R4.02 - Qualité logicielle

Pierre Nerzic - pierre.nerzic@univ-rennes1.fr

février-mars 2024

Abstract

Il s'agit des transparents du cours mis sous une forme plus facilement imprimable et lisible.

Version du 21/03/2024 à 10:35

Table des matières

Semaine 1

Concepts généraux

C'est un cours sur la *qualité logicielle*.

Qualité logicielle ?

C'est en rapport avec ce qu'on attend d'un logiciel, un travail sans erreur, efficace, et aussi *prouvable*. C'est à dire qu'on peut s'assurer par des mécanismes fiables que le fonctionnement est entièrement conforme au cahier des charges.

Comme c'est très vaste et qu'on a très peu de temps, on va se consacrer à ce qui est appelé les « tests », c'est à dire les vérifications de la conformité aux spécifications.

Dans cet enseignement, nous verrons beaucoup d'outils et de concepts pour développer un logiciel en assurant sa qualité.

1.1. Introduction

1.1.1. Extrait du PPN BUT Info semestre 4 Ressource R4.02

« L'objectif de cette ressource est d'approfondir la production de tests, mais également d'identifier les critères de faisabilité d'un projet informatique. »

- Savoirs de référence étudiés
  - Problématique de la non-régression
  - Tests d'intégration
- Prolongements suggérés
  - Tests d'utilisabilité
  - Tests fonctionnels
  - Continuous Integration / Continuous Delivery
  - Test UI
  - Couvertures de tests

1.1.2. Plan simplifié de cet enseignement

- **Tests unitaires** : chaque méthode est vérifiée isolément
  - Outils : Java, Maven, JUnit5, AssertJ, et Mockoon (plus tard)
- **Tests d'intégration** : des groupes de classes et méthodes sont vérifiées ensemble
  - Outils : Mockito
- **Couverture des tests** : est-ce que dans chaque méthode, chaque instruction programmée a été testée ?
  - Outils : JaCoCo

- **Tests fonctionnels** : ici, il s'agira des vérifications du bon fonctionnement d'une interface utilisateur
  - Outils : Robot Framework, Selenium
- **Intégration continue** sur GitLab : tous les tests sont effectués à chaque *commit*
  - Outils : GitLab runners

1.1.3. Bibliographie

- OpenClassRooms - Testez votre code Java pour réaliser des applications de qualité par Geoffrey Arthaud, excellent cours, à connaître.
- Documentation AssertJ, complète et bien écrite, avec des exemples utiles.

1.2. Déroulement des enseignements

1.2.1. Types de cours

L'enseignement consiste en :

- 3 CM comme celui-ci : connaissances générales pour comprendre où on va
- 8 TD et TP : indifférenciés, tous consistent en travaux à faire sur machine, avec des apports théoriques fournis par le sujet et de la documentation externe à consulter

1.2.2. Évaluation

- Travaux pratiques : le principe, c'est que le travail effectué en séance devra être déposé sur Moodle. Ce travail sera noté. Chaque séance rapportera ainsi quelques points dont la somme formera la note pratique.
  - avantages : vrai contrôle continu, pas de pression, pas de gros TP noté tout à la fin
  - mise en garde : interdiction de vous transmettre un travail fait par quelqu'un d'autre.

Mon but est que vous appreniez quelque chose, pas forcément que vous ayez une bonne note. J'ai parfois l'impression que votre but est exactement l'inverse.

1.3. Tests logiciels

1.3.1. De quoi s'agit-il ?

Le concept de test logiciel est simple :

- On dispose d'une classe à tester `Classe1` ayant différentes méthodes.
- On programme une autre classe `TestsClasse1` dont les méthodes vont essayer celles de `Classe1`, et vérifier qu'elles retournent les bonnes valeurs en fonction des paramètres fournis.
- Si les résultats sont ceux attendus, les tests réussissent. Sinon, `Classe1` est mauvaise... ou `TestsClasse1`, ou les deux...

Tester, c'est douter.

1.3.2. Tests unitaires et autres

Le concept précédent est assez vague et recouvre plusieurs étendues de tests.

