT.getRGB() == "0\u00001";  
        assert Couleur.BLEU.getRGB() == "0\u000001";  
    }  
}
```

Les tests aléatoires

4

Les tests fonctionnels

- Le principe
- Les tests exhaustifs
- **Les tests aléatoires**
- Les tests aux limites
- Les tests partitionnels
- Un exemple de test unitaire
- Test fonctionnel d'une classe
- Combinaisons logiques

Les tests aléatoires

Définition

Les tests aléatoires génèrent les données de test de façon aléatoire en se basant, par exemple, sur un échantillonnage uniforme des domaines de définition des données en entrée.

Les tests aléatoires

Avantages

- Permet de générer des tests automatiques
- On peut générer des statistiques sur les défaillances détectées
- Bonne objectivité des données de test

Les tests aléatoires

Inconvénients

- Il faut être capable de déterminer le résultat attendu
- Il ne faut pas qu'il y ait des cas forts peu probables importants
- Le rapport résultat/effort a tendance à “plafonner” là où d'autres ont un comportements plus linéaires

Les tests aléatoires

Exemple

Soit à tester un programme d'inversion de matrice de taille entre 2 et 10.

- On produit des tailles au hasard
- On remplit les matrices au hasard aussi
- On vérifie que le produit du résultat avec la matrice en entrée est égal à la matrice identité

Les tests aléatoires

(Mauvais) exemple

Un test aléatoire ne serait pas satisfaisant pour la fonction
int egal(float x, float y); qui renvoie $x = y$ si on considère comme domaine de définition celui des flottants.

Générateur aléatoire

Comment tester un générateur de nombre aléatoire

Comment nous y prendrions nous, si on voulait tester la méthode Math.random de l'API Java ?

Rappel

public static double Math.random() retourne une valeur comprise dans l'intervalle $[0, 1[$.

Les valeurs rentrées sont choisies pseudo-aléatoirement avec une distribution (approximativement) uniforme sur l'intervalle.

Générateur aléatoire

Réponse ? Non

```
class TestRandom {  
    public static void main(String[] args) {  
        double somme = 0;  
        int num = 100000;  
  
        for (int i = 0; i < num; i++) {  
            somme += Math.random();  
        }  
        double moyenne = somme / num;  
        assert Math.rint(moyenne * 100) == 50 :  
            "Test échoué";  
    }  
}
```

Il faudrait utiliser, par exemple, le test de Kolmogorov-Smirnov.

Le test de Kolmogorov-Smirnov

- On génère n nombres $x_{i,i \in [1,n]}$ aléatoirement et on les trie par ordre croissant
- On calcule $D^+ = \max_{1 \leq i \leq n} \left\{ \frac{i}{n} - x_i \right\}$ et $D^- = \max_{1 \leq i \leq n} \left\{ x_i - \frac{i-1}{n} \right\}$
- On calcule $D = \max\{D^+, D^-\}$ et on le compare à une valeur critique définie par rapport à n

Générateur aléatoire

Le test de Kolmogorov-Smirnov

