Faculté des Sciences
Aix*Marseille Université

Département Informatique et Interactions
Juillet 2017

JDEV

Fiabilité Logicielle en Java

Atelier JDEV

Cet atelier présente une introduction à la fiabilité logicielle en Java en utilisant JUnit. Les outils utilisés sont présentés sur des exemples simples d'application, ce qui permet de rapidement entrevoir leur utilisation et les apports des frameworks abordés.

Vous pouvez récupérez les sources Java des exercices sur la page de l'atelier sur le site des Jdev `http://devlog.cnrs.fr/jdev2017/t4.a02`.

Crédits : Romaric Duvignau, Denis Lugiez, et les étudiants du cours Fiabilité Logiciel 1 de M2 FSIL 2016 pour les programmes à tester (attention, ça peut parfois piquer les yeux).

1 Rapide tour de JUnit en mode piqûre de rappel

Les environnements de tests unitaires existent pour tous les langages de programmation classiques. Ils permettent d'écrire des suites de tests unitaires pour un programme donné, puis l'environnement exécute cette suite et renvoie le résultat de l'exécution des tests. JUnit est l'environnement pour les programmes Java (Cunit pour C, PHPUnit pour PHP,....). Ces environnements peuvent s'utiliser indépendamment ou bien être intégrés dans un Atelier de Génie Logiciel (IDE) comme Éclipse. La version courante est la JUnit4 (la version précédente JUnit3 a une syntaxe proche mais incompatible).

Ce que permet JUnit:

- JUnit permet d'écrire des tests unitaires pour Java.
- JUnit permet l'écriture simultanée d'une classe et d'une classe de tests.
- JUnit permet de rejouer automatiquement les tests à chaque modification du programme.

Ce que JUnit ne fait pas:

- JUnit n'est pas une méthode d'élaboration de tests et la suite de tests est définie par le programmeur.
- JUnit n'est pas conçu pour les tests d'intégration.

De nombreuses informations et exemples peuvent être récupérées à partir du site `http://junit.org` Dans un projet Eclipse, il est nécessaire de signaler l'utilisation de la librairie JUnit (à rajouter via l'onglet propriété) ce qui permettra d'avoir dans le menu l'onglet de création d'une classe de tests (différente de l'onglet de création d'une classe Java).

1.1 Principe d'une suite de tests

JUnit exécute une suite de tests qui comporte plusieurs phases:

1. Une phase d'initialisation (Set-Up).

   Cette phase sert à déclarer ou initialiser des éléments qui sont utilisés par la suite (test fixture). Par exemple, se connecter à une base de donnée qui sera interrogée par les fonction testées ou initialiser un tableau que les fonctions testées manipulent.

2. La suite des tests unitaires.

   Chaque test peut également être précédé d'une phase d'initialisation (par exemple remettre toutes les entrées d'un tableau à 0) et suivi d'une phase de nettoyage. Un test unitaire est conçu pour ne vérifier qu'une fonctionnalité unique (d'où son nom).

3. Une phase de clôture de la phase de test (Tear-down). Cette phase sert à retourner un environnement propre (clean-up); Par exemple se déconnecter proprement d'une base de donnée, fermer des fichiers,...

L'environnement JUnit permet d'écrire les suites de tests alors même que les classes à tester ne sont pas écrites !

Astuce 1. Une fonctionnalité ne signifie pas forcément une fonction ! Même si, pour des fonctions simples, un seul test unitaire peut suffire, on écrit généralement plusieurs tests afin de vérifier le comportement d'une méthode. N'oubliez pas que l'un des buts du test unitaire est de déterminer rapidement la portion de code fautive lorsqu'un test unitaire échoue.

1.2 Ecriture des tests

Une fois créé le squelette de la classe de test (via l'onglet du menu), il faut écrire les tests. Un test est une méthode précédée de l'annotation JUnit `@Test`. Un test utilisera les assertions de la classe `Assert` qui seront utilisées pour vérifier des égalités (de valeurs, objets, ...) et feront échouer le test si elles ne sont pas vraies. Afin d'éviter d'introduire de nouveaux bugs à l'intérieur des tests, il est important que de les conserver le plus simple possible, garantissent ainsi une relecture rapide.

