

Technique de tests

Master ILSSEN

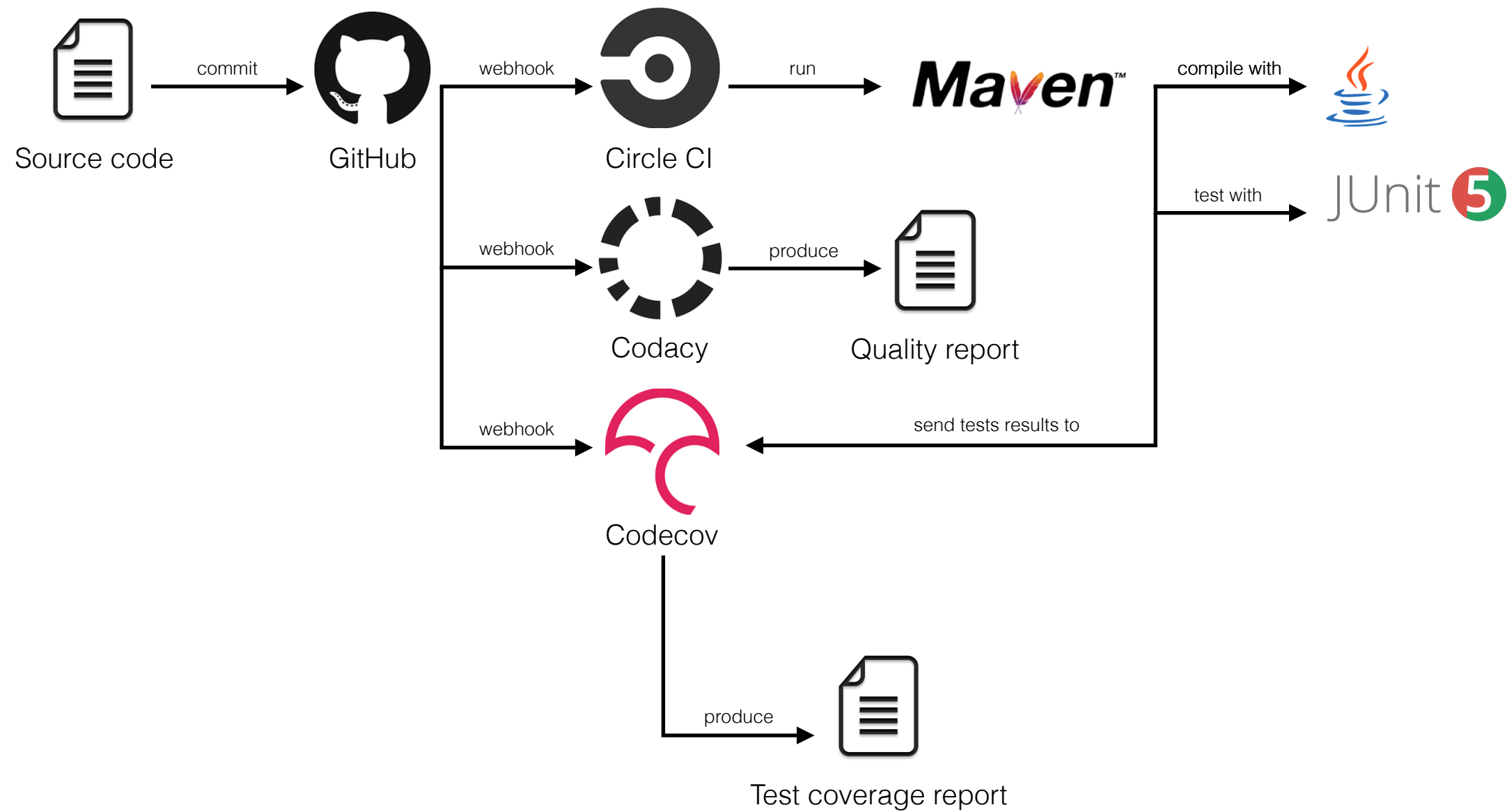
Introduction

Contact : felix.voituret@ismart.fr

- ▶ 6 heures de cours
- ▶ 12 heures de travaux pratiques

Evaluation : rendu de projet

Introduction



Gestionnaire de version

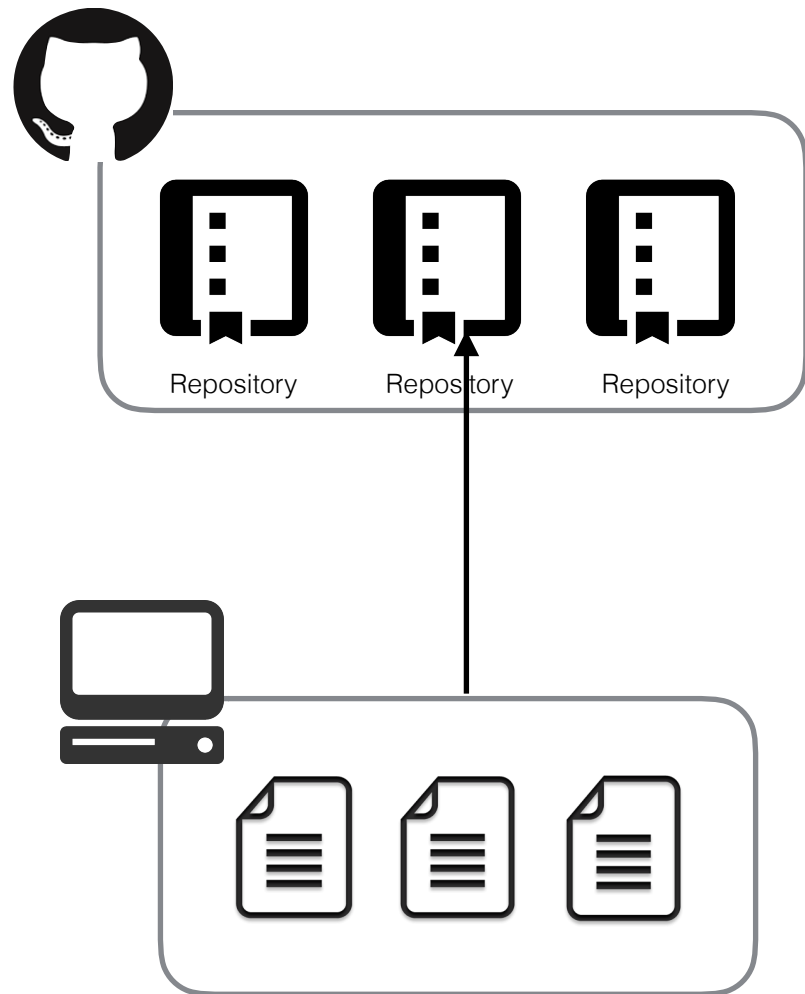
Github



GitHub est un service web d'[hébergement](#) et de gestion de développement de logiciels, utilisant le [logiciel de gestion de versions Git](#).

- ▶ Gestionnaire de code source (versioning, branch, fork, etc ...)
- ▶ Gestionnaire de tickets (issues)
- ▶ Hébergement wiki
- ▶ Serveur web statique basé sur [Jekyll](#)

Repository



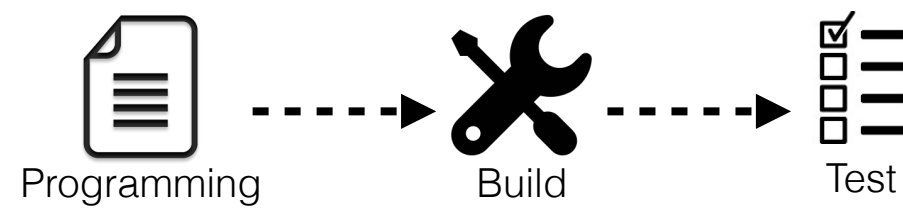
Un repository **GitHub** permet de stocker votre code en ligne, avec une notion de version.

Un repository local est synchronisé avec un repository distant, une opération de **commit** met à jour le repository local, et un **push** synchronise le repository local avec un repository distant.

Intégration Continue

Maven / CircleCI

L'**intégration continue** est un ensemble de pratiques utilisées en génie logiciel consistant à vérifier à chaque modification de code source que le résultat des modifications ne produit pas de **régression** dans l'application développée.