```
class TestKolmogorovSmirnov {  
    ...  
    public static boolean testEchantillon(double[] tab) {  
        double Dplus = 0, Dmoins = 0;  
        for (int i = 1; i <= N; i++) {  
            double dplus = (double)i / N - tab[i - 1];  
            double dmoins = tab[i - 1] - (double)(i - 1) / N;  
            if (dplus > Dplus) Dplus = dplus;  
            if (dmoins > Dmoins) Dmoins = dmoins;  
        }  
        double D = Dplus > Dmoins ? Dplus : Dmoins;  
        double Da_n = a_n / Math.sqrt(N);  
        return D <= Da_n;  
    }  
    ...
```

Comment réaliser un test du générateur à l'aide du test de Kolmogorov-Smirnov ? Est-ce satisfaisant ?

Les tests aux limites

4

Les tests fonctionnels

- Le principe
- Les tests exhaustifs
- Les tests aléatoires
- **Les tests aux limites**
- Les tests partitionnels
- Un exemple de test unitaire
- Test fonctionnel d'une classe
- Combinaisons logiques

Les tests aux limites

Principe

Permet de vérifier que le programmeur a bien réalisé le code pour les valeurs limites :

- extrémité d'intervalle
- valeur nulle
- liste vide
- égalité de deux données
- valeur marginale prévue par la spécification

Les tests aux limites

Définition

Le test aux limites consiste à identifier, pour chaque variable en entrée ou en sortie, ses valeurs limites. Ensuite on les combine pour faire des jeux d'essais.

Les tests aux limites

Exemples de valeurs limites

- $[G, D] \subset \mathbb{R}$, soit d la plus petite valeur représentable. On génère $G - d, G + d, D - d, D + d$.
- Si la variable est un ensemble ordonné de valeurs, choisir le premier, le second, l'avant-dernier, le dernier.

Les tests aux limites

Exemples

Écrire un programme analysant un fichier d'étudiants d'au maximum 100 enregistrements. Chaque enregistrement comporte le nom (20 car. = lettres), le sexe (1 : M, F), 5 notes (2 chiffres : entier de 0 à 20). Il calcule la moyenne par étudiants, par sexe et par matière, nombre de réussites (moyenne ≥ 10).

Limites

- fichier vide, 1 enr., 99, 100, 101.
- nom vide, car. ctl, 19, 20, 21.
- code irrégulier de sexe (rien, "N", 2 car.).
- pas de note, une note, 5 notes, 6 notes.
- notes -1, 0, 1, 19, 20, 21.

Rappels de représentation

Les entiers

- entier naturel : n bits, $[0, 2^n - 1]$.
- entier relatifs : représentation complément à deux, un bit de signe, $(n-1)$ bits de valeur.
 - ≥ 0 : valeur de sa représentation
 - < 0 : on complémente puis on ajoute 1.

Limites : $[10 \dots 0]_2$, $[11 \dots 1]_2$, $[00 \dots 0]_2$,
 $[00 \dots 01]_2$, $[01 \dots 1]_2$

Rappels de représentation

Les entiers

- entier naturel : n bits, $[0, 2^n - 1]$.
- entier relatifs : représentation complément à deux, un bit de signe, $(n-1)$ bits de valeur.
 - ≥ 0 : valeur de sa représentation
 - < 0 : on complémente puis on ajoute 1.

Limites : $[10 \dots 0]_2 = -2^{n-1}$, $[11 \dots 1]_2$, $[00 \dots 0]_2$,

$[00 \dots 01]_2$, $[01 \dots 1]_2$

Rappels de représentation

Les entiers

- entier naturel : n bits, $[0, 2^n - 1]$.
- entier relatifs : représentation complément à deux, un bit de signe, $(n-1)$ bits de valeur.
 - ≥ 0 : valeur de sa représentation
 - < 0 : on complémente puis on ajoute 1.

Limites : $[10 \dots 0]_2 = -2^{n-1}$, $[11 \dots 1]_2 = -1$, $[00 \dots 0]_2$,
 $[00 \dots 01]_2$, $[01 \dots 1]_2$

Rappels de représentation

Les entiers

- entier naturel : n bits, $[0, 2^n - 1]$.
 - entier relatifs : représentation complément à deux, un bit de signe, $(n-1)$ bits de valeur.
 - ≥ 0 : valeur de sa représentation
 - < 0 : on complémente puis on ajoute 1.
- Limites : $[10 \dots 0]_2 = -2^{n-1}$, $[11 \dots 1]_2 = -1$, $[00 \dots 0]_2 = 0$,
 $[00 \dots 01]_2$, $[01 \dots 1]_2$

Rappels de représentation

Les entiers

- entier naturel : n bits, $[0, 2^n - 1]$.
 - entier relatifs : représentation complément à deux, un bit de signe, $(n-1)$ bits de valeur.
 - ≥ 0 : valeur de sa représentation
 - < 0 : on complémente puis on ajoute 1.
- Limites : $[10 \dots 0]_2 = -2^{n-1}$, $[11 \dots 1]_2 = -1$, $[00 \dots 0]_2 = 0$,
 $[00 \dots 01]_2 = 1$, $[01 \dots 1]_2$

Rappels de représentation

Les entiers

- entier naturel : n bits, $[0, 2^n - 1]$.
- entier relatifs : représentation complément à deux, un bit de signe, $(n-1)$ bits de valeur.
 - ≥ 0 : valeur de sa représentation
 - < 0 : on complémente puis on ajoute 1.

Limites : $[10 \dots 0]_2 = -2^{n-1}$, $[11 \dots 1]_2 = -1$, $[00 \dots 0]_2 = 0$,
 $[00 \dots 01]_2 = 1$, $[01 \dots 1]_2 = 2^{n-1} - 1$

Rappels de représentation

Les réels

La norme IEEE définit une représentation possible des réels, il s'agira alors de flottants simple précision. On utilise 32 bits :

$$s e_1 \dots e_8 m_1 \dots m_{23}.$$

- s est le bit de signe ;
- e est l'exposant, il est stocké avec un excès de 127, 0 et 255 sont interdits (0 et NaN) ;
- m est la mantisse, elle est normalisée ($= 1, m_1 \dots m_{23}$).

Ainsi

$$[d_1 d_2 \dots d_{32}] = (-1)^{d_1} 2^{d_2 \dots d_9} 2^{-127} [1, d_{10} \dots d_{32}]_2.$$

Limites des flottants

Exemple

