Tests

Différents niveaux de tests

- Tests unitaires
  - Junit, ctest, …
  - Test Driven Development
  - Intégration continue
- Tests d'intégration
- Tests d'acceptation (UAT, Recette)

Tests unitaires

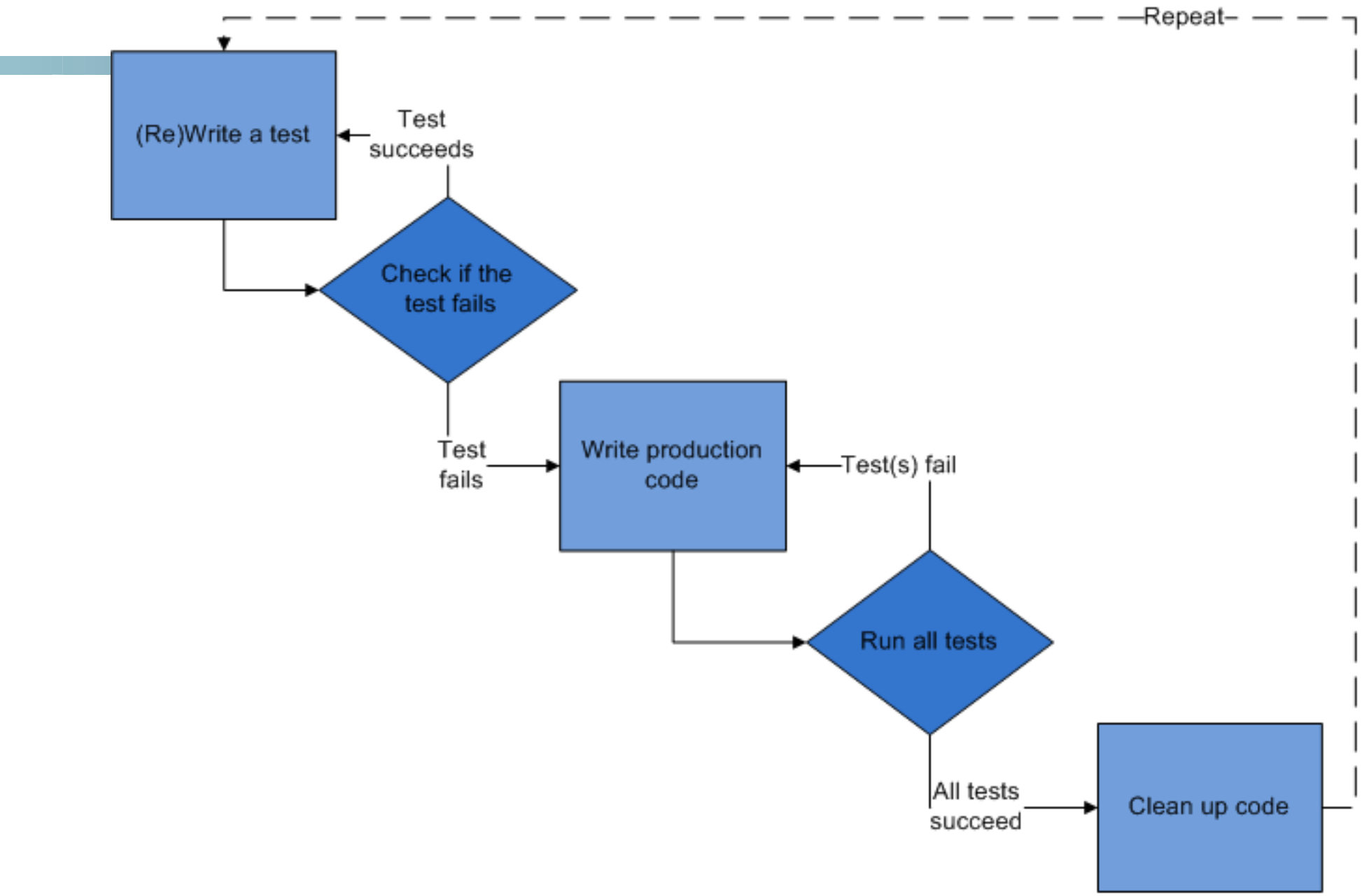
- Ne devrait pas exister !
  - Cela fait partie du développement

- Test = vérifier en exécution
- Vérifier = contrôle que le résultat est conforme (aux spécifications)
- Unitaire : test au niveau le plus bas
  - Fonction, Classe, Module, …

- Données en entrées + exécution + vérification que le résultat est conforme aux attentes + « assert » ou équivalent

Test Driven Developpement

- Le développement piloté par les tests.
- Génère plus de code mais ce code est produit plus vite (code des tests en +)
- Moins d'utilisation du Débuggeur
- Attention a la couverture du code !!


