Tests logiciels

...

les tests unitaires

Plan

1. Pourquoi tester ?
2. Pyramide des tests
3. Tests unitaires
4. Stratégie de tests pour une structure de code
5. Tests d’intégration
6. Tests fonctionnels
7. Autres (contrat, ui, e2e, charge, ...)

Session 1

- Importance des tests
- Tests U
- Mise en pratique TDD : kata en groupe
- En TP : kata survivre dans du code ‘legacy’

Importance des tests / sondage

- Qui a réalisé des tests manuels ? automatisés ?
- Pouvez-vous me donner votre couverture de tests ?
- Pourquoi en avez-vous codé ?
- Un retour d’expérience

Importance des tests / qualité

Il y a une notion de **qualité**

- en s'assurant que le développement respecte les exigences fonctionnelles
  - vérification vs validation

Cela n'implique pas nécessairement :

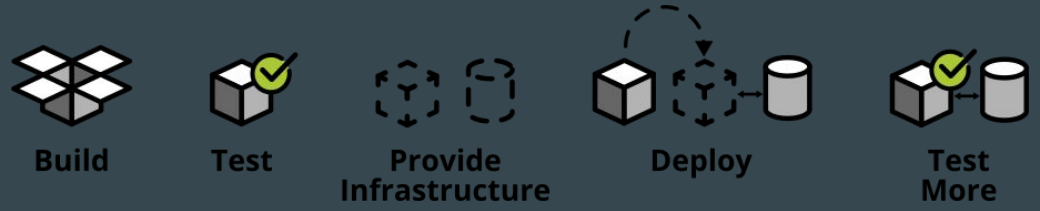
- des tests automatisés
- du code de qualité (lisibilité, maintenabilité, normes, ...)

Importance des tests / compétence professionnelle

- à ne pas sous-estimer
  - sur les offres d'emploi
  - en entretien pro
- connaissance utile à tous les niveaux
  - développeur : software craftmanship
  - chef de projet : mieux comprendre la qualité

Importance des tests / automatisation

- le développement logiciel participe à l'activité économique
- triangle coût / qualité / délai
- objectif : maj logicielles fréquentes sans réduire la qualité : **Continuous Delivery**

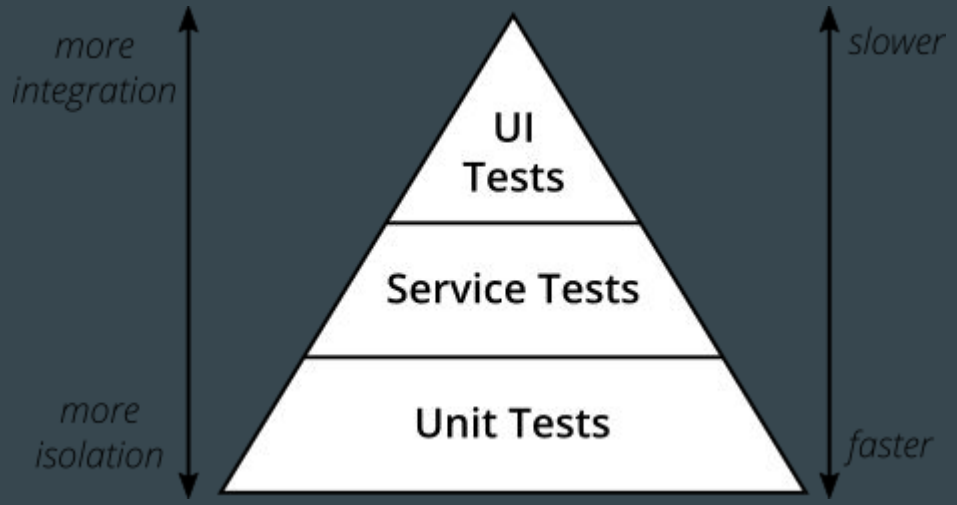


Automatiser les tests permet

- économiser du temps
- construire un filet de sécurité (safety net)
- introduire un feedback positif et quitter 'stress -> moins de tests -> bugs -> stress'

Pyramide des tests

- terminologie floue
- tests de différentes granularité
- quantité en fonction du type
- coût
- anti-pattern :
  - pyramide inversée
  - sablier
  - éviter la redondance