```
int main(int argc, char **argv) {
    float min, max;

    /* Le max représentable (little endian) */
    ((unsigned char*)&max)[0]=255;
    ((unsigned char*)&max)[1]=255;
    ((unsigned char*)&max)[2]=127;
    ((unsigned char*)&max)[3]=127;

    /* Le min représentable */
    ((unsigned char*)&min)[0]=0;
    ((unsigned char*)&min)[1]=0;
    ((unsigned char*)&min)[2]=128;
    ((unsigned char*)&min)[3]=0;

    return 0;
}
```

Représentation des réels par des flottants

Test aux limites : exemple

Quels sont les flottants encadrant 10 000 représenté par un flottant ?

10 000 est représenté par

0 10001100 001110001000000000000000

(ce qui donne $2^{13} \times 1,220703125$.)

Les deux flottants encadrant 10 000 sont donc

0 10001100 0011100010000000000001 et

0 10001100 0011100001111111111111,

soit 10000,000977 et 9999,999023.

Représentation des réels par des flottants

La limite inférieure

Le plus petit réel positif représentable est :

0 00000001 00000000000000000000000000000001,

soit $1,175494 \times 10^{-38}$ (et $1,401298 \times 10^{-45}$ si on autorise un exposant nul).

Des « trous » dans la représentation

100 000 000 000 est représenté par

0 10100011 01110100100001110110111,

soit 99 999 997 952 ! L'entier suivant est 100 000 006 144.

Exemple

Attention à la précision des flottants

Le programme suivant échouera :

```
int main(int argc, char **argv) {
    float f;
    double d;

    f = 100000000000;
    d = 100000000000;

    assert(f == d);

    return 0;
}
```

```
[toto@localhost prg]#./test_float_double
test_float_double: test_float_double.c:15: main:
Assertion `f == d' failed.
Abandon (core dumped)
```

En pratique ...

La classe Integer

En Java, la classe Integer permet d'obtenir la plus petit et le grand entier représentable : Integer.MIN_VALUE et Integer.MAX_VALUE

La classe Float

La classe Float permet d'obtenir :

- La plus grande valeur positive : Float.MAX_VALUE ;
- La plus petite valeur positive : Float.MIN_VALUE ;
- La valeur associée à « not a number » : Float.NaN ;
- La valeur associée – inf : Float.NEGATIVE_INFINITY ;
- La valeur associée + inf : Float.POSITIVE_INFINITY.

Les tests partitionnels

4

Les tests fonctionnels

- Le principe
- Les tests exhaustifs
- Les tests aléatoires
- Les tests aux limites
- **Les tests partitionnels**
- Un exemple de test unitaire
- Test fonctionnel d'une classe
- Combinaisons logiques

Les tests partitionnels

Définition

Les données en entrée se divisent en classes d'équivalence selon la spécification. On teste une donnée par classe, y compris les classes non valides.

Définition

Une classe est dite non valide si les données de cette classe ne devraient pas normalement être fournies en entrée.

Les tests partitionnels

Exemple

Soit la fonction racine qui pour tout $x \geq 0$ doit retourner y tel que $|y^2 - x| < e$ avec $e = 10^{-6}$.

Les réels positifs constituent les données valides et les réels strictement négatifs forment les données non valides.

Même spécification avec en plus : retourner 0 pour $x = 0$.

Il y a alors trois classes d'équivalences.

Les tests partitionnels

Définition des classes d'équivalences

- si donnée = intervalle
 - inférieures, supérieures
 - 1 classe valide
- si donnée = tableau à N éléments
 - tableaux à moins de N éléments, plus de N éléments
 - 1 classe valide

Les tests partitionnels

Définition des classes d'équivalences

- si donnée = ensemble
 - vide, trop
 - 1 classe valide
- si donnée = énumération
 - une classe non valide (si c'est possible)
 - une classe valide par valeur
- si donnée = booléen : obligation, contrainte (forme, syntaxe, sens)
exemple : une présence est détectée en moins de 5 secondes.
 - une classe respectée
 - une classe non respectée

Un exemple de test unitaire

4

Les tests fonctionnels

- Le principe
- Les tests exhaustifs
- Les tests aléatoires
- Les tests aux limites
- Les tests partitionnels
- **Un exemple de test unitaire**
- Test fonctionnel d'une classe
- Combinaisons logiques

Un exemple

Spécification

Soit la fonction

