

R4.02 - Qualité de développement 3

CM3 – Analyse de Mutation

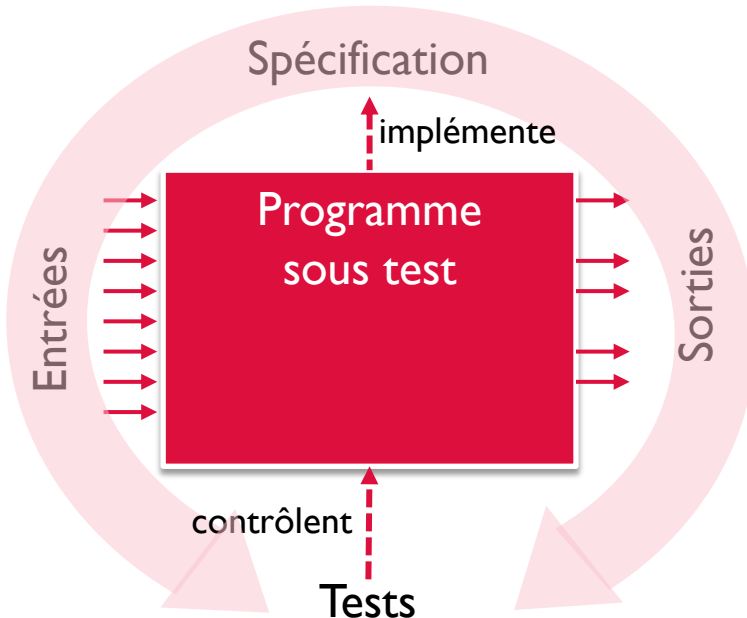
Jean-Marie Mottu
BUT info 2 – IUT Nantes

Rappel Couverture des tests :

Création de tests avec différentes approches

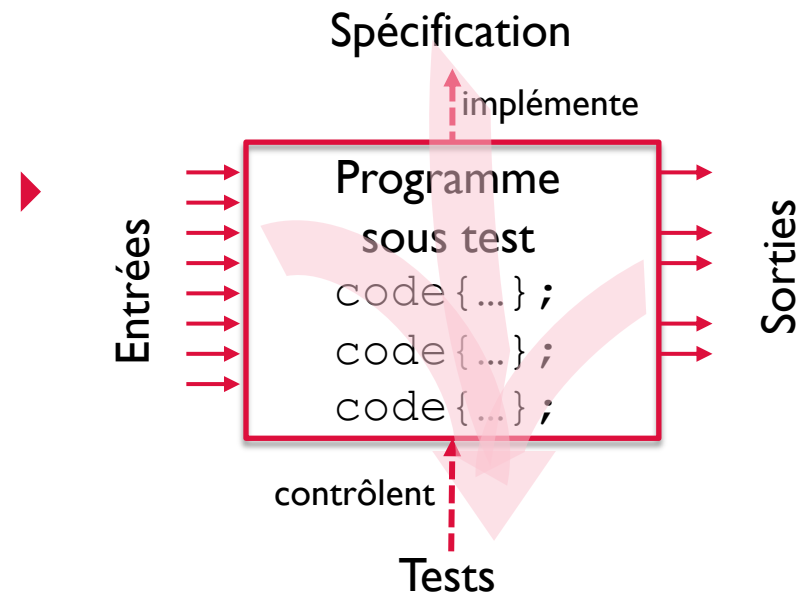
► Test fonctionnel (test boîte noire)