Apache Maven

Apache Maven est un outil pour la gestion et l'automatisation de production logiciel. Les **cycles de vie** basique couvert sont les suivants :

- ▶ compile
- ▶ test
- ▶ package
- ▶ install
- ▶ deploy

```
> mvn goal
```

Project Object Model

Un projet **Maven** est défini par un fichier nommé *POM* (**Project Object Model**) qui définit la configuration des différents cycles de vie de l'application.

```
<project xmlns="http://maven.apache.org/POM/4.0.0">  
  <modelVersion>4.0.0</modelVersion>  
  <groupId>authority</groupId>  
  <artifactId>project_name</artifactId>  
  <version>1.0.0</version>  
</project>
```

Project Object Model

Un projet **Maven** est défini par un fichier nommé *POM* (**Project Object Model**) qui définit la configuration des différents cycles de vie de l'application.

```
<project xmlns="http://maven.apache.org/POM/4.0.0">  
  <modelVersion>4.0.0</modelVersion>  
  <groupId>authority</groupId>  
  <artifactId>project_name</artifactId>  
  <version>1.0.0</version>  
</project>
```

Project Object Model

Un projet **Maven** est défini par un fichier nommé *POM* (**Project Object Model**) qui définit la configuration des différents cycles de vie de l'application.

```
<project xmlns="http://maven.apache.org/POM/4.0.0">  
  <modelVersion>4.0.0</modelVersion>  
  <groupId>authority</groupId>  
  <artifactId>project_name</artifactId>  
  <version>1.0.0</version>  
</project>
```

Un projet **Maven** est défini par un fichier nommé *POM* (**Project Object Model**) qui définit la configuration des différents cycles de vie de l'application.

```
<project xmlns="http://maven.apache.org/POM/4.0.0">  
  <modelVersion>4.0.0</modelVersion>  
  <groupId>authority</groupId>  
  <artifactId>project_name</artifactId>  
  <version>1.0.0</version>  
</project>
```

Un projet **Maven** est défini par un fichier nommé *POM* (**Project Object Model**) qui définit la configuration des différents cycles de vie de l'application.

```
<project xmlns="http://maven.apache.org/POM/4.0.0">  
  <modelVersion>4.0.0</modelVersion>  
  <groupId>authority</groupId>  
  <artifactId>project_name</artifactId>  
  <version>1.0.0</version>  
</project>
```

Project Object Model

Les paramètres de constructions sont indiqués dans un noeud **build**. Des **profiles** permettent de définir des configurations de build.

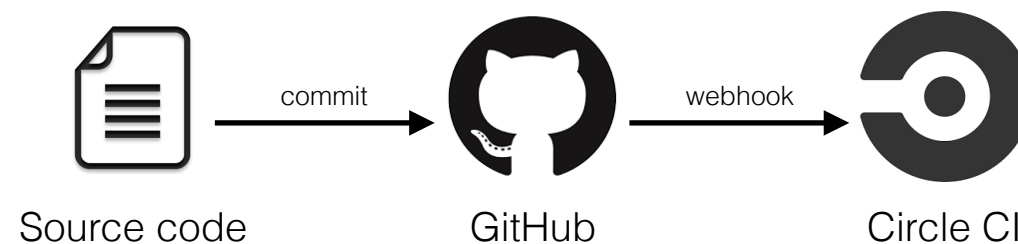
```
<project xmlns="http://maven.apache.org/POM/4.0.0">
  ...
  <build>...</build>
  <profiles>
    <profile>
      <build>...</build>
    </profile>
  </profiles>
  ...
</project>
```

Un grand nombre d'**artifact maven** sont disponible sur un repository nommée **Maven Central** et peuvent être intégré dans n'importe quel projet en spécifiant un noeud **dependencies** dans le *POM*.

```
<project xmlns="http://maven.apache.org/POM/4.0.0">
  ...
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
    </dependency>
    ...
  </dependencies>
  ...
</project>
```