```
std::list < std::complex >
resolv(const float a, const float b, const float c);
```

qui renvoie la liste des solutions à l'équation $ax^2 + bx + c = 0$.

Entrées / sorties

- On a trois entrées : a, b, c des flottants simple précision « constants ».
- Une sortie : une liste de « complex ».

Analyse partitionnelle

Partition des entrées

Il faut tester les entrées valides (par exemple 1) et les entrées non valides (NaN) pour chacune des entrées.

$a = \text{NaN}$	V	V	V	V	F	F	F	F
$b = \text{NaN}$	V	V	F	F	V	V	F	F
$c = \text{NaN}$	V	F	V	F	V	F	V	F

Une analyse uniquement sur le types des entrées n'est pas pertinente.

Analyse partitionnelle

Partition des entrées

Coefficients des monômes :

- $a \neq 0$: équation du second degré.
- $a = 0, b \neq 0$: équation du 1er degré.
- $a = 0, b = 0$: équation constante.

Valeur du discriminant :

- $\Delta = 0$: une racine double réelle.
- $\Delta > 0$: deux racines réelles.
- $\Delta < 0$: deux racines imaginaires.

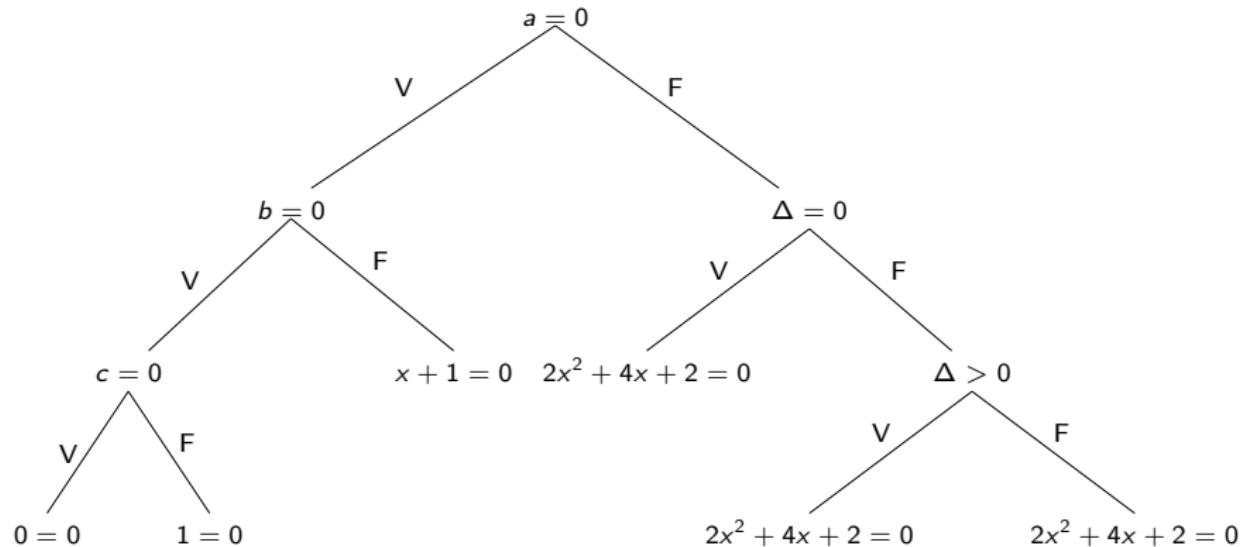
Un tableau pour récapituler

coef.	Δ	Données	Résultats
$a \neq 0$	$= 0$	$2x^2 + 4x + 2 = 0$	-1
$a \neq 0$	> 0	$2x^2 + 5x + 2 = 0$	$-\frac{1}{2}, -2$
$a \neq 0$	< 0	$x^2 - 2x + 2 = 0$	$1+i, 1-i$
$a = 0, b \neq 0$	/	$x + 1 = 0$	-1
$a = 0, b = 0$	/	$4 = 0$	\emptyset

Et l'équation $0 = 0$???

Comment réaliser un oracle ? Comment tester qu'une liste contient exactement les éléments voulus ?

Un arbre de décision pour récapituler



Test fonctionnel d'une classe

4

Les tests fonctionnels

- Le principe
- Les tests exhaustifs
- Les tests aléatoires
- Les tests aux limites
- Les tests partitionnels
- Un exemple de test unitaire
- **Test fonctionnel d'une classe**
- Combinaisons logiques

Test d'une classe

Tester une classe c'est tester chaque méthode de la classe.

Mais comment tester :

- une méthode qui prend des paramètres mais ne renvoie rien ;
- une méthode qui ne prend pas des paramètres mais renvoie une valeur dépendant de l'état de l'instance ;
- une méthode qui renvoie une valeur dépendant de ses paramètres mais aussi de l'état de l'instance ?

Inspectrices / modifcatrices

Définition

On classe les différentes méthodes d'une classe en trois catégories :

- les modifcatrices qui changent l'état de l'instance ;
- les inspectrices qui nous renseignent sur l'état de l'instance ;
- les inspectrices/modifcatrices qui modifient l'état de l'instance et nous renvoient des informations sur son état.

Test fonctionnel d'une classe

État d'une instance

Lors du test d'une classe, c'est l'état de l'instance qui nous intéresse. En règle général, cet état ne nous est pas directement accessible : ce sont les inspectrices qui nous renseignent sur l'état de l'instance.

Test fonctionnel d'une classe

Tests

- Pour tester une inspectrice, on part d'un état d'une instance et on vérifie que l'inspectrice renvoie une valeur en accord avec cet état.
- Pour tester une modificatrice, on part d'un état d'une instance et on vérifie que l'état en sortie concorde avec l'état attendu.

Corollaire

On se sert donc des inspectrices pour vérifier que l'état suivant une modificatrice concorde à l'état espéré \implies les inspectrices sont testées lors du test des modificatrices.

Un exemple

La classe Pile