- Couverture de la spécification

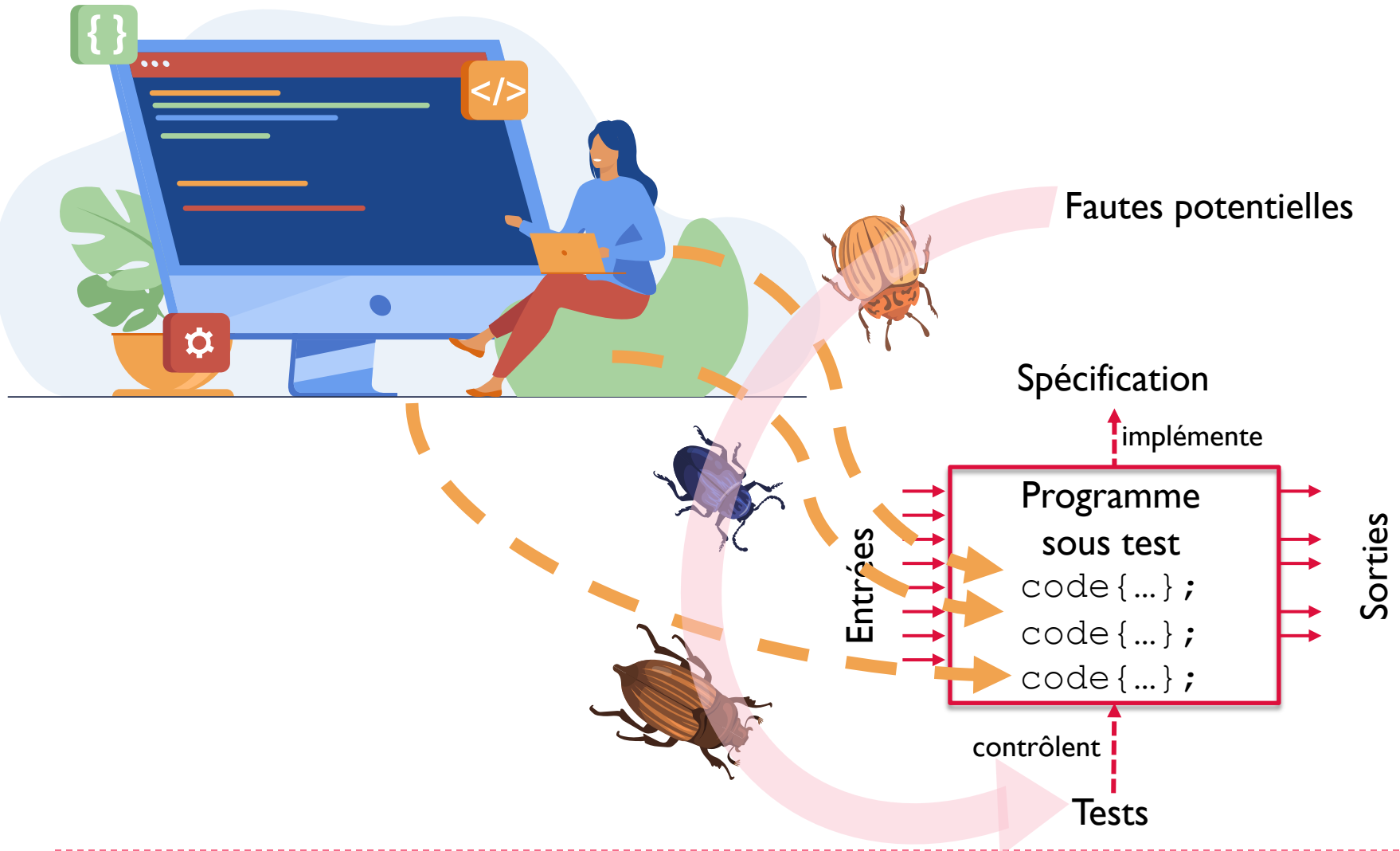


► Test structurel (test boîte blanche)

- Couverture du code



Couverture des tests : L'analyse de mutation pour couvrir les fautes potentielles



Intuition de l'analyse de mutation

- ▶ On augmente la confiance dans le programme en augmentant la qualité des tests
 - ▶ la qualité à détecter des fautes (grâce à des techniques basées sur différentes couvertures)
- ▶ L'analyse de mutation permet d'évaluer la qualité des tests vis-à-vis du logiciel
 - ▶ Couverture des fautes potentielles
 - ▶ On peut parler de « tester les tests »
- ▶ Si les tests peuvent détecter des fautes injectées intentionnellement, ils peuvent détecter des fautes réelles

Quelles fautes injecter ?