CircleCI est une plateforme cloud offrant des services d'**intégration continue** dans des **conteneurs**. Un compte gratuit permet d'avoir un conteneur d'exécution gratuitement relié à n repository (**Github** ou **Bitbucket**)



Après un commit sur votre repository **Github**. Un [conteneur](#) est créée, et cherche un fichier `circle.yml` à la racine, décrivant les étapes et la configuration de votre [intégration continue](#) :

- ▶ machine
- ▶ checkout
- ▶ dependencies
- ▶ database
- ▶ compile
- ▶ test
- ▶ deployment

Test unitaire

JUnit

Test unitaire

Un **test unitaire** est un test s'assurant du bon fonctionnement d'une partie de code ou unité.

```
public final class SumTest {  
  
    @Test  
    public void testSum() {  
        final int sum = 2 + 2;  
        assertEquals(4, sum);  
    }  
}
```





powered by JUnit 

JUnit est un framework pour l'écriture et l'exécution de test unitaire pour **Java**, intégré nativement dans la plupart des outils de développement actuel.

Test case : Test couvrant une portion de l'application.

Test suite : Ensemble de test case.

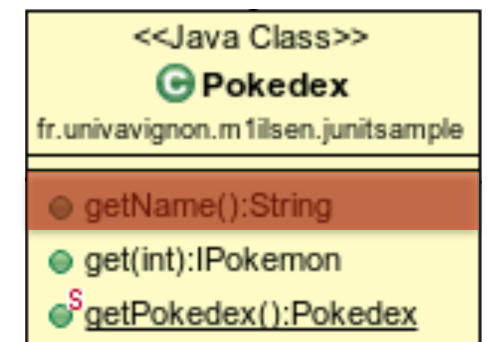
Test case

<<Java Class>>	
 Pokedex	
fr.univavignon.m1ilsen.junitsample	
	getName():String
	get(int):IPokemon
	<u>getPokedex():Pokedex</u>

Test case

Un **test case** est une simple classe Java disposant d'au moins une méthode annoté avec `@org.junit.Test` :

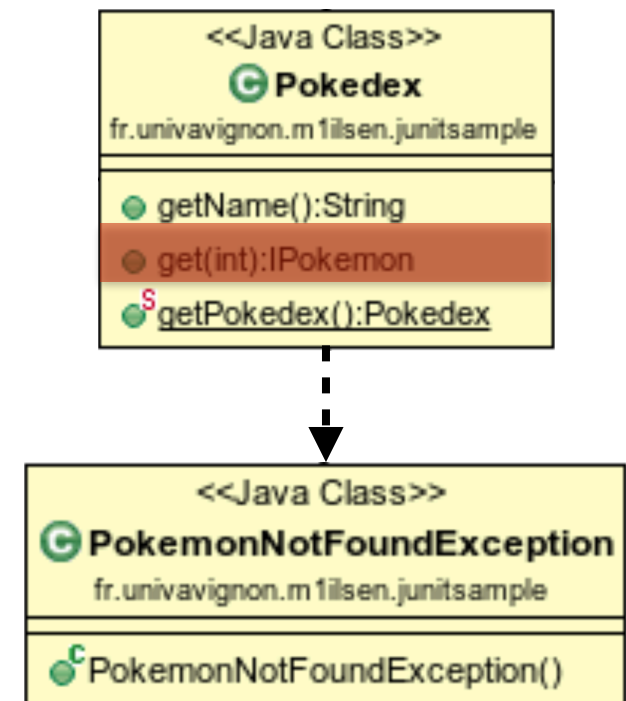
```
public final class PokedexTest {  
  
    @Test  
    public void testName() {  
        final Pokedex pokedex = Pokedex.getPokedex();  
        final String name = pokedex.getName();  
        assertEquals("Dexter", name);  
    }  
}
```



Test case

Pour tester les exceptions, un paramètre **expected** peut être donné à l'annotation `@org.junit.Test` :