Figure 1: Tests unitaires et d'intégration

- Les **tests unitaires** (*unit tests*) font en sorte de ne tester qu'une seule méthode chacun, avec une seule combinaison de valeurs pour ses paramètres. Cette méthode doit retourner un résultat spécifié. Il y a donc une multitude de tests unitaires, pour chaque classe, pour chaque méthode, pour chaque combinaison de valeurs qu'on peut passer à ces méthodes.
- Les **tests d'intégration** vérifient les interactions entre deux classes ou seulement deux méthodes, que l'une a appelé l'autre dans des conditions précises.
- Les **tests fonctionnels** vérifient les fonctionnalités visibles par l'utilisateur du logiciel, par exemple l'interface graphique.

1.3.3. Tests d'intégration

Consulter une page où cette image est présente ainsi que de nombreux exemples.

1.3.4. Complexité des tests

Ces catégories de tests n'ont pas tous la même complexité.

- Les tests unitaires sont rapides à vérifier, extrêmement nombreux, relativement simples à programmer, mais par définition, ils sont indépendants entre eux et travaillent sur des données figées, éloignées de ce que les utilisateurs pourront faire.
- Les tests d'intégration sont intermédiaires, moyennement nombreux, pas très faciles à programmer, pas très rapides.
- Les tests fonctionnels sont de loin les plus lents et les plus complexes à programmer. Ils correspondent à des scénarios d'usages réalistes par les utilisateurs (*use cases*).

On ne doit faire passer les suivants que si les précédents ont réussi.

1.3.5. Pyramide des tests

En résumé :

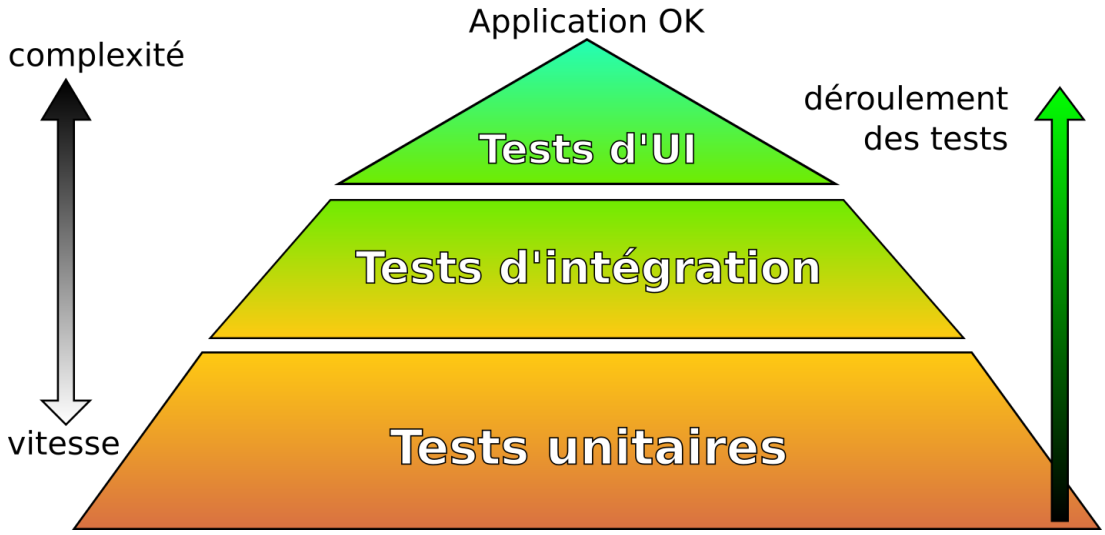

Figure 2: Pyramide des tests

1.3.6. Couverture des tests

On doit, en principe, tester tout le logiciel. C'est ce qu'on appelle la **couverture des tests** (*test coverage*).

L'idée est de vérifier si toutes les lignes de programme font l'objet d'un test unitaire, ce qui témoigne de la qualité du logiciel :

- toutes les affectations, tous les calculs, tous les appels de méthodes...
- toutes les alternatives (telle condition vraie, telle condition fausse)...
- toutes les itérations (aucune boucle, au moins une boucle)...
- les interceptions d'exceptions (pas d'exception, une exception)...

Les grandes entreprises exigent de leurs sous-traitants un taux de couverture approchant 100%.