Hypothèses

- ▶ **Le programmeur est compétent**
 - ▶ Les programmeurs de niveau professionnel sont compétents et implémentent des programmes *presque* corrects.
 - ▶ Ces programmes sont seulement un peu différents de la version correcte.
- ▶ **L'effet de couplage.**
 - ▶ Une faute complexe commise par un programmeur compétent est la combinaison de fautes simples

Détecter les fautes simples

=>

Détecter les fautes les plus complexes qu'un programmeur compétent pourrait commettre

Analyse de mutation

- ▶ Fondée sur l'injection de fautes : des mutations
 - ▶ Les mutations sont appliquées au programme sous test
- ▶ Un mutant est le programme sous test ayant subi une et une seule mutation
 - ▶ Il y a autant de mutants que de mutations possibles
- ▶ Un test tue un mutant s'il détecte sa mutation
 - ▶ Un mutant reste vivant si aucun test n'a pu le tuer
- ▶ L'analyse de mutation évalue la qualité des tests comme leur capacité à détecter les fautes potentielles
 - ▶ Plus les tests tuent de mutants et plus ils sont de qualité
- ▶ Le score de mutation est le pourcentage de mutants tués

Opérateurs de mutation

- ▶ Les fautes potentielles sont modélisées par des *opérateurs de mutation*
 - ▶ Principalement des modifications syntaxiques du code
 - ▶ Les opérateurs sont appliqués autant que possible pour créer autant de mutant que d'application
 - ▶ Une seule mutation par mutant

Programme sous test :

```
int sum(int x, int y) {  
    int res = 0;  
    if (x == 0) {  
        res = y;  
    } else {  
        res = x + y;  
    }  
    return res;  
}
```

op : + → -

Mutant I :

```
int sum(int x, int y) {  
    int res = 0;  
    if (x == 0) {  
        res = y;  
    } else {  
        res = x - y;  
    }  
    return res;  
}
```



Opérateurs de mutation : procédural

- ▶ Remplacement d'un opérateur arithmétique
 - ▶ '+' par '-', par '*' etc.
- ▶ Remplacement d'un opérateur logique
 - ▶ and, or, nand, nor, xor remplacés,
 - ▶ les expressions sont remplacées par TRUE et/ou FALSE
- ▶ Remplacement des opérateurs relationnels
 - ▶ <, >, <=, >=, =, != remplacés
- ▶ Suppression d'instruction
- ▶ Perturbation de variable et de constante
 - ▶ +1 sur une variable
 - ▶ booléens remplacés par leur complément.

Opérateurs de mutation OO

- ▶ **Exception Handling Fault**
 - ▶ force une exception
- ▶ **Visibilité**
 - ▶ passe un élément privé en public et vive-versa
- ▶ **Faute de référence (Alias/Copy)**
 - ▶ passer un objet à null après sa création.
 - ▶ supprimer une instruction de clone ou copie.
 - ▶ ajouter un clone.
- ▶ **Inversion de paramètres dans la déclaration d'une méthode**
- ▶ **Polymorphisme**
 - ▶ affecter une variable avec un objet de type « frère »
 - ▶ appeler une méthode sur un objet « frère »
 - ▶ supprimer l'appel à super
 - ▶ suppression de la surcharge d'une méthode

Autres opérateurs

- ▶ Une multitude depuis les années 70
- ▶ Dédiés à un langage
 - ▶ En Java
 - ▶ erreurs sur *static*
 - ▶ mettre des fautes dans les librairies
- ▶ Sélectionnés par un outils d'analyse de mutation
 - ▶ <https://pitest.org/quickstart/mutators/>
- ▶ Moins syntaxiques, plus basés sur un paradigme
 - ▶ OO, AOP, IDM

Génération de mutants

- ▶ Première partie d'un outil d'analyse de mutation
 - ▶ A partir d'un programme sous test génère un ensemble de mutants
 - ▶ Par analyse et modification du code, du bytecode, d'un modèle du code
- ▶ Les mutants générés sont nombreux
- ▶ Exemple d'outils
 - ▶ Pour java, kotlin : <http://pitest.org/>
 - ▶ Pour JS, C#, Scala : <https://stryker-mutator.io/>