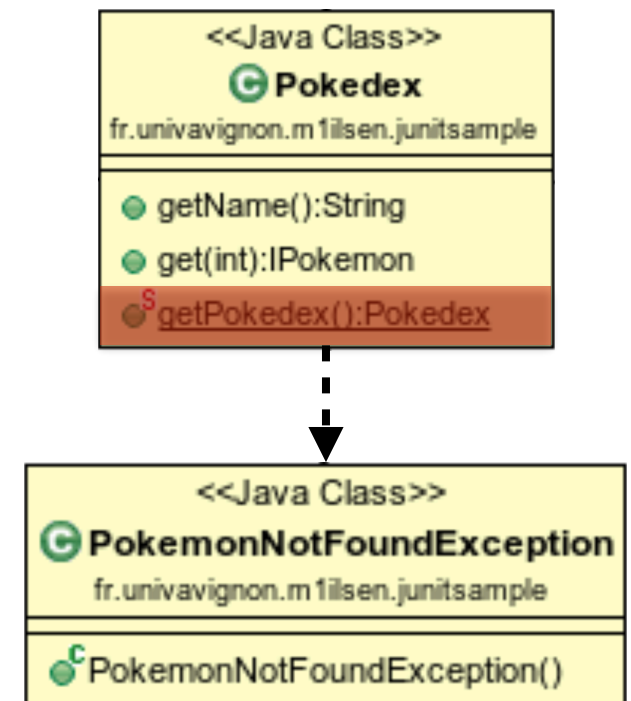
```
public final class PokedexTest {  
  
    @Test(expected=PokemonNotFoundException.class)  
    public void testPokemonNotFoundException() {  
        final Pokedex pokedex = Pokedex.getInstance();  
        pokedex.get(-1);  
    }  
}
```



Test case

L'annotation `@org.junit.Before` permet de définir une méthode exécutée avant chaque test :

```
public final class PokedexTest {  
  
    private Pokedex pokedex;  
  
    @Before  
    public void setUp() {  
        this.pokedex = Pokedex.getPokedex();  
    }  
  
    @Test  
    public void testName() {  
        assertEquals("Dexter", pokedex.getName());  
    }  
  
    @Test(expected=PokemonNotFoundException.class)  
    public void testPokemonNotFoundException() {  
        pokedex.get(-1);  
    }  
}
```



Test case

Un équivalent statique est l'annotation `@org.junit.BeforeClass` qui sera exécuté une seule fois avant tous les tests :

```
public final class PokedexTest {

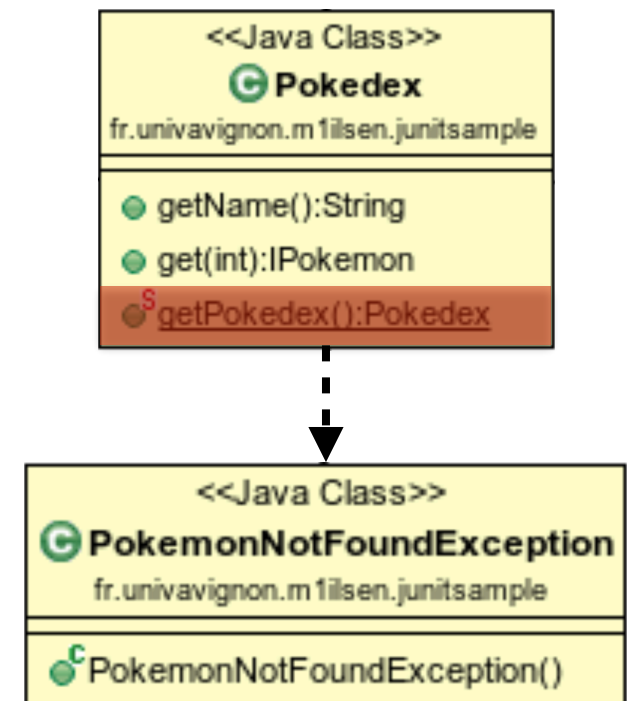
    private static Pokedex pokedex;

    @BeforeClass
    public static void setUp() {
        pokedex = Pokedex.getPokedex();
    }

    @Test
    public void testName() {
        assertEquals("Dexter", pokedex.getName());
    }

    @Test(expected=PokemonNotFoundException.class)
    public void testPokemonNotFoundException() {
        pokedex.get(-1);
    }

}
```