Repeat
(Re)Write a test
Test succeeds
Check if the test fails
Test fails
Write production code
Test(s) fail
Run all tests
All tests succeed
Clean up code

Limites du TDD

- TDD est difficile a utiliser quand des tests fonctionnels complets sont nécessaires pour déterminer le succès ou l'échec. GUI,BD,Net.
  - TDD encourage à minimiser le code des modules de haut-niveau et à maximiser ce qui peut être placé et tester dans des librairies
    - Et usage de Faux, Bouchons, Mockup, pour représenter le monde extérieur (ie. code qui retourne toujours le même résulat, ou des résultats aléatoires).
- LE soutien de la hiérarchie est fondamental (sinon l'ecriture de code de tests est considérée une perte de temps).
- Les tests doivent aussi être maintenus => coût
  - si les tests sont mal codés (chaines codées en dur, …)
  - des tests qui plantent souvent sont dur à maintenir.
    - un test qui plante souvent peut finir par être ignoré
  - Pour une maintenance raisonnable => activité de nettoyage régulier

Limites (2)

- Le niveau de couverture des tests obtenu pendant des cycles TDD est dur à conserver lors de la maintenance
  - => les tests originaux deviennent de plus en plus précieux.
- Le niveau de couverture des tests obtenu pendant des cycles TDD est dur à conserver lors de la maintenance
  - => les tests originaux deviennent de plus en plus précieux.
- Des trous dans la couverture de tests apparaissent pour différentes raisons :
  - Un développeur pas convaincu !
  - tests détruits ou commentés
  - => baisse la confiance mal placée (risque bugs non détectés)
- Les tests unitaires sont écrits par le développeur du code
  - Une erreur d'interprétation de la fonctionnalité peut créer du code et des tests compatibles mais faux tout les deux !

Limites (3)

- Les tests unitaires même très nombreux sont insuffisants il faut avoir des tests d'intégration et de conformité

Intégration continue

- Terme un peu ambigü
  - Consiste en premier lieu à intégrer les tests unitaires de manière très forte au cycle de développement
  - À ne pas confondre avec les « tests d'intégration »

Intégration continue

- Principes :
  - maintenir un dépôt unique de code source versionné
  - automatiser les compilations
  - rendre les compilations auto-testantes
  - tout le monde committe tous les jours
  - tout commit doit compiler sur une machine d'intégration
  - maintenir une compilation courte (< 10 mn)
  - tester dans un environnement de production cloné
  - rendre disponible facilement le dernier exécutable
  - tout le monde doit voir ce qui se passe
  - automatiser le déploiement

Pour aller plus loin...

- http://www.martinfowler.com/articles/continuousIntegrat
- http://hudson-ci.org/

Tests d'acceptation (Recette)

- Les test unitaires ont trouvé 90% des problèmes
- Les tests d'intégration 9%
- Les tests d'acceptation en trouvent 0,9%

→ Pour que seuls 0,1% des bugs se retrouvent en production !

- Responsabilité du « métier », du « client » (MOA ou AMOA)

Refactoring

- Améliorer la conception de code existant
  - « Lifting permanent »
  - Doit s'appuyer sur un usage intensif et efficace des tests
    - Sinon, les restructurations crée trop de régressions
  - Règles générales pour repérer et appliquer du refactoring
    - « Refactoring, Improving the design of existing code », Martin Fowler
    - Inclusion d'outils dans Eclipse ! (#1 : faciliter le renommage)
  - Un moment privilégié pour introduire ou enlever des Design Patterns
    - « Refactoring to Patterns », J. Kerievsky

Refactoring

- Vision traditionnelle du développement de logiciels
  - D'abord un bon design, ensuite la programmation
  - Mais avec le temps le code sera modifié
    - l'intégrité du système par rapport au design va s'estomper
    - le code passe d'une approche « génie logiciel » → hacking
- Refactorisation : approche inverse
  - Modifier petit à petit le code pour obtenir un design sain
  - Le design se fait continuellement pendant le développement
  - En construisant le système,
    on découvre comment en améliorer le design

Catalogue de techniques de refactorisation

- Composition de méthodes
  - « Extract Method »
  - « Split with Temporary Variable »
  - « Replace Temp with Query »
- Déplacement de « propriétés » entre les objets
  - « Move Method »
- Organisation des données
  - « Self Encapsulate Field »
  - « Replace Type Code with State/Strategy »
- Simplification des expressions conditionnelles
  - « Replace Conditional with Polymorphism »
- Simplification des invocations de méthodes
- Gestion de la généralisation
  - « Form Template Method »

...

Exemple: Extract Method

- Un fragment de code peut être regroupé
  - Le mettre dans une méthode BIEN nommée

```
void printOwing() {
        printBanner();

        //print details
        System.out.println ("name:        " + _name);
        System.out.println ("amount      " + getOutstanding());
}
```

```
void printOwing() {
        printBanner();
        printDetails(getOutstanding());
}
void printDetails (double outstanding) {
        System.out.println ("name:        " + _name);
        System.out.println ("amount      " + outstanding);
}
```

Exemple: Split Temporary Variable

- Une même variable temporaire est utilisée pour plusieurs buts
  - Avoir plusieurs variables temporaires BIEN nommées

```
double temp = 2 * (_height + _width);
System.out.println (temp);
temp = _height * _width;
System.out.println (temp);
```

```
final double perimeter = 2 * (_height + _width);
System.out.println (perimeter);
final double area = _height * _width;
System.out.println (area);
```

Références

- Martin Fowler, et al., *Refactoring, Improving the design of existing code*, Addison-Wiley, 2000.
  - Chapitre 15 : a longer example
    - http://www.refactoring.com/rejectedExample.pdf
  - Catalogue de refactorisations
    - http://www.refactoring.com/catalog/index.html

- Refactoring to Patterns, J. Kerievsky
  - ajout/suppression de DP lors du refactoring

- Site web dédié à la refactorisation
  - http://www.refactoring.com/