```
class Pile {
public:
    Pile();
    void empiler(int i);
    void depiler();
    int sommet();
    int hauteur();
private:
    ...
}
```

Un exemple

Inspectrices

- int sommet();
- int hauteur();

Modificatrices

- void empiler(int i);
- void depiler();
- Pile();

Test de Pile();

Un oracle pour le constructeur

```
Pile p;  
assert( p.hauteur() == 0 );
```

Test de empiler();

La méthode `empiler()`; prend un paramètre, mais la valeur du paramètre ne joue aucun rôle dans l'état de l'instance.

Tests

- Test partitionnel : un seul cas, par exemple l'état pile vide.
- Test aux limites : pile vide, une valeur empilée, beaucoup de valeurs.
- Test aléatoire : un nombre aléatoire de valeur empilées.

Test de empiler();

Le squelette du test

```
int h = p.hauteur();
p.empiler(v);
assert(p.sommet() == v);
assert(p.hauteur() == h + 1);
```

Test de empiler();

Un test plus complet pour vérifier l'état de la pile

```
int i;  
for ( i = 0 ; i < beaucoup ; i++ )  
    p.empiler( i );  
  
int h = p.hauteur();  
p.empiler(v);  
assert(p.sommet() == v);  
assert(p.hauteur() == h + 1);  
p.depiler();  
  
while ( i > 0 ) {  
    p.depiler();  
    assert( p.sommet() == i - 1 );  
    i--;  
}
```

Test de dépiler();

- Même principe que empiler();
- le test de empiler(); teste aussi dépiler();

Test des inspectrices

Les deux inspectrices hauteur(); et sommet(); sont également testées lors du test de empiler();

Un test aux limites

```
int i, h, beaucoup=100;
Pile p;

assert( p.hauteur() == 0 );
for ( i = 0 ; i < beaucoup ; i++ ) {
    p.empiler( i );
    assert( p.sommet() == i );
    assert( p.hauteur() == i + 1 );
}
while ( i > 1) {
    i--;
    p.depiler();
    assert( p.sommet() == i - 1 );
    assert( p.hauteur() == i );
}
p.depiler();
assert( p.hauteur() == 0 );
```

Qu'est-ce qui manque ?

Qu'est-ce qui manque ?

On n'a pas testé les cas d'utilisations illicites : dépiler une pile vide ou récupérer le sommet d'une pile vide.

La classe Pile

```
class Pile {  
    public:  
        Pile();  
        void empiler(int i);  
        void depiler() throw(PileVide);  
        int sommet() throw(PileVide);  
        int hauteur();  
    private:  
        ...  
}
```

Test des cas illicites

Utilisation anormale

```
assert( p.hauteur() == 0 );
try { p.depiler(); assert(false); }
catch (PileVide) { assert( p.hauteur() == 0 ); }
catch ( ... ) { assert( false ); }