Exemple :

```
@Test
public void testAdditionZero(){
    expected=1;                     //la valeur espérée
    val=MaClasse.addition(1,0);     //la valeur calculée
    Assert.assertEquals("test 0 neutre pour addition",expected,val);
                                    //comparaison entre les deux valeurs
}
```

Quelques assertions possibles (prenant aussi un premier argument de type `String` optionnel donnant un message explicitant l'assertion) :

- – `assertEquals(long expected, long actual)`
  – `assertEquals(Object expected, Object actual)`
  – `assertEquals(double expected, double actual, double delta)`, égalité de *double* à *delta* près
- `assertFalse(boolean condition)` et `assertTrue(boolean condition)`
- `assertNotNull(Object object)` et `assertNull(Object object)`
- `assertNotSame(Object unexpected, Object actual)` et son complément `assertSame`
- `fail()`, une assertion échouant à tous les coups.

D'autres annotations JUnit permettent de gérer les initialisations et cas particuliers. Elles sont de la forme `@motClé`.

- `@BeforeClass` et `@AfterClass` permettent d'exécuter des instructions avant et après l'exécution de la suite de tests.

- `@Before` permet de définir des initialisations à faire avant chaque test (typiquement définir un objet qui sera utilisé par tous les tests) et `@After` est similaire mais est effectué après chaque test.
- `@Ignore` permet de ne pas effectuer le test qui suit.
- `@Test(expected=MonException.class)` teste si la méthode déclenche bien une exception de la classe *MonException*.
- `@Test(timeout=val)` fera échouer le test quand le temps d'exécution dépasse la valeur *val* de timeout (donnée en millisecondes).

Le site `junit.org` donne une description plus détaillée de l'outil.

1.3 exo1.item

Exercice. Le package *exo1.item* contient des classes utilisées par une application d'achat en ligne. La classe `Item.java` correspond aux articles, et les classes `ItemsSortedList{Bad,Good}.java` au panier dans lequel les éléments sont rangés par prix croissant. `ItemsSortedListBad` implémente une recherche dichotomique d'un élément dans le panier, alors que `ItemsSortedListGood` une recherche linéaire.

1. Importez les sources java de l'atelier dans votre IDE préféré en créant un nouveau projet et en le paramétrant pour utiliser JUnit. Sous éclipse, *Build Path* → *Add Librairies* → *JUnit* ou pour maven, ajoutez à votre `pom.xml` :

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
</dependency>
```

2. Définir les tests à effectuer pour la fonction de recherche d'un élément (pour chacune des classes données).
3. Peut-on tester si la fonction `isPresentArticle(Item item)` utilise effectivement une recherche dichotomique ?
4. La fonction `isSortedList` n'est pas implémentée.
   (a) Donner une spécification précise de cette fonction.
   (b) Écrire une classe de test permettant de valider une fonction réalisant les spécifications.
   (c) Uniquement maintenant, écrire le code de la fonction et valider votre implémentation.

Astuce 2. Rappels :

- la classe testant `MaClasse` est généralement appelée `MaClasseTest`;
- les classes de tests sont généralement placées dans une architecture parallèle à `src`, *i.e.* à `src/exo1.item/ISL.java` correspondra `test/exo1.item/ISLTest.java` – sous maven, on trouve de manière équivalent `src/main/java` et `src/test/java`;
- le plus souvent, une seule assertion permet de valider un test unitaire.

1.4 exo1.triangle

La classe *Triangle* implémente un Triangle.

Spécification de la classe:

- La classe `Triangle` contient un constructeur `Triangle(double a, double b, double c)` qui initialise un objet de côtés a, b, c et lève une exception de type `IllegalArgumentException` lorsque les paramètres ne forment pas un triangle (si l'un des côtés est supérieure à la somme des deux autres).
- La classe contient une fonction publique `int type()` qui renvoie un des 7 types suivants:
  0. scalène (3 côtés de longueur $\neq$) acutangle (que des angles aigus $< 90°$)
  1. scalène obtusangle (avec un angle $> 90°$)
  2. scalène rectangle (avec un angle droit $= 90°$)
  3. isocèle (exactement deux côtés égaux) acutangle
  4. isocèle obtusangle
  5. isocèle rectangle
  6. équilatéral (tous les côtés de même longueurs)
- `public static Triangle read(String filename) throws IOException` qui lit un fichier texte contenant les résultats des mesures des trois cotés d'un triangle (une valeur par ligne) et affecte les attributs privés `coteA`, `coteB`, `coteC` de type `double` de la classe aux valeurs lues. Si le fichier n'existe pas ou s'il ne correspond pas à trois valeurs de type double, la méthode déclenche une exception de type `IOException`.
- `public static void write(Triangle t, String filename) throws IOException` écrit dans le fichier *filename* les côtés du triangle *t*, de manière à être compatible avec la fonction `Triangle.read()`.

Tests

- Écrire une suite de tests pour la classe `Triangle`.

Astuce 3. Trois méthodes pour tester les exceptions :

- via l'annotation `@Test(expected=IOException.class)`, la plus simple et adaptée si on n'a besoin de tester que la classe de l'exception levée;
- via une règle :

