Bonjour tout le Monde

Annotations

→ Décorateur

classe → membres → décorer → ajouter de méta Données → Technique

```
@Important
@Metadata ( comment = "ma class")
class MyClass extend Testable
{
```

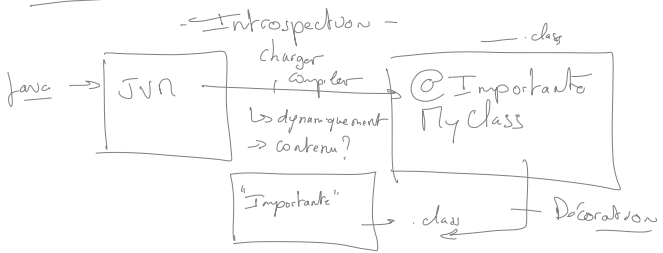


Figure 4.2 Running the `JUnit4SUTTestSuite` in IntelliJ using both the JUnit 4 dependency and the JUnit Vintage dependency

Listing 4.6 `JUnit5SUTTest` class

```
class JUnit5UTTest {
    private static ResourceForAllTests resourceForAllTests;
    private SUT systemUnderTest;

    @BeforeAll                                                    ①'
    static void setUpClass() {
        resourceForAllTests =
                new ResourceForAllTests("Our resource for all
tests");
    }

    @AfterAll                                                     ②'
    static void tearDownClass() {
        resourceForAllTests.close();
    }

    @BeforeEach                                                   ③'
    void setUp() {
        systemUnderTest = new SUT("Our system under test");
    }

    @AfterEach                                                    ④'
    void tearDown() {
        systemUnderTest.close();
    }

    @Test                                                         ⑤'
    void testRegularWork() {
        boolean canReceiveRegularWork =
                systemUnderTest.canReceiveRegularWork();

        assertTrue(canReceiveRegularWork);
    }

    @Test
    void testAdditionalWork() {
```

Pattern AOP

→ Prog. Modularisée - Métaphore = couture

→ points de coupe
" de jonction

→ Plugins

interface — impl
impl

liste des impl

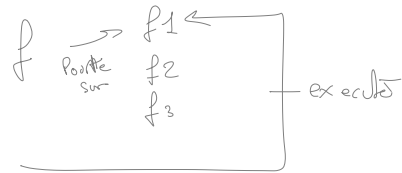
appel de méthode "vide"

Composant 1
before
after

Composant 2
before
after

Programmation fonctionnelle

- "Pointeur sur fonction"
  - → var. qui détient une adresse de fonction
    - → branchement dynamique (au lieu du statique à la compilation)



.net

→ Délégués

Référence vers des fonctions

f = f1; f += f2; f();

.net Java

→ fonctions anonymes

→ Lambdas :

= ( ) → _____

expression renvoyée au retour

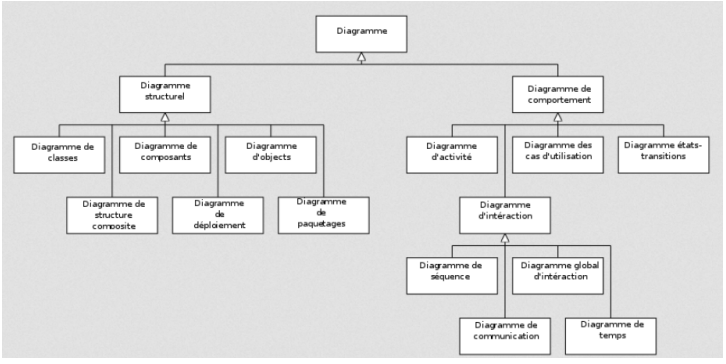
= ( ) → { ~ ; ~ ; } ← fonction Lambda

Authentification

Cas :
- Principaux → mot de passe conforme
- alternatifs → mot de passe vide
- Erreurs → mdp non conforme

Tester

UML



Temps passé = Temps d'une fonction × nb d'appels

1er Critère

2nd Critère.

