

# Cours de Test Logiciel

## Leçon 4 : Tests de régression

Sébastien Bardin

CEA-LIST, Laboratoire de Sûreté Logicielle

`sebastien.bardin@cea.fr`

`http://sebastien.bardin.free.fr`

**Tests de régression** : à chaque fois que le logiciel est modifié, s'assurer que "ce qui fonctionnait avant fonctionne toujours"

---

Pourquoi modifier le code déjà testé ?

- correction de défaut
- ajout de fonctionnalités

Quand ?

- en phase de maintenance / évolution
- ou durant le développement

Quels types de test ?

- tous : unitaires, intégration, système, etc.

## Évolution / maintenance des tests

- les tests (DT/oracle) sont-ils encore valides (format I/O, oracle) ?
- tests invalides → refaire des tests (surcoût)

## Sélection / minimisation des tests à rejouer

- tout rejouer à chaque modification = extrêmement coûteux
- ne rien rejouer = suicidaire
- sélectionner un sous-ensemble pertinent de tests à rejouer

- Contexte
- Maintenance des tests
- JUnit
- Sélection
- Minimisation / Prioritisation
- Discussion

Objectif : avoir une méthode automatique pour

1. rejouer automatiquement les tests
2. détecter les tests dont les DT / oracle ne sont plus syntaxiquement corrects
3. détecter les tests dont l'oracle n'est plus sémantiquement correct
4. corriger automatiquement les tests

Nous verrons des solutions automatisées aux points 1 et 2.

## Script-based testing

- tests écrits dans un langage exécutable (langage de programmation, langage de script, langage dédié (TTCN-3))
- simplifie l'exécution et le rejeu des tests (juste tout relancer)
- simplifie la détection d'une partie des tests non à jour : erreurs à la compilation, tests recompilés en même temps que le programme
- simplifie le stockage et la réutilisation des tests (ex : tests de MyClass dans MyClassTest)
- exemple : JUnit pour Java, TTCN-3

Voir JUnit dans la suite du cours et en TP

## Limitations :

- on doit corriger les tests à la main
- ne peut détecter un oracle invalide

## Quelques palliatifs

- diminuer l'effort de correction : idéal = scripts de tests passent par une API de quelques fonctions bien définies, comme ça on réécrit uniquement ces fonctions (notion de action-based testing)
- détecter les problèmes d'oracle : lier chaque test à la spécification (informelle) qu'il teste. Quand une spécification change, lister automatiquement tous les tests potentiellement affectés à partir de l'identifiant de la spéc, et les inspecter

- Contexte
- Maintenance des tests
- JUnit
- Sélection
- Minimisation / Prioritisation
- Discussion



Outil de gestion des *tests unitaires* pour les programmes Java, JUnit fait partie d'un cadre plus général pour le test unitaire des programmes, le modèle de conception (*pattern*) XUnit (CUnit, ObjcUnit, etc.).

JUnit offre :

- des primitives pour créer un test (*assertions*)
- des primitives pour gérer des suites de tests
- des facilités pour l'exécution des tests
- statistiques sur l'exécution des tests
- interface graphique pour la couverture des tests
- points d'extensions pour des situations spécifiques

dans une archive Java `junit.jar` dont le principal packaging est `junit.framework`.

Il existe un *plug-in* Eclipse pour JUnit.

<http://junit.org>

1. Pour chaque fichier `Foo.java` créer un fichier `FooTest.java` (dans le même repertoire) qui inclut (au moins) le paquetage `junit.framework.*`
2. Dans `FooTest.java`, pour chaque classe `Foo` de `Foo.java` écrire une classe `FooTest` qui hérite de `TestCase`
3. Dans `FooTest` définir les méthodes suivantes :
  - ▶ le constructeur qui initialise le nom de la suite de tests
  - ▶ `setUp` appelée avant chaque test
  - ▶ `tearDown` appelée après chaque test
  - ▶ une ou plusieurs méthodes dont le nom est prefixé par `test` et qui implementent les tests unitaires
  - ▶ `suite` qui appelle les tests unitaires
  - ▶ `main` qui appelle l'exécution de la suite

Dans les méthodes de test unitaire, les méthodes testées sont appelées et leur résultat est testé à l'aide d'**assertions** :

- `assertEquals(a,b)`  
teste si a est égal à b (a et b sont soit des valeurs primitives, soit des objets possédant une méthode equals)
- `assertTrue(a)` et `assertFalse(a)`  
testent si a est vrai resp. faux, avec a une expression booléenne
- `assertSame(a,b)` et `assertNotSame(a,b)`  
testent si a et b réfèrent au même objet ou non.
- `assertNull(a)` et `assertNotNull(a)`  
testent si a est null ou non, avec a un objet
- `fail(message)`  
si le test doit echouer (levée d'exception)