Mock

Un **mock** est une implémentation factice d'une interface permettant de tester ces dernières ainsi que leur interaction.



```
public final class PokemonMock {

    @Override
    public int getIndex() {
        return 1;
    }

}
```

Test case

```
public final class PokedexTest {

    private static Pokedex pokedex;

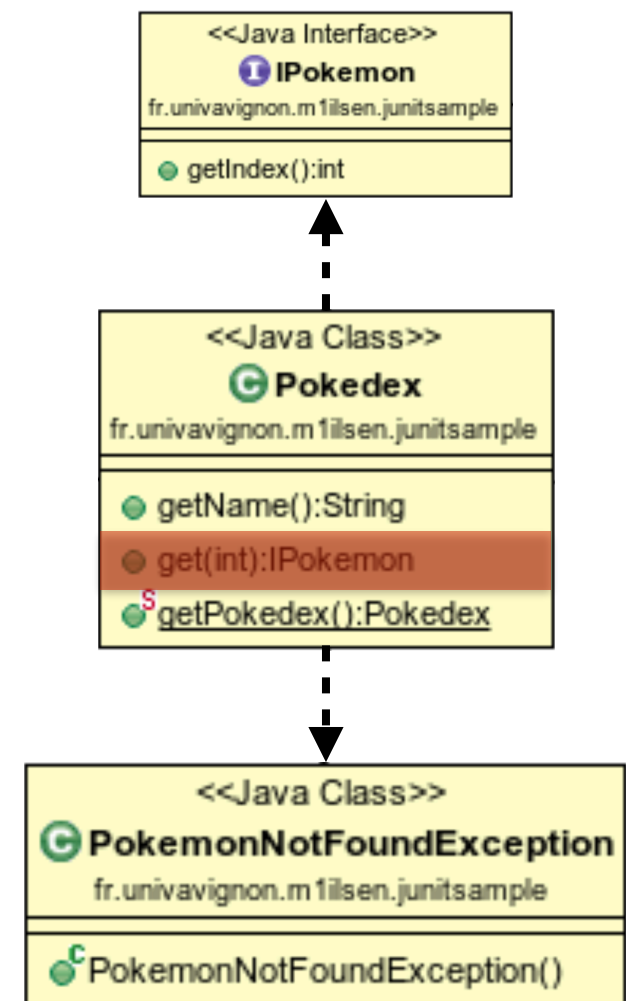
    @BeforeClass
    public static void setUp() {
        pokedex = Pokedex.getPokedex();
    }

    @Test
    public void testName() {
        assertEquals("Dexter", pokedex.getName());
    }

    @Test
    public void testGet() {
        final IPokemon pokemon = pokedex.get(1);
        assertEquals(1, pokemon.getIndex());
    }

    @Test(expected=PokemonNotFoundException.class)
    public void testPokemonNotFoundException() {
        pokedex.get(-1);
    }

}
```



Test case

Annotation disponible pour l'écriture d'un **test case** :

- ▶ @org.junit.Test
- ▶ @org.junit.Before
- ▶ @org.junit.After
- ▶ @org.junit.BeforeClass
- ▶ @org.junit.AfterClass