```
@Rule
public ExpectedException thrown = ExpectedException.none();

@Test
public void test() throws MonException{
    thrown.expect(MonException.class);
    thrown.expectMessage("expected message");
    ...
}
```

- via un classique try/catch (permet d'accéder à tous les paramètres de l'exception) :

```
try{
        ...
        fail("échec");
}
catch(MonException e){
        ...
};
```

1.5 exo1.date

La classe *Date* est une implémentation Java partielle de l'objet *date* du module Python *datetime*.
Elle contient notamment les fonctions suivantes :

- `int toOrdinal()` convertit la date dans un entier correspondant au nombre de jour depuis le début du calendrier Grégorien.
- `Date fromOrdinal(int i)` correspond à l'opération inverse.

La spécification précise de chaque méthode est donnée par la page de documentation python3 `https://docs.python.org/3.3/library/datetime.html#datetime.date`. Pour information, le calendrier `grégorien proleptique` est le calendrier gréorien (celui actuellement en vigueur dans la grande majorité du monde) étendu de sorte que toute date passée et future appartienne aussi au calendrier grégorien. Dans ce calendrier, seules les années multiples de 4 mais non multiple de 100, et les années multiples de 400 sont bissextiles (contiennent un 29 février). De plus, la doc python précise que l'année minimum est 1 et l'année maximum est 9999.

Exercice Écrire une suite de tests permettant de vérifier que les deux fonctions implémentent correctement la spécification. Vous prendrez soin de choisir des valeurs pertinentes pour vos tests. Pour cela, en utilisant le principe du **test en Boîte noire** (code source inaccessible), vous devez anticiper les comportements différents (appelés aussi classes d'équivalence) attendus en fonction de l'entrée, puis choisir un cas de test par classe. De plus, il est intéressant de tester les valeurs limites, ou extrémales des classes d'équivalence pour détecter les erreurs du type $\leq$ à la place de $<$, etc. Tester avec certaines des implémentations fournies.

Vous pouvez obtenir les valeurs attendues aisément via Python :

```
>>> from datetime import *
>>> date.fromordinal(1)
datetime.date(1, 1, 1)
>>> date.fromordinal(32)
datetime.date(1, 2, 1)
>>> date(2016,10,17).toordinal()
736254
```

1.6 Utilisation d'un outil de couverture de code

Installer le plugin emma [1] via la marketplace d'eclipse.

Pour améliorer la qualité de nos tests, nous allons utiliser le principe de **test en Boîte blanche**, *i.e.* le code source va être utilisé pour accroitre la fiabilité dans le logiciel.

Vérifier la couverture des tests avec Emma (Onglet *Coverage as* $\rightarrow$ *JUnit Test*). Une ligne de code est considérée comme couverte si elle a été exécutée lors du batch de test. Toutes les branches d'une expression conditionnelle doivent être parcourues pour considérer la ligne correspondante entièrement couverte.

Ajouter les tests nécessaires afin d'atteindre une couverture proche de 100%. Étudier la redondance éventuelle de vos tests en supprimant ou refactorisant un ou plusieurs tests puis en recalculant la couverture. Vérifier que les classes testées ne contenaient pas de code mort (impossible à atteindre).

2 Objets factices

Récupérer Jmockit sur le site officiel `http://jmockit.org/` et ajouter les `.jar` dans le projet, ou via la dépendence maven

[1] voir `http://eclemma.org/` pour la documentation