try { p.sommet(); assert(false); }
catch (PileVide) { assert( p.hauteur() == 0 ); }
catch ( ... ) { assert(false); }
```

Test fonctionnel d'une classe

4

Les tests fonctionnels

- Le principe
- Les tests exhaustifs
- Les tests aléatoires
- Les tests aux limites
- Les tests partitionnels
- Un exemple de test unitaire
- Test fonctionnel d'une classe
- Combinaisons logiques

Combinaison logique

Cas d'application

La sortie est fonction d'une expression logique composée de propositions liants les variables de l'entrée.

Modèle de faute

Un tel test peut être erroné à cause :

- d'erreurs dans l'interprétation des conditions d'un algorithme ;
- d'erreurs logiques (mauvais connecteurs logique, inversions).

Exemple

Spécification

Dans la colonne 1 il doit y avoir 'M' ou 'F' et dans la colonne 2 un naturel. Si c'est le cas émettre le message OK. Si le premier caractère est erroné, émettre le message M1, s'il n'y a pas de naturel émettre le message M2.

Causes-Effets

Cause

- C1 : 'M' en colonne 1
- C2 : 'F' en colonne 1
- C3 : naturel en colonne 2

Effets

- OK
- M1
- M2

Causes-Effets

Définition

Un variant est une combinaison unique des variables (booléennes) d'entrée et des variables de sortie (*i.e. une ligne dans la table de vérité correspondant à l'expression*).

Si on veut tester tous les cas, on a au pire 2^n variants différents.

Exemple

Table de vérité

C1	C2	C3	OK	M1	M2
0	0	0	0	1	1
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	0	0
1	0	0	0	0	1
1	0	1	1	0	0

Réalisation du test

Pour réaliser le test, il faut :

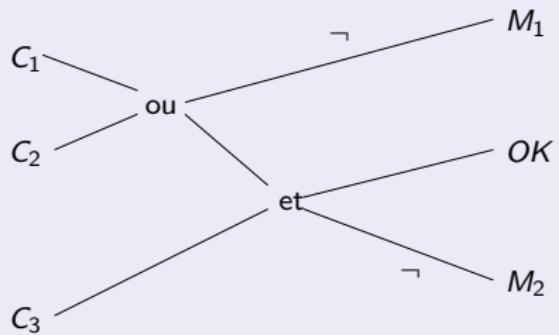
- construire une représentation manipulable de l'expression logique : cette expression doit être la plus simple possible pour limiter la génération de cas de test ;
- appliquer un critère de test (stratégie de test) ;
- sélectionner les valeurs d'entrées permettant de réaliser le test.

Représentations testables

Modèles permettant d'appliquer une stratégie de sélection

- table de décision : chaque colonne est une variable d'entrée ou de sortie (on suppose que toutes les variables sont modélisées comme des booléens), chaque ligne est une combinaison
- arbre : chaque nœud de l'arbre est une variable, chaque branche est un état de cette variable, les feuilles sont les valeurs de variables de sorties correspondant
- graphe de causes-effets : les nœuds de départ sont les variables d'entrée, les nœuds de sortie sont les variables de sortie, les nœuds intermédiaires sont des opérateurs booléens
- matrices de Karnaugh
- BDD

Graphe de causes-effets : exemple



Remarque

Il ne manque pas un cas ?

Cas particuliers

- **don't care** : la valeur d'une variable n'a pas d'importance dans le cas considéré. Permet de simplifier une expression ou une table de décision
- **can't happen** : un cas est impossible (cas des variables liées, si plusieurs variables propositionnelles utilisent une même variable d'entrée avec des valeurs différentes, appelé exclusion mutuelle sûre). Attention aux faux sentiments de sécurité...
- **don't know** : cas non prévu par la spécification → **PROBLÈME.**

Critères de couvertures

- tous les variants : 2^n dans le pire des cas, impraticable au delà de quelques variables