## Exemple : Conversion binaire/entier

```
// File Binaire.java
public class Binaire {
    private String tab;
    public Binaire() {tab = new String(); }
    public Binaire(String b, boolean be) {
        tab = new String(b); if (be) revert(); }

    private void revert() {
        byte[] btab = tab.getBytes();
        for (int i = 0; i < (btab.length >> 1); i++) {
            byte tmp = btab[i]; btab[i] = btab[btab.length - i];
            btab[btab.length - i] = tmp;        }
        tab = new String(btab); }

    public int getInt() {
        int nombre = 0;
        /* little endian */
        for (int i = tab.length()-1; i >= 0; i--) {
            nombre = (nombre << 1) + (tab.charAt(i) - '0'); }
        return nombre; }
}
```

## Exemple : Test Conversion binaire/entier (2)

```
// Fichier BinaireTest.java
import junit.framework.*;

public class BinaireTest extends TestCase {
    private Binaire bin; // variable pour les tests

    public BinaireTest(String name) {super(name); }

    protected void setUp() throws Exception {
        bin = new Binaire(); }

    protected void tearDown() throws Exception {
        bin = null; }

    public void testBinaire0() {
        assertEquals(bin.getInt(),0); }

    public void testBinaire1() {
        bin = new Binaire("01",false);
        assertEquals(bin.getInt(),2); }
```

Lancer en utilisant Eclipse :

- ajouter JUnit dans les libraries du projet
- exécuter BinaireTest.java comme test JUnit
- Remarque : dans ce cas, les méthodes main et suite ne sont pas nécessaires

- Ecrire les test en même temps que le code.
- Exécuter ses tests aussi souvent que possible, idéalement après chaque changement de code.
- Ecrire un test pour tout bogue signalé (même s'il est corrigé).
- Ne pas tester plusieurs méthodes dans un même test : JUnit s'arrete à la première erreur.
- Attention, les méthodes privées ne peuvent pas être testées !

- Contexte
- Maintenance des tests
- JUnit
- Sélection
- Minimisation / Prioritisation
- Discussion



Compromis entre tout rejouer (sûr mais trop cher) et ne pas rejouer assez.

- certains tests ne passent pas par les modifications : les ignorer

Problème additionnel : temps total pour le jeu limité

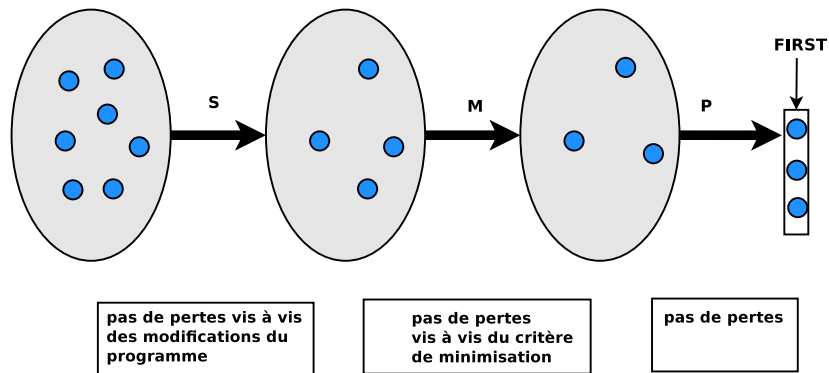
- on arrête après  $N$  tests
- avec cette limite, le jeu total est risqué
- faire tests pertinents d'abord

---

Trois phases distinctes dans la sélection :

- **Sélectionner** les tests pertinents (aucune perte)
- **Minimiser** les tests pertinents (perte possible)
- **Prioritiser** les tests restants (aucune perte)

## Problème SMP (2)



Principe : ne garder que des tests passant effectivement par des instructions modifiées

Pré-requis

- marquage des blocs du programme modifiés
- avoir récupéré et stocké les traces d'exécution des tests

Algorithme :

- entrées : blocks modifiés, ensemble des traces
- garder un test ssi sa trace contient au moins un bloc modifié

Principe : ne garder que des tests passant effectivement par des instructions modifiées

Pré-requis

- marquage des blocs du programme modifiés
- avoir récupéré et stocké les traces d'exécution des tests

Algorithme : [on ne perd rien !]