Couverture de Code

Couvrir toutes les specs

↳ Couverture = ex 95% du code

Déduire : - Soit tous les tests ne sont pas écrits

- A l'inverse, le code est obsolète
  ↳ il est a retirer

— Exhaustif pour autant ? NON → c'est une aide

Versionning Sémantique

MAJEUR. Mineur

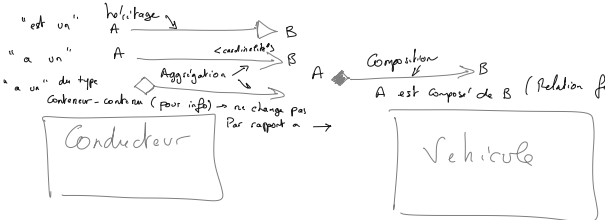
↗ Mineur = ajout de fct, optim, correction, garantie Compat ascendante

↘ MAJEUR = modifs qui cassent la Compat ascendante

- Accessibility testing
- Acceptance testing
- Black box testing : tester un système inconnu.
- End to end testing : tester un workflow fonctionel
- Functional testing : s'assurer qu'il fait exactement ce qui était prévu.
- Interactive testing : eq tests manuels
- Integration testing : test en environnement équivalent à la prod dans son ensemble.
  - Interface tests
- Load testing
- Non functional testing : catégorie de tests d'interface (conformité), accessibilité, performance, ...
- Performance testing : recherche des meilleurs métriques, benchmark.
- Regression testing
- Sanity testing : contrôle que les bugs ciblés sont bien corrigés
- Security testing : contrôle face aux menaces
- Single user performance testing
- Smoke testing : valider les fonctions critiques d'un système, envers la stabilité.
- Stress testing : test selon des conditions exceptionnelles de charge, au dela des specs.
- Unit testing

White-box testing : teste les entrées sorties d'un logiciel bien connu, concernant le design, l'utilisabilité, la sécurité, la stabilité.

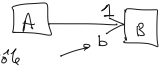
https://junit.org/junit5/docs/current/user-guide/junit-user-guide-5.8.0.pdf



Héritage :

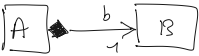
class B

class A : B



class B

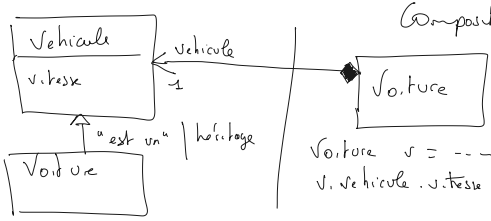
class A
{
B b;
}



class B

```
class A
{
    B b = new B();
}
```

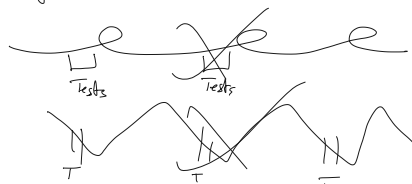
Composition =
- le cycle de vie de B est le même que A
- b ne devra jamais être partagé (à une autre instance que A)

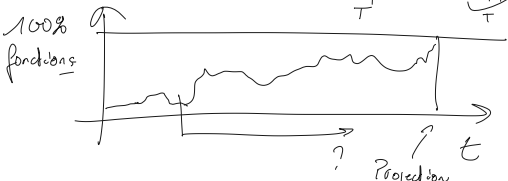


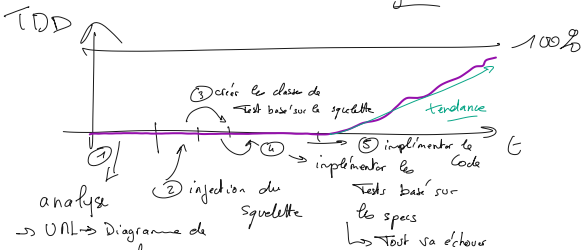
Test Driven Development

=> Methodologie de Projet

- Agile
- Scrum
- TDD







-> UML -> Diagramme de classes

-> Génération d'un squelette de classe

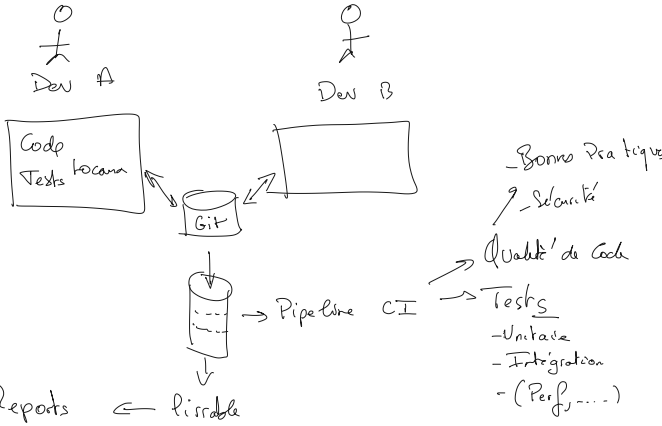
-> Toutes les classes, packages, données membres, méthodes, sans le code