Test case

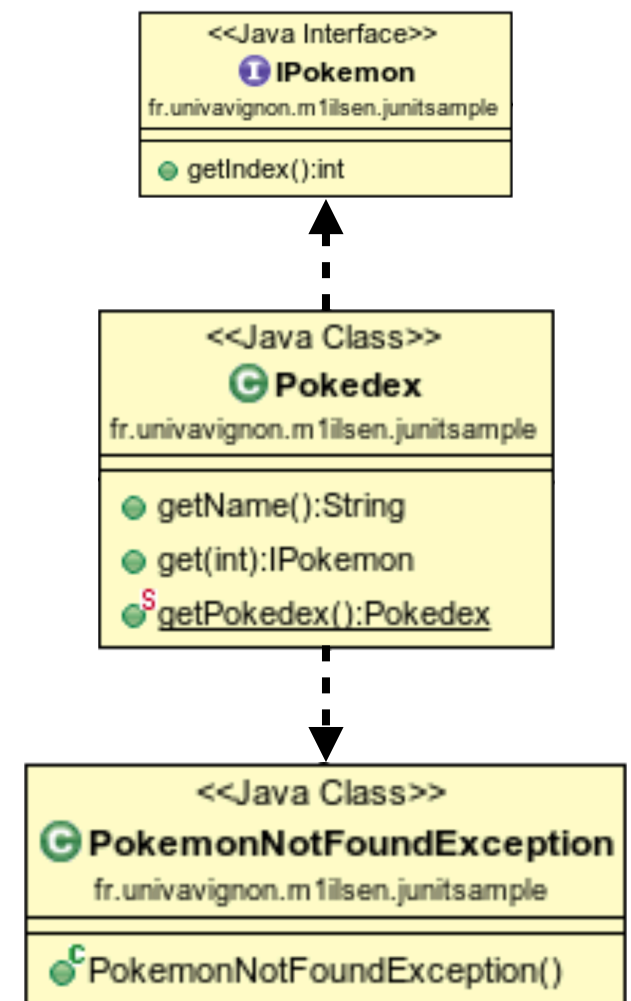
Méthodes disponible pour l'écriture d'un **test** :

- ▶ assertEquals
- ▶ assertTrue
- ▶ assertFalse
- ▶ assertNull
- ▶ assertNotNull
- ▶ fail

Test suite

Une **test suite** consiste en l'exécution de plusieurs **test cases** défini par l'annotation `@org.junit.runner.Suite`:

```
@RunWith(Suite.class)
@Suite.SuiteClasses(
    PokedexTest.class,
    PokemonTest.class
)
public final class AllTests {
}
```



Mock

Mockito

Mock

Un **mock** est une implémentation factice d'une interface permettant de tester ces dernières ainsi que leur interaction.



```
public final class PokemonMock {

    @Override
    public int getIndex() {
        return 1;
    }

}
```

Mockito



Mockito est un framework Java permettant de générer automatiquement des **mock**. Pour l'intégrer à votre projet il suffit de rajouter la dépendance **Maven** suivante :

```
<dependency>  
  <groupId>org.mockito</groupId>  
  <artifactId>mockito-all</artifactId>  
  <version>1.9.5</version>  
</dependency>
```

Pour créer un **mock** il suffit d'annoter un attribut de la classe de test avec l'annotation **@Mock**.
La règle de test **MockitoRule** permet l'instantiation automatique des attributs annotés.

```
public final class PokemonTest {  
    @Mock private IPokemon pokemonMock;  
    @Rule public MockitoRule mockitoRule = MockitoJUnit.rule();  
}
```

Mockito fourni des méthodes statiques pour configurer les résultats lors des appels de méthodes mockées.

```
when(pokemonMock.getName()).thenReturn(« Bulbizarre »)
```

```
when(pokemonMock.getName()).thenThrow(new Exception())
```

Une fois les **mocks** utilisés dans vos tests unitaires, leur utilisation peut être contrôlée grâce à la méthode statique **verify**.