```
<dependency>
    <groupId>org.jmockit</groupId>
    <artifactId>jmockit</artifactId>
    <version>1.8</version>
    <scope>test</scope>
</dependency>
```

L'outil `jmockit` est riche et vous devez lire la documentation pour découvrir toutes ses fonctionalités (les tutoriels GettingStarted et Mocking sont un bon début). Attention, placer JMockit avant JUnit dans le classpath ou rajouter l'annotation `@RunWith(JMockit.class)` avant la classe junit (voir tutoriel *GettingStarted*).

2.1 Présentation

Un **objet factice** est un objet virtuel qui simulera le comportement de certains objets lors des tests, par exemple lorsque le comportement de ces derniers n'est pas reproductible (appel à un service externe, non-déterminisme, ...) ou pour découpler des classes lors du test. L'outil *jMockit* construit les (méthodes des) objets factices à l'exécution et il n'est pas nécessaire de définir les classes factices. Pour le test, il suffit de définir le comportement des méthodes des objets factices. Par exemple, on peut spécifier qu'une méthode est appelée avec certains arguments, renvoie une certaine valeur, est appelée n fois, ...

Dans les tests, on distingue 3 phases : (1) spécification (`Expectations()` qui décrit les appels factices attendus (arguments, valeur de retour, nombre d'appel,...) et les enregistre, (2) le rejeu qui doit se conformer à la spécification et exécute le code instrumenté (qui appelera les méthodes factices), (3) la vérification (`Verifications()`) qui vérifie que l'exécution a bien effectué certains appels factices. On peut également mettre des assertions JUnit dans le code instrumenté. La partie (1) ou la partie (3) peuvent être absentes. `Expectations` et `Verifications` peuvent être strictes ou non et ont des rôles complémentaires. Un mocking stict (voir StrictExpectations et VerificationsInOrder/FullVerifications) attend les appels dans le même ordre que celui fourni.

Un exemple simple : `calc` est un objet qui implémente une interface avec une méthode `getTaux` qui est déclarée factice dans la classe de test par :

```
@Mocked ITauxChange conv;

@Test
public void testEuro2euroOnce() {
        new Expectations (){{
                conv.getTaux(anyString,anyString);
                result=1.0;
                times=1;
            }
        };
        calc =new DeviseCalcWithInterface(conv);
        double expected=1.0;
        double value=calc.euro2euro(1.0);
        Assert.assertTrue(expected==value);
}
```

Afin que l'on puisse effectivement simulée l'objet moquée, il faut par contre que l'implémentation le permette en utilisant par exemple via l'**injection de dépendance**. Ici, la classe à testée `DeviseCalcWithInterface` prend bien soin de ne pas instancier l'objet de type `ITauxChange` dont il a besoin. Ainsi, on peut facilement remplacer l'objet attendu par un objet factice.

2.2 Test d'une application carte bancaire

Un terminal de paiement par carte bancaire permet de lire un numéro de carte, de l'authentifier via un service externe de validation, puis au possesseur de la carte de se connecter en entrant son code secret. Si celui-ci est correct, la carte est connectée. Si le numéro de carte n'est pas légal, le terminal refuse la connection, et si l'utilisateur entre plus de 4 fois un code erroné le terminal invalide la carte.

Il s'agit de tester la classe *connectToTerminal* qui réalise les opérations du terminal. Elle utilise les services des classes `Card` et `Validator`. Elle contient un attribut privé `connectedCard` de type `Card` et la méthodes publiques :

- `public boolean validateCardNumber(int cardnumber)` qui interroge son validateur (type `Validator`) et peut déclencher une exception `IllegalCard` et sinon met à jour l'attribut privé `connectedCard`;
- `public void authenticateCode(int secretCode)` qui peut renvoyer une exception de type `NumberOfTryExceeded` ou `IllegalCard`.

La classe `Validator` contient la méthode publique `Card validateCard(int number)` qui renvoie la carte correspondant au numéro de carte, sinon `null`.

La classe `Card` a les méthodes suivante

- `boolean isConnected()` renvoie *true* si la carte est connectée, $false$ sinon;
- `void setConnection(boolean v)` établit le statut de connection de la carte à v;
- `boolean isValid()` qui renvoie la valeur d'un attribut indiquant si la carte est valide ou non;
- `public void setValidation(boolean v)` qui assigne l'attribue précédent à v;
- `boolean checkSecretCode(int code)` qui renvoie *true* si le code secret est égal à *code*, $false$ sinon.

Exercice

1. Définir les tests nécessaires pour la classe `connectToTerminal`.
2. Écrire la classe `connectToTerminal` et la tester en utilisant jmockit pour simuler les classes `Card` et `Validator`.
3. Ajouter une fonctionnalité pour retirer du cash lorsque la carte est connectée, et tester-la !

3 Test sur Base de Données

Cet exercice traite du test automatisé de base de donnée.

3.1 Créer la base de donnée

Créer[2] une base de données permettant de gérer les soutenances de stage des étudiants sachant que :

1. Un étudiant a un nom, prénom, statut (fsi ou isl) et un numéro d'étudiant.
2. Une plage de soutenance a un jour (1 à 5), une plage horaire (1 à 8 pour 4 plages le matin de 8h à 12h, 4 plages l'après-midi de 14h à 18h) et un numéro de salle.
3. Une soutenance a un titre, un étudiant et une plage de soutenance.

[2]Dans l'atelier, nous utiliserons SQL.

Vous pouvez créer la base à partir du script SQL suivant (disponible dans le package exo3) :