Modèle RIP : Reach/Infect/Propagate

3 conditions pour tuer un mutant

▶ Reach

- ▶ Le cas de test doit atteindre la mutation

▶ Infect

- ▶ L'exécution du cas de test doit être impactée par la mutation

▶ Propagate

- ▶ L'erreur provoquée doit se propager jusqu'à être observée par le cas de test (et son oracle)

Modèle RIP : Reach/Infect/Propagate

3 conditions pour tuer un mutant

Programme sous test :

```
int sum(int x, int y) {  
    int res = 0;  
    if (x == 0) {  
        res = y;  
    } else {  
        res = x + y;  
    }  
    return res;  
}
```

Mutant I :

```
int sum(int x, int y) {  
    int res = 0;  
    if (x == 0) {  
        res = y;  
    } else {  
        res = x - y;  
    }  
    return res;  
}
```

op : + → -



Considérer les cas de test :

Reach ?

Infect ?

Propagate ?

=> Killed

Cas de Test 1 : ((x=0, y= 5), expected res = 5)
=> don't Reach

Cas de Test 2 : ((x=5, y= 0), expected res = 5)
=> Reach, don't Infect

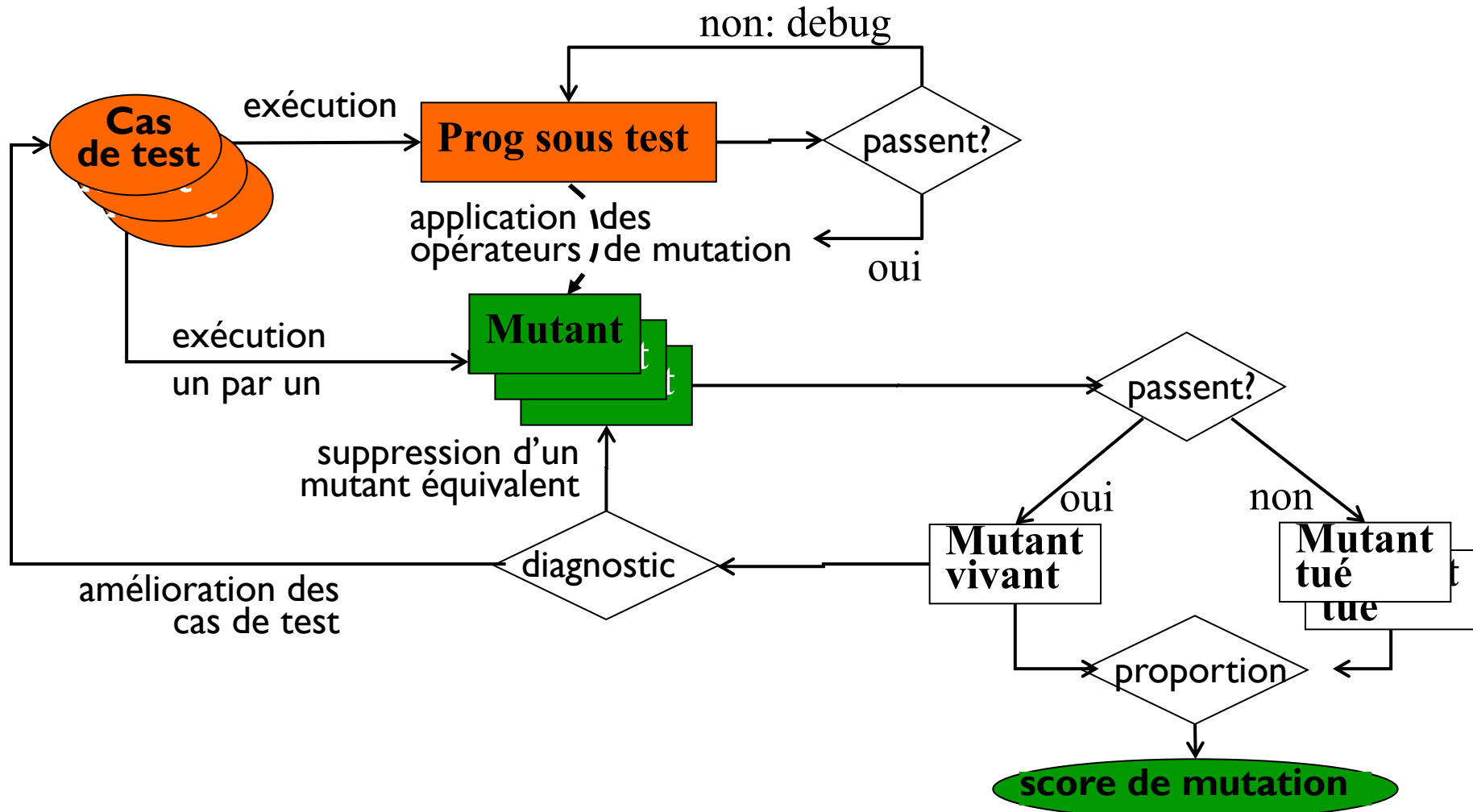
Cas de Test 3 : ((x=5, y= 1), expected res > 0)
=> Reach, Infect, don't Propagate

