Tests
Slides Complémentaires

Julien Romero

Tests Unitaires

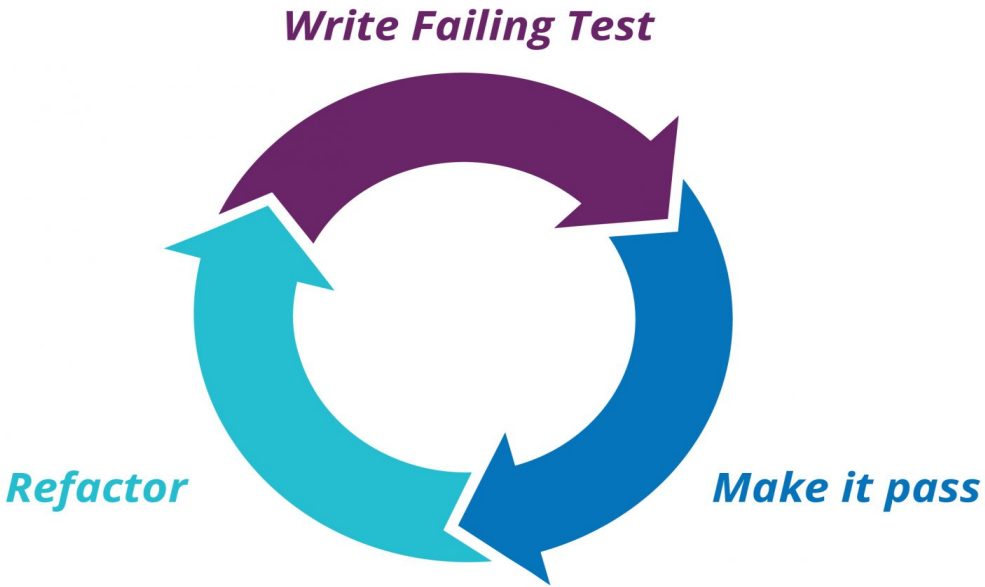
F.I.R.S.T.

- **Fast :** les tests doivent être rapide, sinon ils ne seront plus utilisés
- **Independent :** les tests ne doivent pas dépendre les uns des autres pour éviter toute cascade d'erreurs
- **Repeatable :** les tests doivent pouvoir d'effectuer dans tous les environnements
- **Self-Validating :** le résultat d'un test est binaire : soit il passe, soit il échoue. Aucune intervention humaine ne doit avoir lieu.
- **Timely :** un test est écrit avant le code testé. L'inverse rend l'écriture des tests compliquée.

Test Driven Development – Trois lois

- Il est interdit d'écrire du code de production avant d'avoir écrit un test qui **échoue.**
- Il est interdit d'écrire plus d'un test unitaire qui échoue, et ne pas compiler est échouer
- Il est interdit d'écrire plus de code de production qu'il est nécessaire pour faire passer le test en train d'échouer : le code doit être **minimal**

Test Driven Development – Cycle



Garder les tests propres

- Les tests sont aussi importants que le code de production
- Les tests gardent le code flexible, modifiable et réutilisable
- Les tests doivent être encore plus faciles à lire que le code de production
- Un seul *assert* par test

Langage de test

- Les tests se complexifient avec l'application:
  - Création de tests plus haut niveau
  - Mise en place d'un langage de tests spécifique

Exemple:

Flask (Serveur Web en Python) propose de simuler des connexions au serveurs avec des fonctions spécialisées: get(url) permet d'obtenir la page généré pour l'url alors qu'en pratique il faut un navigateur ou faire un accès réseau.

Code de production vs tests

Les tests doivent être:

- Simple
- Succinct
- Expressif
- Lisible

Contrairement au code de production, l'efficacité n'est pas important

Par exemple, faire de la concaténation de String est ok

Demo

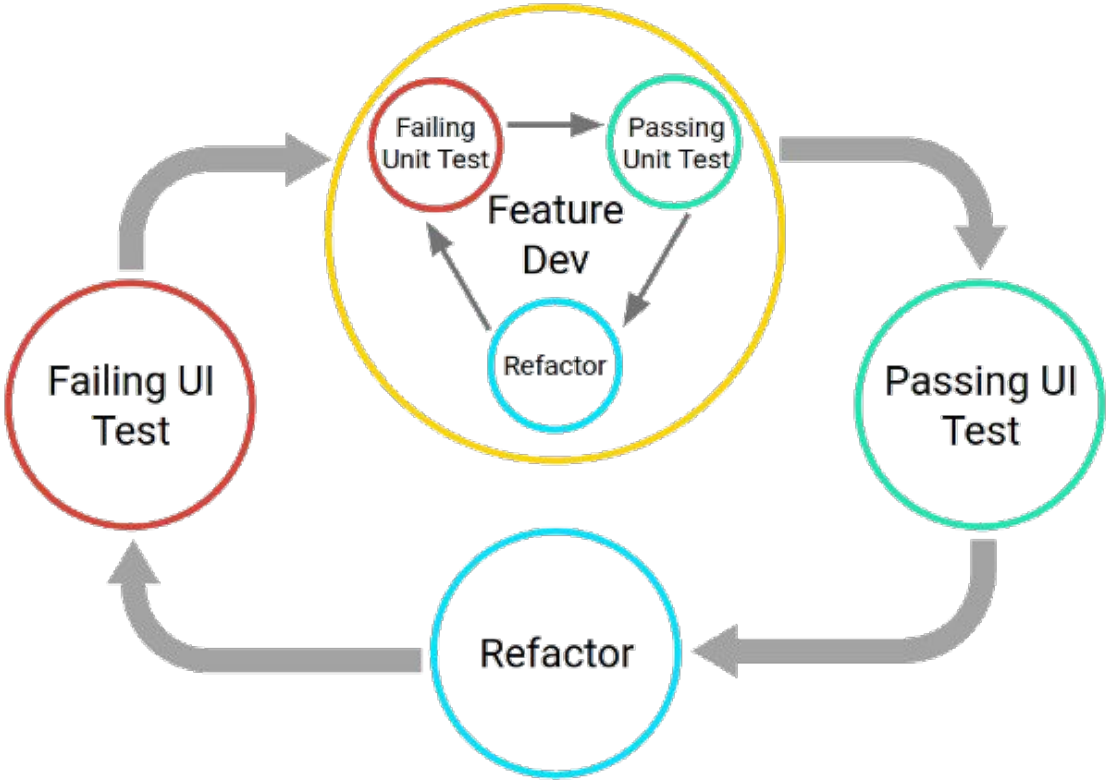
Tester les interfaces graphiques

Tester une interface graphique

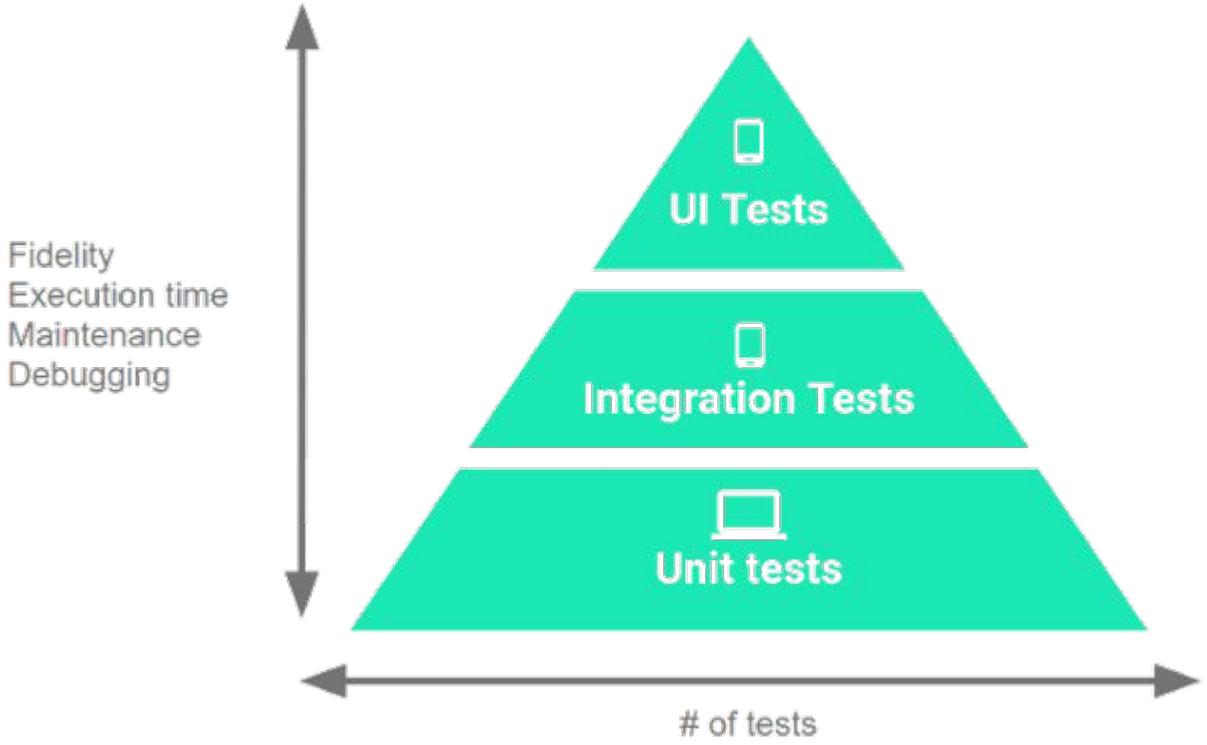
Les tests sur les interfaces graphique:

- Doivent simuler un utilisateur (clic, texte, …)
- Sont plus longs à exécuter
- Demandent plusieurs composants intégrés
- Nécessite de simuler l'interface
- Utilisent d'autres bibliothèques

User Interface in TDD



Proportions of tests in TDD



Android

- https://developer.android.com/training/testing/fundamentals
- Vrai téléphone vs Emulation vs Simulation
- Robolectric (http://robolectric.org/) ou Expresso (https://developer.android.com/training/testing/espresso)

Site Web

- Nécessite un navigateur pour rendre la page et exécuter du Javascript
- Selenium (https://selenium.dev/) permet de contrôler un navigateur depuis du code

Tester le code concurrent

Quelques conseils

- Ne pas ignorer les erreurs comme étant “rares”
- Faire fonctionner le code non parallèle en premier
- Ne jamais débugger du code parallèle et du code séquentiel en même temps
- Faire s’adapter le code parallèle à pleins de situations
- Rendre le code parallèle facilement configuration (nb threads)
- Tester plus de threads que de processeurs
- Tester sur plusieurs OS
- Introduction automatique ou manuellement de la variabilité dans le code. c.f. jiggling

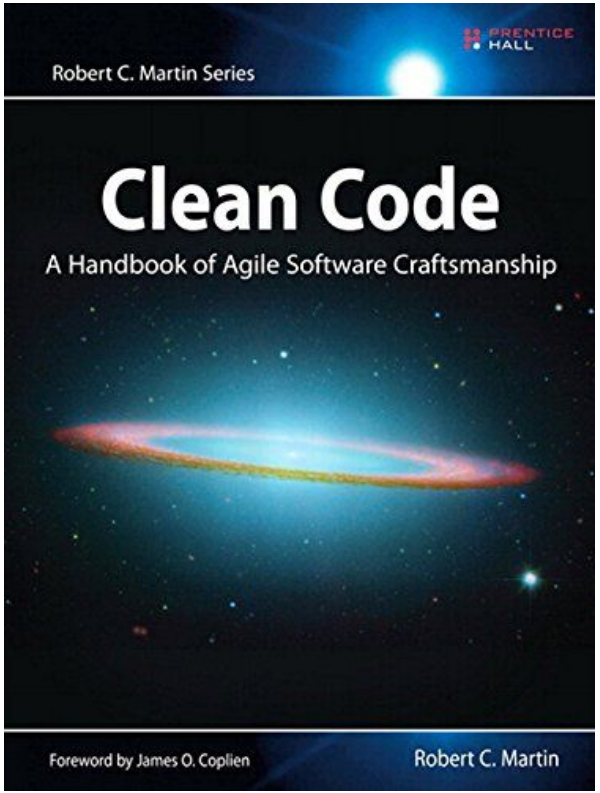
Derniers conseils

Quelques conseils

- **Faire suffisamment de tests**: Toutes les conditions doivent être testées.
- **Utiliser un outil de couverture de code**
- **Ne pas éviter les tests triviaux**
- **Un test ignoré est une ambiguïté**
- **Tester les conditions limites**
- **Faire beaucoup de tests près des bugs**: Quand un bug est découvert, il faut écrire des tests autour de lui.
- **Faire attention aux motifs dans les erreurs**: Ils sont souvent révélateur d'une insuffisance dans les tests, comme par exemple des entrées trop petites.
- **Faire attention aux motifs dans la couverture du code**
- **Les tests doivent être rapides**

Intégration Continue

Ressources



Uncle Bob