---

Risques d'erreurs et l'écriture des tests

-> On ne peut les confronter

-> On ne Teste pas les Tests



Sécurité → Référentiels

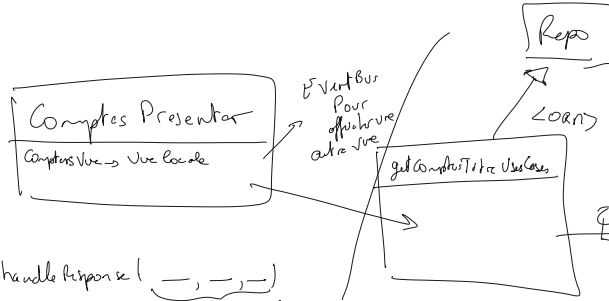
https://www.sans.org/top25-software-errors/

base de toutes les failles recensées : https://nvd.nist.gov

https://www.okta.com/fr/identity-101/whats-the-difference-between-oauth-openid-connect-and-saml/

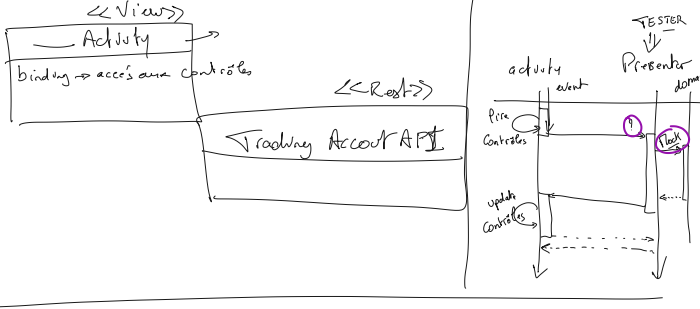
MVVM Avec Flow Junit : https://developer.android.com/kotlin/flow/test

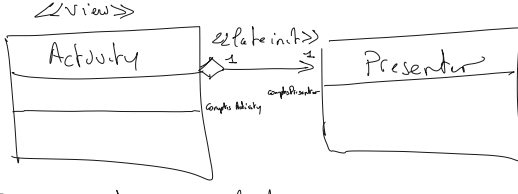
MVP



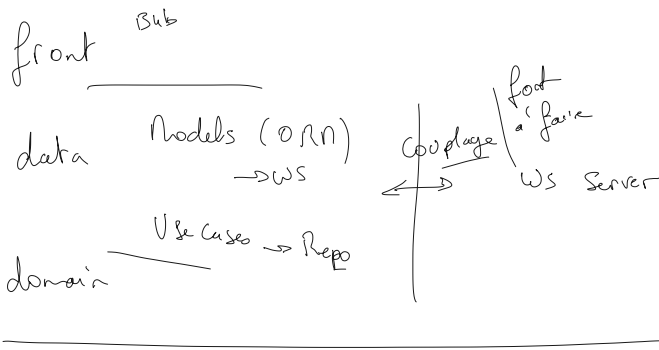
View . handleResponse (— , — , —)

update de la Vue.





Sur event → appel dans le Presenter



Activity → (Testeur)

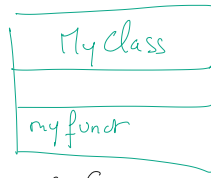
Presenter ⇒ TESTER

data →

Java → tests

Java → code

Uses Cases : → Normaux
Alternatifs
Erreurs



1 Méthode → n Cas -

Question : (1) 1 Méthode de Test par Méthode ou Tester pour tous les cas.
→ Q = Combien de lignes de code ?

(2) 1 Méthode par type de Cas
(2') 3 classes / 1 par type de Cas
(3) 1 Méthode pour chaque Cas
↳ offre la granularité pour les suites
(4) 1 Classe en façade qui délègue dans des classes amies par Méthode, des Méthodes Par cas —

Chaque test devra toujour être lancé depuis un état clean indépendant des tests précédents

1

MyClassTest

testMyFunct

2

MyClassTest

testNormalMyFunct
testAltMyFunct
testErrorsMyFunct

Si un peu trop de code

2'

MyClassNormalTest

myFunct()

Alt

Error

3

MyClassTest

testCase1 MyFunct
testCase2 MyFunct

beaucoup de Méthodes

Avantage
→ les cas sont granuleux dans les Suites

4

MyClassTest

@Test

Milliers de lignes de code

ParMethode

@Test
(par test)

1 Package par classe

déléguer la Technique dans des classes

STATE FULL → Etat se modifie entre deux requête.