- tous les variants explicites
- tous les termes vrai/faux
- MC/DC
- Pairwise testing

Modified Condition / Decision Coverage

Critère de test classique, applicable pour les expressions logiques :

- chaque valeur possible de chaque variable de l'expression doit être testé au moins une fois (condition coverage) ;
- chaque valeur possible de l'expression doit être testé au moins une fois (decision coverage) ;
- chaque valeur possible d'une variable produisant un résultat de décisions différent indépendamment des autres variables doit être testé ;

Exemple

$A \vee B$

- condition coverage : (1, 0), (0, 1)
- decision coverage : (1, 0), (0, 0)
- decision et condition : (1, 1), (0, 0)
- M C/D C : (1, 0), (0, 0), (0, 1)

Le principe

Identifier pour chaque variable en entrée les conditions nécessaires pour que, toutes les autres variables étant fixées (toutes choses égales par ailleurs), la modification de cette variable se reflète dans le résultat prévu.

Représentation d'un graphe : construction d'un BDD

Binary Decision Diagram

Représentation compacte d'une expression booléenne. Un diagramme de décision binaire représente tous les états possibles d'un ensemble de variables logiquement liées (ou non). Un BDD est un graphe dirigé acyclique dont les nœuds sont des variables logiques et les arcs des valeurs logiques vrai ou faux. Les (deux) nœuds terminaux du graphe sont les valeurs vrai et faux.

Les BDD

Propriétés

- Canonicité par rapport à un certain ordre total des variables : tout chemin de la racine du BDD à une feuille respecte un ordre total des variables
- minimalité : chaque nœud est unique au sens où le triplet $(var, low, high)$ est unique dans le BDD.

Les BDD

Tous les BDD d'une expression ne sont pas égaux. Trouver le BDD minimal est un problème NP-complet.

Intérêt

Réduit le problème de la sélection des D.T. à un parcours de tous les chemins d'un graphe.

MC/DC vs BDD

Pour toute expression logique, le nombre de cas de test pour les critères “tous les chemins du BDD” et “MC/DC” est équivalent.

Pairwise testing

Modèle de faute

Les défauts dépendent de l'interaction d'au plus 2 (ou n) paramètres, pas de la combinaison de tous les paramètres.

Applicabilité

Le comportement de l'AAT dépend de la combinaison d'un ensemble de paramètres à valeurs finies : drapeaux, énumérations, plages de valeurs ou partitions de domaines. La taille du domaine et le nombre de paramètres doivent être dans des proportions raisonnables. Le nombre réel de configurations à tester en combinant toutes les possibilités est trop important.

Pairwise testing

Principe

Critère de test basé sur l'hypothèse réaliste que les défauts sont causés par l'interaction d'au plus deux facteurs (paramètres).

Permet de réduire l'explosion combinatoire pour la sélection des cas de tests lorsqu'un comportement dépend de plusieurs variables dont le domaine est fini.

- Aussi appelé méthode de tableaux orthogonaux.
- Généralisation du principe MC/DC

Carrés latins et gréco-latins

Définition

Un **carré latin** d'ordre n est un tableau carré à n^2 cases où sont répartis n nombres ou symboles de manière telle qu'aucun n'apparaisse deux fois dans une même ligne ou dans un même colonne.

Exemple

a	b	c	d	e	f