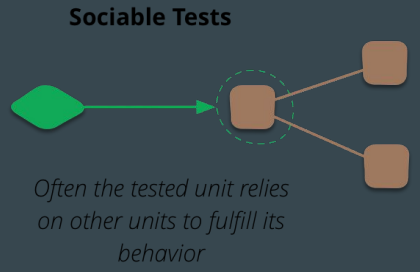


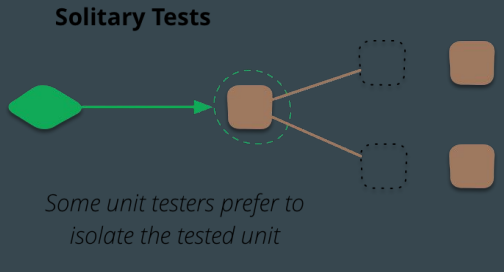
Tests unitaires / définition

- objectifs
  - garantir le bon fonctionnement d'une unité du logiciel (system under test)
  - être lisibles et maintenables (dry / damp)
- caractéristiques
  - périmètre réduit
  - écrits par le développeur
  - rapides
  - nécessitent uniquement le code source = sont isolés (pas de container, pas de BD, ...)
- taille d'une unité
  - la classe, voir un groupe de classes en POO
  - une fonction en programmation fonctionnel

Tests unitaires / différentes écoles

- Teste un élément à la fois mais nécessite souvent des collaborateurs
- école “sociable”
  - utilisation de collaborateurs “réels”
  - généralement le code métier
- école “solitaire”
  - utilisation de test doubles
  - services de coordinations





Les deux sont utiles, et répondent à des besoins différents

Tests unitaires / Mock et Stub

- 2 types de test doubles
- utilisation fréquente en TU
- Stub = bouchonner des données (orienté état)
- Mock = préparer et vérifier les interactions (orienté comportement)

Tests unitaires / quoi ?

- tester les interfaces publiques
  - code smell : tester méthodes privées
- tester les parties non triviales ...
- ... tout en évitant d'être trop proche de l'implémentation
  - tests deviennent fragiles

Tests unitaires / comment ?

- outils Java : JUnit / Mockito / Hamcrest
- structure d'un test
  - une classe de test par classe de production
  - given / when / then
    - setup
    - appel à la méthode testée
    - assertions

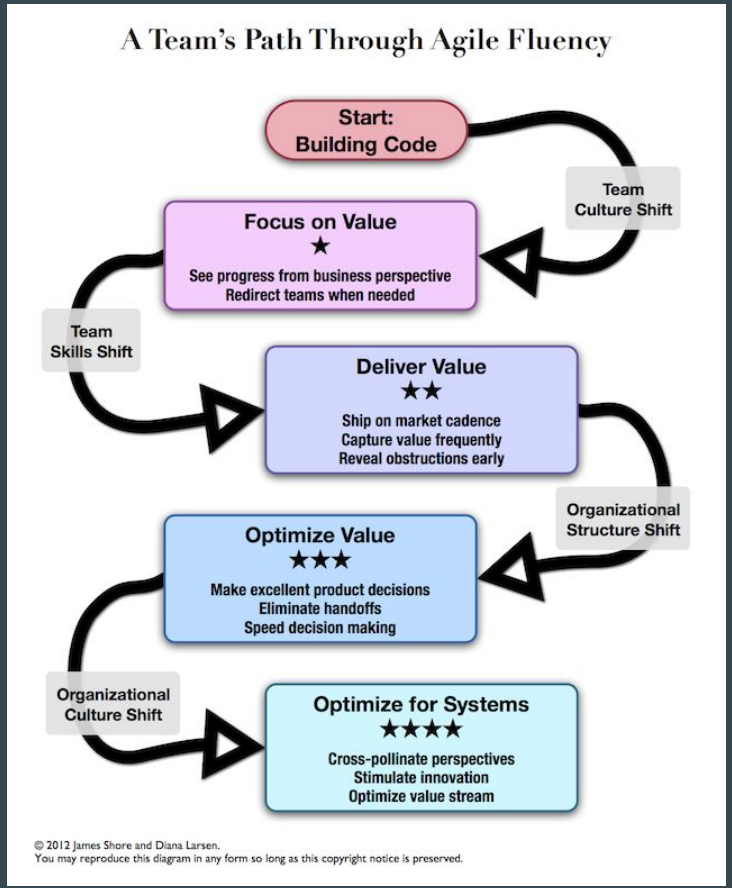
```
public class ExampleControllerTest {

    private ExampleController subject;

    @Mock
    private PersonRepository personRepo;

    @Before
    public void setUp() throws Exception {
        initMocks(this);
        subject = new ExampleController(personRepo);
    }

    @Test
    public void shouldReturnFullNameOfAPerson() throws Exception {
        Person peter = new Person("Peter", "Pan");
        given(personRepo.findByLastName("Pan"))
            .willReturn(Optional.of(peter));

        String greeting = subject.hello("Pan");

        assertThat(greeting, is("Hello Peter Pan!"));
    }

    @Test
    public void shouldTellIfPersonIsUnknown() throws Exception {
        given(personRepo.findByLastName(anyString()))
            .willReturn(Optional.empty());

        String greeting = subject.hello("Pan");

        assertThat(greeting, is("Who is this 'Pan' you're talking about?"));
    }
}
```