```
CREATE DATABASE soutenances;
GRANT ALL PRIVILEGES ON soutenances.* TO 'id'@'localhost' IDENTIFIED BY "password";
USE soutenances;
CREATE TABLE etudiant(nom VARCHAR(25), prenom VARCHAR(25), statut VARCHAR(3), numero INT PRIMARY KEY NOT NULL);
CREATE TABLE plage(id INT PRIMARY KEY NOT NULL AUTO_INCREMENT, jour INT, horaire INT, salle VARCHAR(25));
CREATE TABLE soutenance(id INT PRIMARY KEY NOT NULL AUTO_INCREMENT, titre VARCHAR(255),
etudiant INT, plage INT, FOREIGN KEY(etudiant) REFERENCES etudiant(numero), FOREIGN KEY(plage) REFERENCES plage(id));

CONSTRAINT pk_filesName PRIMARY KEY (files_name)

EXIT;
```

3.2 Une application Java

La base de données est utilisée par une application Java. Celle-ci comporte les classes `Etudiant`, `Soutenance`, `Plage` et `Admin`. Les trois premières comportent les `getters` et `setters`, la dernière sert à administrer les soutenances et à accéder à la base de données. Elle permet d'ajouter ou supprimer un étudiant, une soutenance ou une plage de soutenance. Ajout ou suppression lèvent une exception (à définir) si les objets existent déjà ou n'existent pas. L'administration a également la fonction `modifiePlageSoutenance(Soutenance sout, Plage oldPlage, Plage newPlage)` permettant de remplacer la plage horaire `oldPlage` d'une soutenance `sout` par une nouvelle plage horaire `newPlage`.

a) Écrire une suite de tests permettant de vérifier le bon fonctionnement des fonctions :

- de création d'une soutenance, étudiant, plage de soutenance,
- de suppression d'une soutenance, étudiant, plage de soutenance,
- de modification d'une soutenance.

3.3 Écriture de tests avec DbUnit

DbUnit est un logiciel permettant d'effectuer des tests unitaires sur les bases de données en utilisant Junit et en utilisant un format interne à l'outil pour représenter les données d'une base de données. Les .jar (version 2.4.9 en téléchargement) seront à ajouter aux bibliothèques du projet (il peut être nécessaire d'installer d'autres .jar: par exemple `slf4j-simple-1.7.21.jar` et `slf4j-api-1.7.21.jar` de SLF4J), ou via maven

```
<dependency>
    <groupId>org.dbunit</groupId>
    <artifactId>dbunit</artifactId>
    <version>2.5.3</version>
</dependency>
```

Les données récupérées par dbunit sont dans des fichiers de format `flat XML file` qui peuvent être lus ou écrits (à la main ou par l'outil à partir d'une base de données). Cela va permettre d'initialiser les bases de données systématiquement, puis de pouvoir comparer les résultats obtenus aux résultats souhaités. Un test lancera l'opération à tester sur une base de données initialisée avec les bonnes données et le résultat obtenu se comparera avec ce qui est attendu (résultats et valeurs attendues étant dans le même format de données *IDataSet*). Un élément de ce type s'obtient soit via une connection à la base, soit par lecture d'un fichier flat XML.

1. Lire la documentation `dbunit` qui explique le fonctionnement et donne des exemples de classes de test. Avant de commencer à écrire la moindre ligne de code, il est fortement conseillé de lire les exemples et la partie Core Components.

2. Réécrire les tests précédents en utilisant `dbunit`. Quelques informations utiles (cf documentation).

- L'interface `IDatabaseConnection` représente une connexion avec une base de données.
- L'interface `IDataSet` représente une suite de tables. Un objet implémentant ce type se crée à partir de la lecture d'un fichier xml avec `build` de `FlatXmlDataSetBuilder()`.
- La classe abstraite `DataBaseOperation` représente une opération effectuée avant et après chaque test.
- L'initialisation (SetUp) et le nettoyage (TearDown) pour chaque test peuvent s'effectuer avec `getTearDownOperation` et `getSetUpOperation`.

4 Analyse statique de programme Java

4.1 Findbugs

Findbugs (`http://findbugs.sourceforge.net/`) est un outil d'analyse statique de code Java basé sur la recherche de motifs (qui correspondent à des cas d'erreurs ou litigieux) dans le code. Il peut s'utiliser directement en ligne de commande ou via eclipse après installation d'un plugin.

1. Récupérer le plugin eclipse de findbugs et l'installer. Lire le manuel pour comprendre le fonctionnement du logiciel.
2. Lancer l'analyseur sur le fichier `exo4/ExampleAS1.java`et identifier les erreurs qu'il signale.

4.2 Jlint

Jlint (`http://jlint.sourceforge.net/`) est un autre outil d'analyse statique pour Java. Le nom provient de lint, analyseur statique pour C (noter qu'il existe Jslint pour Javascript). Cet outil se lance en ligne de commande sur les .class. Le lancer sur la classe `ExampleAS1.class` et identifier les erreurs qu'il signale. Le lancer également sur les classes `Triangle.class` et `StringArray.class` et analysez les résultats observés.

4.3 Type-Checker Framework

Le type checker (développé à l'université de Washington) est un outil d'analyse statique différent car il est basé sur les annotations de type ajoutées par le programmeur (il peut vérifier certains cas de référence nulle sans aucune annotation).

1. Installer le plugin eclipse (`http://types.cs.washington.edu/checker-framework/eclipse/`) et ajoutez le jar *checker-qual.jar* sur les projets testés.
2. Utiliser le type-checker sur les exemples comme décrit dans le manuel.
3. Soit la classe:

```
public class Toto{
    private @NonNull Object o;
    public Toto(int x, int y){ System.out.print(x+y);}
    public Toto(int x){ this.o.toString();}
    public Toto(int x, int y,int z){foo();}
    public void foo(){this.o.toString();}
}
```

   Déterminer à l'avance où le typechecker devrait lever un warning, puis le vérifier.

Reprendre la classe `Triangle.java` du TP sur la couverture et ajouter des annotations de type pour éviter un nom de fichier vide et des données négatives ou nulles (vous ajouterez des méthodes permettant de retourner des objects de ce type `String` non vide et `Double` positifs).

5 Encore plus de tests svp (suite de l'exo1.date)

5.1 Formaliser une spécification

À partir de la documentation python `https://docs.python.org/3.3/library/datetime.html#datetime.date`, on souhaite écrire une spécification d'une classe java *Date* qui donne une traduction de la classe python date du module datetime. La classe *Date* doit bien entendu s'abstenir d'utiliser les APIs classiques de java, telles que **Date**, **Calendar** ou autre **java.time** de java, **sauf** pour la fonction **date.today()**.

Toutes les fonctions de l'objet python *date* doivent être implémentées, à l'exception de l'attribut de classe `date.resolution`, des méthodes `date.timetuple()`, `date.strftime(format)` et de `date.__format__(format)`, ainsi que la surcharge des opérateurs (non disponible en java).

L'ensemble des fonctionnalités attendues est donc le suivant :

- Accès aux attributs de classe **min**, **max**.
- Accès aux méthodes de classe **today()**, **fromtimestamp()**, et **fromordinal()**.
- Accès aux attributs d'instance **year**, **month**, **day**.
- Accès aux méthodes d'instance **replace(year, month, day)**, **toordinal()**, **weekday()**, **isoweekday()**, **isocalendar()**, **isoformat()**, **__str__()**, **ctime()**.

Exercice définir les fonctions publiques de la classe java *Date* et donner leur spécification (en pointant vers les fonctionnalités python correspondantes).

5.2 Écriture d'un jeu de test

Exercice À partir de votre spécification, définir un jeu de test couvrant l'ensemble des fonctionnalités attendues. Les parties non-déterministes seront moquées.

5.3 Attribuer une note aux étudiants

Exercice Faîtes tourner vos tests sur les classes `Date{0-9}.java`. Pour cela, il est intéressant de créer une interface de test commune suivant votre spécification (contenant toutes les méthodes à implémentées). Pour tester les appels statiques, vous pouvez utilisé des objets de tests qui implémenteront cette interface factice. L'utilisation d'une classe abstraite et d'une " Suite class " permet d'automatiser la série de tests de toutes les classes. La syntaxe JUnit est la suivante

```
@RunWith(Suite.class)
@SuiteClasses({ ATest.class,
        BTest.class,
        CTest.class,
        DTest.class })
public class AllTests {

}
```

Exercice Attribuer la note suivante aux étudiants

$$\frac{\#\text{tests validés}}{\#\text{tests totaux}} * 20.$$