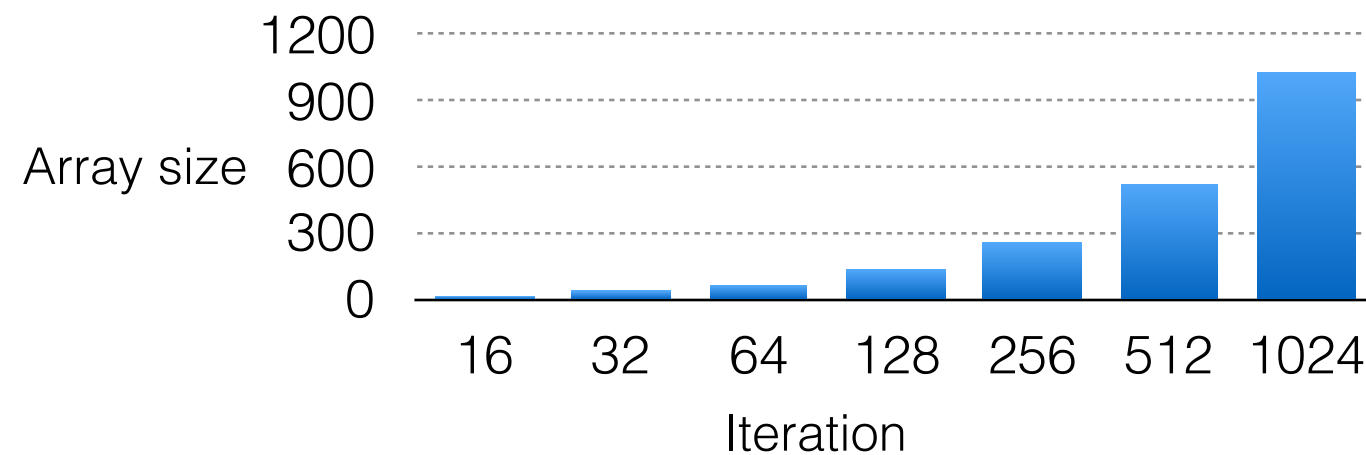
```
verify(test).testing(Matchers.eq(12));  
verify(test, times(2)).getUniqueId();  
verify(mock, never()).someMethod("never called");  
verify(mock, atLeastOnce()).someMethod("called at least once");  
verify(mock, atLeast(2)).someMethod("called at least twice");  
verify(mock, times(5)).someMethod("called five times");  
verify(mock, atMost(3)).someMethod("called at most 3 times");
```

Qualité de code

Codacy

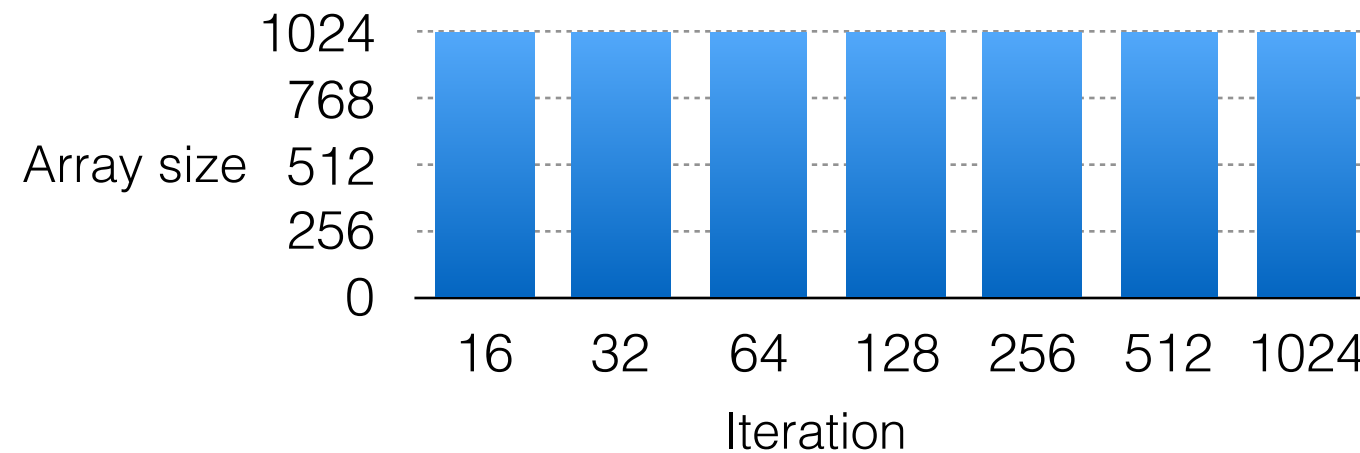
Bien connaître les API standards de **Java** permet d'optimiser les performances :

```
final List<Integer> list = new ArrayList<>();  
for (int i = 0; i < 1024; i++) {  
    list.add(i);  
}
```



Bien connaître les API standards de **Java** permet d'optimiser les performances :

```
final List<Integer> list = new ArrayList<>(1024);  
for (int i = 0; i < 1024; i++) {  
    list.add(i);  
}
```



Bien connaître les API standards de **Java** permet d'optimiser les performances :

```
final String text = "foo" + "bar" + "coin";
```

```
final StringBuilder builder = new StringBuilder();  
builder.append("foo");  
builder.append("bar");  
builder.append("coin");  
final String text = builder.toString();
```