STATE LESS → Pas de modif d'état

Cas . statefull → Réinitialiser entre les cas

. Stateless → Réutilisable

Ex: 1 Un SRV web interne pour servir une API.

↳ STATE | Full → Si Session
Less → Si API.

Ex 2 Un client web pour faire des Appels Serveur

STATE | FULL → Cookies
→ HSTS
→ executer JS
→ ...
Less → @beforeEach

Ex 3 @beforeEach

@Test
Cas 1 -
Cas 2 -
STATELESS

@Test Cas 1
@Test Cas 2
STATEFULL

Liste des assertions : https://junit.org/junit5/docs/5.8.0/api/org.junit.jupiter.api/org/junit/jupiter/api/Assertions.html

Injecter un fichier de ressources pour les tests :

```
@ParameterizedTest
  @CsvFileSource(resources = "/data.csv")
  void testDisplayCsvFile( int id, String value, TestReporter reporter )
  {
    reporter.publishEntry("id: "+id, "Value : "+value);
  }
```

Dans pom.xml :

```
<resources>
    <resource>
      <directory>resources</directory>
    </resource>
  </resources>
 </build>
</project>
```

dans src/resources :
data.csv :
1, un
2, deux

Tests conditionnels :

```
/@Tag("deux")
  @Tag("trois")
  @TestFactory
  Iterator<DynamicTest> generateMyTests( TestReporter reporter, TestInfo testinfo )
  {
    ArrayList<DynamicTest> tests=new ArrayList<DynamicTest>();

    tests.add( dynamicTest("Display name: mon test1", () -> assertTrue(true) ));
    if(testinfo.getTags().contains("deux"))
      tests.add( dynamicTest("Display name: mon test2", () -> assertTrue(false) ));

    for(int cpt=0;cpt<1000;cpt++)
      tests.add( dynamicTest("Display name: mon test"+cpt, () -> assertTrue(true) ));

    reporter.publishEntry("Génération des tests");
    return tests.iterator();
  }
```

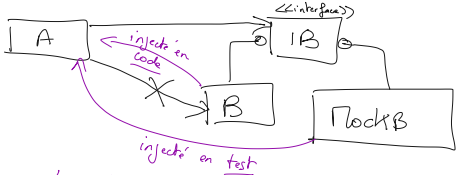
Quelques solutions JUnit 5 :
https://javabydeveloper.com/category/junit-5/

Tests Unitaires

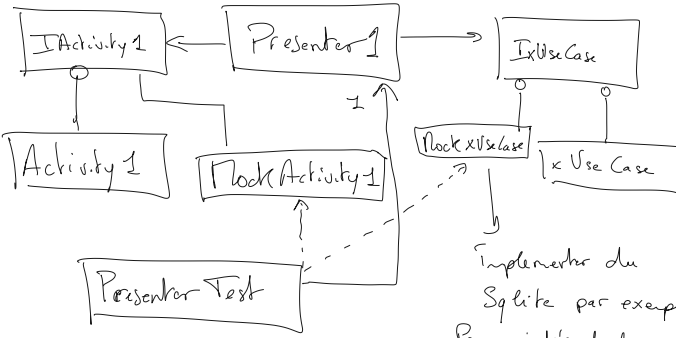
test → A nécessite un B
→ le fonctionnement de A est dépendant du bon " " de B

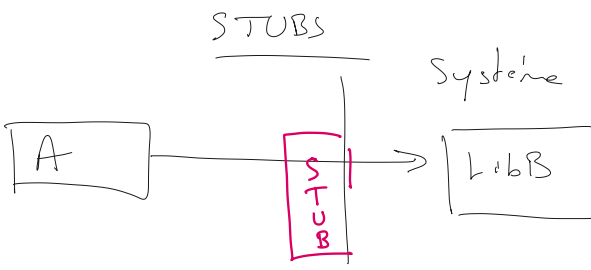
Principe : Utiliser des Mocks → Mocking
"Singer"

Refactoring à faire :



→ l'intérêt de l'IoC → injecter selon le contexte





Tests du MVP :
https://www.pluralsight.com/guides/mvp-with-testing-part-2