Tests unitaires / TDD

- Test Driven Development
- améliorer la qualité + outil de conception intéressant
- Principes
  - ‘clean code that works’
  - écrire les tests avant le code de production
  - Green / Red / Refactor mantra

Tests unitaires / TDD & XP & Agile

- Pair programming
  - conversations autour du code + switch si fatigue
- Intégration continue
- Livraison continue
- Simplicité (design)
- Courage
- Refactoring
- Permet d'obtenir une équipe agile à 2 étoiles



Synthèse et temps d’échange

En synthèse

- offrir un logiciel de qualité
- créer un filet de sécurité
- le code de test est aussi important que le code de production
- outil d’amélioration de ses techniques de développement

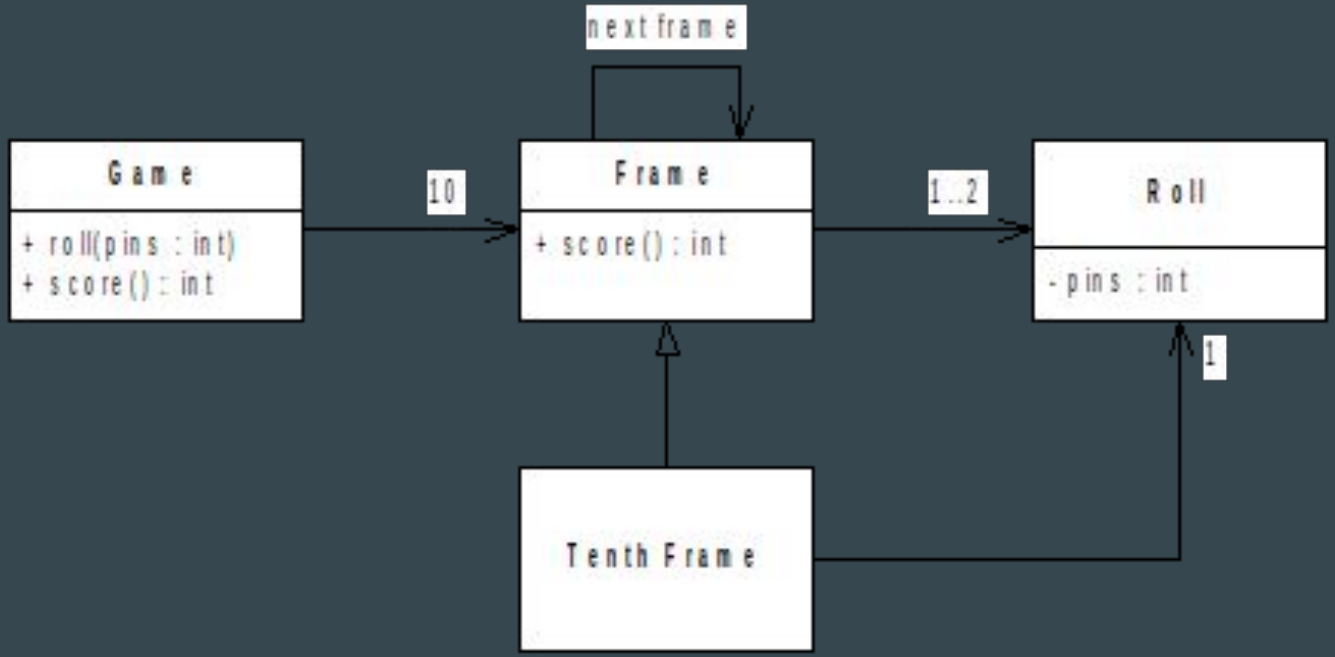
Kata Bowling / contexte

- en mob programming : driver / navigators
- en TDD
- en respectant quelques principes
  - Test List
  - Test First
  - Assert First et/ou Given / When / Then

Kata Bowling / exigences

- 10 “frames” composées de 2 lancers avec pour objectif de faire tomber 10 quilles
- le score d’une frame = nombre de quilles + bonus en cas de spare / strike
- bonus spare : nb de quilles du lancer suivant
  - ex: frame 3 = 14 + (10 + 5) = 29
- bonus strike : nb de quilles des 2 lancers suivants
  - ex: frame 5 = 49 + (10 + 0 + 1) = 60
- on peut jouer 3 boules lors de la dernière frame

Kata Bowling / modélisation



Kata Bowling / liste tests

1. test jeu “gouttières”
2. test 1 quille à chaque lancer
3. test spare
4. test strike
5. test jeu parfait

TP1 / Objectifs

- écriture de tests
- exécuter les tests régulièrement
- amélioration du code incrémentale