- entrées : blocks modifiés, ensemble des traces
- garder un test ssi sa trace contient au moins un bloc modifié

## Méthode des slicing dynamiques

- un test qui passe par une instruction modifiée ne pouvant changer le résultat ne sert à rien (sauf la première fois pour vérifier pas de crash)
- utiliser le slicing le long des chemins (DFG) : ne considérer que les instructions modifiées affectant la valeur finale

## Traces et Graphes de dépendances réduits

- compacter les traces et les graphes de dépendances
- unifier instructions identiques
- gain mémoire

Les aspects coûteux dépendent de la taille du programme

- trouver les modifications du programme (1 seule fois)
- calculer et sauvegarder les traces et impacts

Souvent : ok pour tests unitaires (ne pas considérer les modifications des fonctions appelées), mais difficile pour tests système

Une possibilité : perdre en précision

- traces : appels de fonctions plutôt que instructions
- instruction  $i$  de fonction  $F$  modifiée :  $F$  considérée modifiée
- dépendances CFG et DFG au niveau des fonctions (callgraph)

- Contexte
- Maintenance des tests
- JUnit
- Sélection
- Minimisation / Prioritisation
- Discussion

Se restreindre aux seuls tests pertinents peut encore être trop coûteux !!

idée : se donner un critère de qualité (souvent : couverture de contrôle I, D voir F (fonctions))

- calculer le score atteint par le jeu de test sélectionné :  $\text{score}(\text{TS})$
- trouver un jeu de test  $\text{TS}' \subseteq \text{TS}$  tel que  $\text{score}(\text{TS}) = \text{score}(\text{TS}')$

exemple simple : couverture des instructions et algorithme de minimisation glouton

Obtenir une minimisation importante :

- algorithme de minimisation très fin (attention : problème NP-complet)  
(pas de perte de qualité)
- critère de qualité assez faible (perte de qualité)



# Minimisation (2)

## Algorithme glouton

- Input : une suite de tests TS, chacun ayant un “score” (ensemble des instructions couvertes)
- Output :  $TS' \subseteq TS$  tq  $score(TS') = score(TS)$

1.  $X := TS$
2.  $TS' := \emptyset$
3. **Tant que**  $X$  non vide et  $score(TS') < score(TS)$  **faire**
4.     soit  $t \in X$  tq  $score(TS' + t)$  est maximal
5.      $X := X - t$
6.      $TS' := TS' + t$
7. **fin faire**
8. **return**  $TS'$

Attention : TS' couvre moins de comportement que TS en général

On peut accentuer la minimisation en choisissant TS' tel que

$$\text{score}(\text{TS}') \geq \text{score}(\text{TS}) - \varepsilon$$

idée : on ne minimise pas l'ensemble de tests, mais on se donne une priorité et les tests sont joués selon leur ordre de priorité

Mesure de priorité classique : se donner un critère de qualité, et ordre = gain résiduel de qualité

Intérêt :

- en cas de budget contraint, les tests les plus importants sont joués en premier
- en cas de budget suffisant, tous les tests sont rejoués (pas de perte)

Liens forts prioritisaiton - minimisation

- une méthode de minimisation : prioriser et ne garder que les  $K$  premiers
- une méthode de prioritisaiton : minimiser avec l'algo glouton puis ajouter les tests redondants

- Contexte
- Maintenance des tests
- JUnit
- Sélection
- Minimisation / Prioritisation
- Discussion

Pour les systèmes réactifs embarqués “boîtes noires”, les tests doivent s'enchaîner : le test  $t_n$  met l'état interne du système dans l'état requis pour le test  $t_{n+1}$

Dans ce cas, toutes les techniques de sélection / minimisation / priorisation posent problème

Idées ?

- spécifier les liens entre tests et les prendre en compte dans les algos ?

Il existe de nombreux outils automatiques pour outiller la maintenance et la sélection des tests de régression (pour des logiciels type PC).

Intéressant

- techniques assez simples donc faciles à utiliser et efficaces
- tests de régression coûteux : coûts  $\searrow$ , # bugs trouvés  $\nearrow$

cf. expérience à Microsoft

État de la technique = tests de régression doivent être automatisés

- maintenance : cohérence des DT vérifiée automatiquement
- sélection en (grande) partie automatisée
- rejou, verdict et rapport complètement automatique

## Maintenance + rejeu

- frameworks à la JUnit (libre, léger)
- autres outils commerciaux (plus de suivi, plus lourds)

## Sélection / Minimisation / prioritisation + rejeu

- outils commerciaux, surtout pour C
- ATAC/ $\chi$  Suds (Telecordia Techno., 1992)
- TestTube (AT&T Bell Labs, 1994)
- Echelon (Microsoft, 2002)

Si les tests sont générés à partir d'un modèle (model-based testing), alors :

- il est envisageable de détecter certaines erreurs d'oracle automatiquement
- idéal : ne pas stocker les tests, mais les régénérer ??