Cas de Test 4 : ((x=5, y= 1), expected res = 6)
=> Reach, Infect, Propagate



Appliqué à l'ensemble des mutants et des cas de test

- ▶ En préalable, les cas de test passent sur le programme sous test



Mutants équivalents


Programme sous test :

```
int sum(int x, int y) {  
    int res = 0;  
    if (x == 0) {  
        res = y;  
    } else {  
        res = x + y;  
    }  
    return res;  
}
```

op : +1 sur valeur

Mutant 2 :

```
int sum(int x, int y) {  
    int res = 0+1;  
    if (x == 0) {  
        res = y;  
    } else {  
        res = x + y;  
    }  
    return res;  
}
```



Stratégies selon les phases et les objectifs

► Performance à priori

- L'analyse de mutation étant très longue
 - On ne considère plus un mutant dès qu'il est tué
 - On ordonne les tentatives en commençant par les tests ayant le plus de chance de tuer beaucoup de mutants
 - En particulier ceux ayant la plus grande couverture
 - Ce qu'on mesure sur le programme sous test

► Performance à posteriori, soit

- Effectuer l'analyse complète
 - Puis réduire la matrice de mutation
- Une retro-analyse partielle
 - Ordonner l'analyse « à priori » puis relancer les tests ayant l'exclusivité du meurtre d'un mutant contre tous les mutants, puis réduire la matrice de mutation partielle

► Analyse

- Pour pouvoir tuer des mutants vivants il est utile de comparer les traces des tests les tuant et ne les tuant pas et ainsi créer de nouveaux tests

Matrice de mutation

► Complète

	CT1	CT2	CT3	Killed
Mutant1	K	A	A	K
Mutant2	K	K	A	K
Mutant3	A	K	K	K
Mutant4	A	A	A	A

► Partielle, optimisée à priori

	CT1	CT2	CT3	Killed
Mutant1	K			K
Mutant2	K			K
Mutant3	A	K		K
Mutant4	A	A	A	A

► Réduite à postériori

	CT1		CT3	Killed
Mutant1	K		A	K
Mutant2	K		A	K
Mutant3	A		K	K
Mutant4	A		A	A

Mutants vivants et score de mutation

- ▶ Si un mutant reste vivant?
 - ▶ Qualité des cas de test insuffisantes
 - ▶ ajouter des cas de test avec de nouvelles données (RIP)
 - ▶ Améliorer les oracles (Propagate)
 - ▶ Mutant équivalent
 - ▶ supprimer le mutant
- ▶ Score de mutation $MS \{CT_i\} = \#k / \#m$
 - ▶ $\#k$ = nombre de mutants tués
 - ▶ $\#m$ = nombre de mutants non équivalents
- ▶ $MS \{CT_i\} = 100\%$ n'implique pas l'absence de faute
- ▶ Le MS de départ d'un ensemble $\{CT_i\}$ conçu méthodiquement est facilement élevé ($>70\%$), ce sont les derniers pourcentages qui font la différence