1.3.7. Intérêt des tests

Les classes de tests accompagnent le logiciel durant toute sa vie. Le moindre changement est confronté aux tests :

- Les tests doivent continuer à réussir. Si ce n'est pas le cas, alors le changement a causé une **régression**, c'est à dire un retour à un moins bon fonctionnement, le retour d'un bug, etc.
- Il est important de découvrir des problèmes le plus tôt possible dans la vie d'un logiciel. Les coûts de correction d'une erreur augmentent très fortement en fonction du retard dans sa découverte, car de plus en plus de lignes de code sont à revoir.
- Il se peut aussi que les tests doivent évoluer, en particulier en développement Agile. Il faut maintenir les tests à chaque évolution du cahier des charges.
- On peut même se servir des tests pour guider le développement (**TDD** = *Test Driven Development*).

1.3.8. Test Driven Development

Le principe, voir wikipedia, est d'organiser le travail en cycles très courts :

1. Écriture de tests destinés à vérifier une nouvelle fonctionnalité ; en principe, ces tests échouent car la fonctionnalité n'est pas encore programmée,
2. Programmation de la fonctionnalité pour faire réussir les tests,
3. Nettoyage, simplification du code source, en préservant la réussite de tous les tests.

L'inconvénient est qu'il est préférable de confier la programmation des tests et des méthodes testées à des personnes différentes, afin de ne pas commettre les mêmes erreurs de conception (oublier de définir ce qui arrive aux valeurs `null`, par exemple).

1.4. Tests unitaires en Java

1.4.1. JUnit5

En TP, nous allons utiliser un outil appelé *JUnit5*. C'est un ensemble de classes permettant de programmer des tests unitaires, et de les exécuter.

Les tests unitaires sont chacun dans une méthode annotée par `@Test`. Cette méthode est dans une classe qui a le même *package* que la classe qu'elle teste.

Et également, cette classe est dans un autre dossier que celui des sources du projet (voir l'organisation Maven en TP).

Pour finir, la classe des tests ne contient pas de méthode `main`.

1.4.2. Exemple de classe et de test

Classe testée :

```
package fr.iutlan.ql.persons;


class Person {

    public Person(...) {
        ...
    }

    public String getInfo() {
        ...
    }
}
```

Classe de test :

```
package fr.iutlan.ql.persons;

import ...(JUnit5)...

class TestsPerson {
```

```
    @Test
    void personMust...() {
        ...getInfo()...
    }

    ...
}
```

1.4.3. Lancement des tests

L'exécution des tests est entièrement automatisée. Dans Eclipse, on a un menu `Run as... JUnit Test` et dans Maven, les tests sont faits obligatoirement avant la construction du *jar* (voir en TP).

Voici ce qu'on voit dans Eclipse quand les tests ont réussi :

1.4.4. Analyse d'un test échoué

Par contre, en cas d'échec, les résultats sont assez horribles :

Il faut trouver `expected: <chose> but was: <chauze>`, puis aller voir ce qu'il en est à la fois dans le test lui-même et dans la méthode qui est testée.

Cela nous montre pourquoi les tests unitaires doivent être les plus élémentaires possible : pour isoler les problèmes et n'avoir aucune difficulté à comprendre ce qui n'a pas marché et quoi corriger.

Cela passe par une bonne présentation des tests : leur nom et la structuration des opérations de test à l'intérieur. En effet, ces tests ayant la durée de vie du logiciel et étant très nombreux, il n'est pas question de devoir se plonger dans du code compliqué six mois après.

1.4.5. Normes de présentation des tests unitaires

Les tests sont des méthodes dans une classe. La classe elle-même doit être nommée pour faire référence à celle qu'elle teste. Et chaque méthode doit porter un nom, en CamelCase, pouvant être très long, explicitant totalement ce qu'elle fait, de manière à ce qu'une seule lecture suffise.

Exemples :

- `setLastnameWithNullMustThrowNullPointerException`
- `getFirstnameOnUninitializedInstanceMustReturnNull`
- `setAddressEmptyMustThrowIllegalArgumentException`
- `getCarsCountOnEmptyCarsListMustReturnZero`

Il peut vraiment y avoir des centaines de méthodes comme ça, donc interdiction absolue de noms comme `test1`, `testOk`, etc.