c	d	e	f	a	b
e	f	a	b	c	d
b	c	d	e	f	a
d	e	f	a	b	c
f	a	b	c	d	e

Carrés latins et gréco-latins

Définition

Un **carré gréco-latin** d'ordre n est un tableau de couples obtenu en superposant deux carrés latins d'ordre n de manière à ce que les n^2 couples obtenus soient distincts deux à deux.

Exemple

αa	βb	γc	δd
γb	δa	αd	βc
δc	γd	βa	αb
βd	αc	δb	γa

Test

On utilise des carrés gréco-latins pour trouver des données de test répondant au critère :

- on suppose que tous les k paramètres ont la même taille n
- on isole deux paramètres qui seront les indices d'une matrice $n \times n$
- pour chacun des $k - 2$ paramètres restant, on construit un carré latin tel que chacune des superpositions de deux carrés latins forme un carré gréco-latin.

Exemple

Avec 4 paramètres de taille 3, on peut choisir les deux carrés latins suivants :

a	b	c
b	c	a
c	a	b

et

α	β	γ
γ	α	β
β	γ	α

Exemple

D.T.

Ce qui nous donne 9 données de test :

- $(1, 1, a, \alpha)$
- $(1, 2, b, \beta)$
- $(1, 3, c, \gamma)$
- $(2, 1, b, \gamma)$
- $(2, 2, c, \alpha)$
- $(2, 3, a, \beta)$
- $(3, 1, c, \beta)$
- $(3, 2, a, \gamma)$
- $(3, 3, b, \alpha)$

au lieu de $3^4 = 81$ pour un test exhaustif.

Pairwise testing

Comment traiter les variables n'ayant pas le même nombre de valeurs possibles :

- utiliser la taille maximale pour le calcul du tableau orthogonal
- modifier le mapping de telle sorte que plusieurs valeur du tableau pointent vers les mêmes valeurs concrètes
- simplifier les cas de test : en remplaçant des valeurs, on tombe souvent sur des redondances

Pairwise testing

Mise en œuvre

- Identifier les différentes variables et leurs valeurs possibles ;
- mapper ces valeurs sur une séquence (nombre, lettre) éventuellement avec le même nombre de valeurs pour chaque variables ;
- construire un tableau orthogonal couvrant avec les valeurs arbitraires ;
- remapper sur des valeurs concrètes ;
- construire les cas de test correspondants (*i.e.* l'oracle)

Pairwise testing

Mise en œuvre

- Identifier les différentes variables et leurs valeurs possibles ;
- mapper ces valeurs sur une séquence (nombre, lettre) éventuellement avec le même nombre de valeurs pour chaque variables ;
- construire un tableau orthogonal couvrant avec les valeurs arbitraires ;
- remapper sur des valeurs concrètes ;
- construire les cas de test correspondants (*i.e.* l'oracle)

Outils

jenny est un outil qui peut être utilisé pour calculer les k -uplets :
<http://burtleburtle.net/bob/math/jenny.html>

Interprétation sous forme de graphe

Chaque valeur de paramètre est un nœud d'un graphe $G(V, E)$.
Pour k paramètres, le graphe G est k -partie complet :

- l'ensemble des nœuds est partitionné en k sous-ensembles deux à deux disjoints ;
- chaque nœud possède un arc vers tous les autres nœuds qui n'appartiennent pas à sa propre partition

Interprétation sous forme de graphe

Les arcs représentent donc les couples de valeurs possibles. S'il y a des couples impossibles, les arcs sont supprimés.

Un cas de test est donc un chemin dans le graphe tel que chaque partition est représentée une et une seule fois. Une suite de test est adéquate pour le critère toutes les paires si tous les arcs du graphe sont couverts par au moins un chemin.