Ensuite, le corps de chaque méthode de test doit être organisé de la manière suivante :

- Une partie appelée **Arrange** ou *Given* qui prépare des données en vue de mener un test, par exemple créer une instance de la classe testée et la paramétrer avec des méthodes testées par ailleurs elles-aussi.
- Une partie **Act** ou *When* qui effectue le test unitaire.
- Une partie **Assert** ou *Then* qui compare le résultat à ce qui est attendu.
- Ces trois parties sont à séparer par une ligne vide.

Cette norme s'appelle **AAA** et nous l'appliquerons en TP.

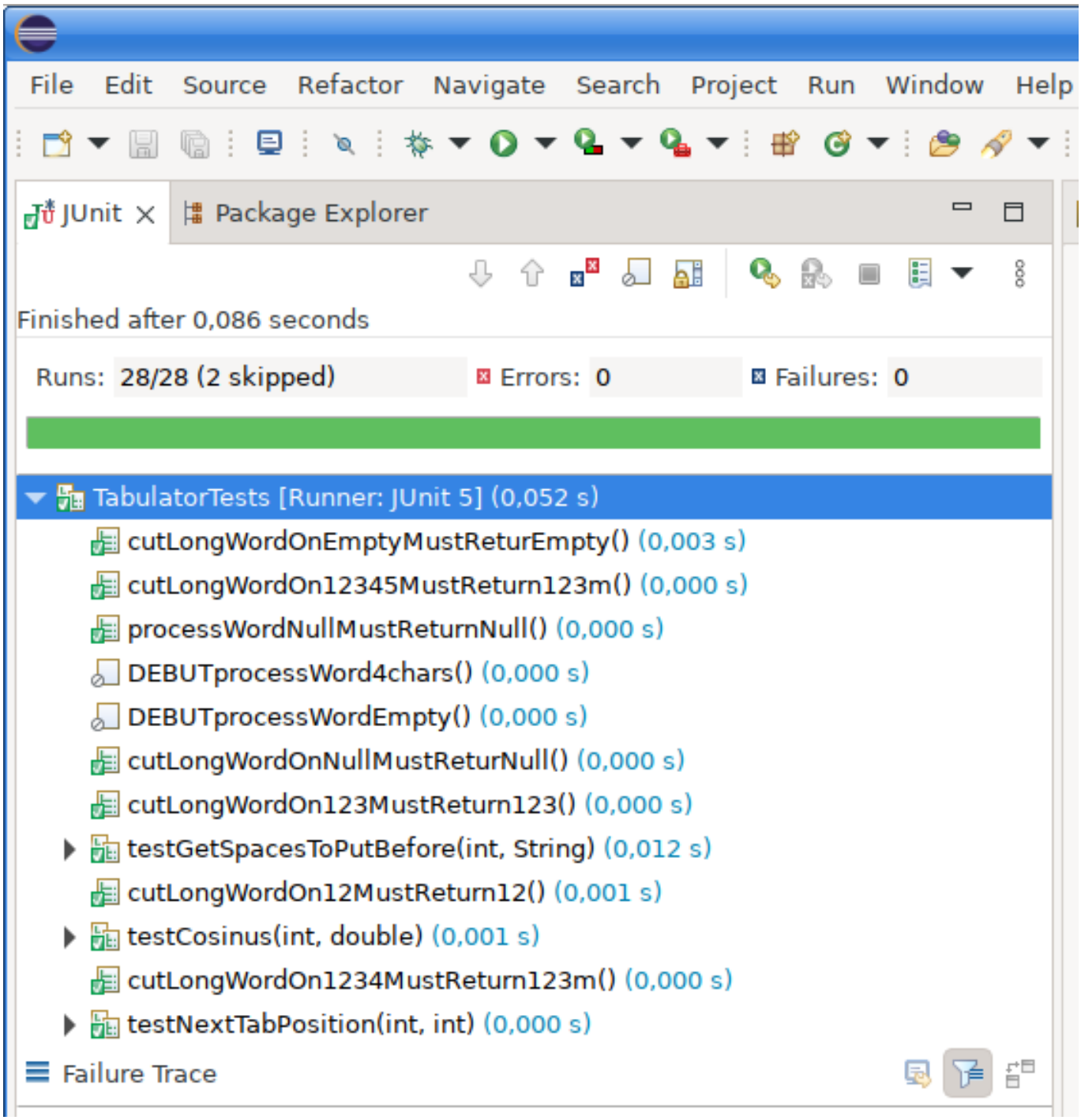

Figure 3: Succès des tests

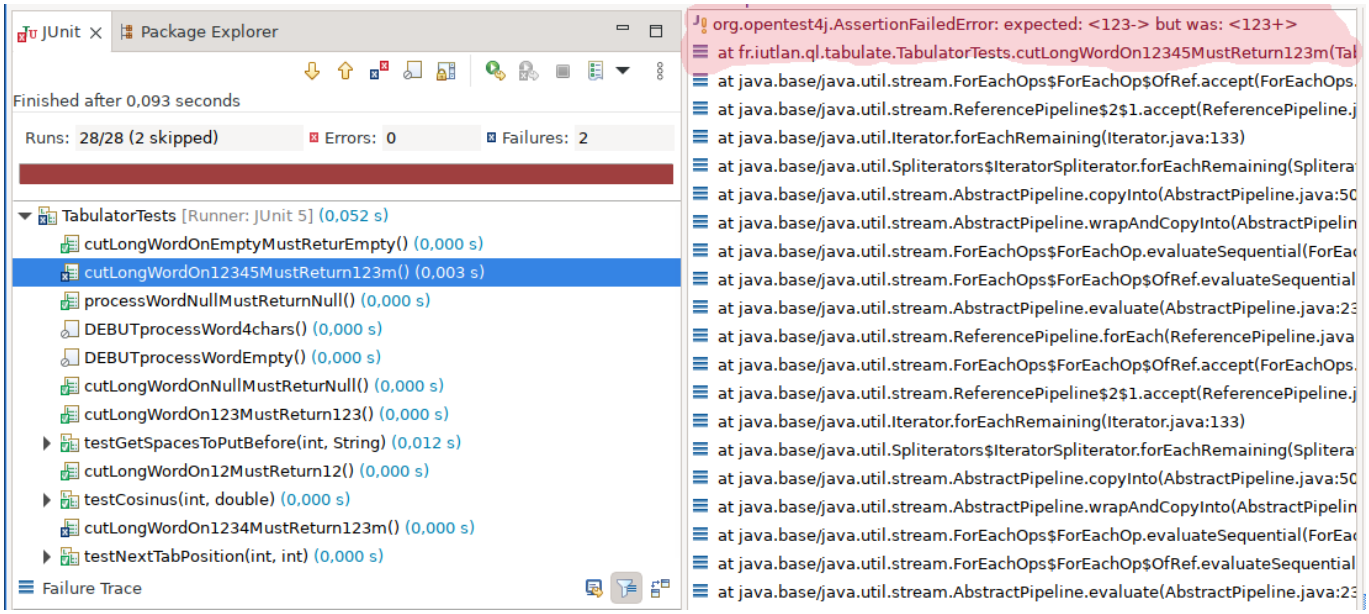

Figure 4: Échec d'un test

1.4.6. Exemple de méthode JUnit5 en AAA

```java
@Test
void personToStringMustReturnFirstNameSpaceLastName() {
    // ARRANGE
    Person pers = new Person();
    pers.setLastName("Nerzic");
    pers.setFirstName("Pierre");

    // ACT
    String result = pers.toString();

    // ASSERT
    assertEquals("Pierre Nerzic", result);
}
```

1.4.7. Intérêt de ces normes

L'intérêt de ces normes est de permettre la compréhension des tests unitaires par n'importe qui dans l'équipe.

Le pire serait d'avoir des méthodes de test bizarres, voire boguées, difficile d'être certain de ce qu'elles testent.

La contrainte, c'est de perdre un peu de temps à faire une jolie présentation. En même temps, c'est un signe de qualité de votre travail.

Rq: dans quelques cas, le patron *AAA* doit être un peu modifié, dans le cas des exceptions ou de mises en place en cascade.

1.4.8. Assertions JUnit5

Les vérifications sont exprimées sous forme d'**assertions**. C'est à dire des propositions qui doivent être vraies, par exemple telle méthode retourne telle valeur quand on l'appelle avec tels paramètres.

JUnit5 n'a pas beaucoup d'assertions à proposer :

- `assertEquals(attendu, résultat)` : le résultat doit être égal à attendu
- `assertNotEquals(attendu, résultat)` : le résultat ne doit pas être égal à attendu
- `assertNull(résultat)` et `assertNotNull(résultat)` : le résultat est comparé à `null`
- `assertTrue(résultat)` et `assertFalse(résultat)`

1.4.9. Assertions pour les exceptions

Dans le cas des exceptions, il faut faire ceci :

```
@Test
void addCarNullMustThrowNullPointerException() {
    // ARRANGE
    Person pers = new Person();

    // ASSERT
    assertThrows(NullPointerException.class, () -> {

        // ACT
        pers.addCar(null);
    });
}
```

Le test réussit si la partie *ACT* déclenche l'exception attendue.

1.4.10. Initialisation des tests

Il est fréquent d'avoir besoin de la même initialisation pour chaque test, par exemple, la même création et paramétrage d'instance.

Au lieu de répéter les mêmes instructions au début de chaque test, il y a la possibilité de définir :

- Une méthode d'initialisation de *chaque* test ; elle est annotée avec `@BeforeEach` au lieu de `@Test`,
- Une méthode d'initialisation de *tous* les tests de la classe, annotée `@BeforeAll`, **attention** la méthode doit être `static`.

Attention à la différence : la première est ré-exécutée pour chaque test, la seconde n'est exécutée qu'une seule fois en tout.

1.4.11. Exemple avec `@BeforeEach`

```
class TestPerson {
```

```
    Person persPN;

    @BeforeEach
    void initPersPN() {
        persPN = new Person();
        pers.setLastName("Nerzic");
        pers.setFirstName("Pierre");
    }

    @Test
    void personMust...() {
        ...
    }
```

1.4.12. Exemple avec `@BeforeAll`

```
class TestPerson {

    static Person persPN;

    @BeforeAll
    static void initPersPN() {
        persPN = new Person();
        pers.setLastName("Nerzic");
        pers.setFirstName("Pierre");
    }

    @Test
    void personMust...() {
        ...
    }
```

1.4.13. Différences entre ces deux exemples

Dans le second, `@BeforeAll`, la méthode `initPersPN` n'est appelée qu'une seule fois en tout. Donc l'instance de `Person` qui est employée par les tests est toujours la même. Si l'un des tests modifie cette instance, la modification sera transmise à tous les tests ultérieurs.

Dans le premier, `@BeforeEach` la méthode `initPersPN` est appelée autant de fois qu'il y a des tests, et avant chacun. Donc l'instance de `Person` est différente à chaque fois, mais initialisée de la même manière.

À quoi sert donc `@BeforeAll` ? À initialiser quelque chose qui doit persister, mais qui n'est pas affecté en soi par les tests. Par exemple à ouvrir une connexion avec une base de données.

1.4.14. Autres annotations

Pour libérer des ressources réservées par `@BeforeAll` et `@BeforeEach`, il y a les deux annotations inverses, `@AfterAll` et `@AfterEach`. Dans un environnement de tests, il est important de maîtriser les ressources utilisées. Tout ce qui a été alloué doit être libéré en miroir.

L'annotation `@Timeout(n)` permet de vérifier qu'une méthode ne dure pas plus de n secondes.

Si on veut désactiver temporairement un test, par exemple, pour qu'il ne gène pas le temps de le mettre au point, on peut lui ajouter l'annotation `@Disabled`.

1.5. Tests paramétrés

1.5.1. Mêmes tests avec des valeurs différentes

Soit une méthode à tester, qui demande des paramètres, par exemple numériques. Il est utile de vérifier le comportement face à la plus petite valeur possible, la plus grande possible, la valeur moyenne, des valeurs induisant des erreurs, etc.

Cela fait plusieurs tests très similaires, seules les valeurs fournies à la méthode changent. Faut-il créer autant de variantes du même test ?

Non : il est plus simple de définir un seul test, mais qui prend un paramètre, et fournir les valeurs qu'il devra passer à la méthode testée.

1.5.2. Mise en œuvre de tests paramétrés

Voici comment faire en *JUnit5* : on remplace `@Test` par `@ParameterizedTest` et on lui ajoute une autre annotation `@ValueSource` pour spécifier les valeurs à employer :

```
@ParameterizedTest
@ValueSource(ints = { 18, 19, 22, 34 })
void checkAgeIsAdult(int age) {
    // ARRANGE
    Person pers = new Person();
    pers.setAge(age);
    // ACT
    boolean result = pers.isAdult();
    // ASSERT
    assertTrue(result);
}
```

1.5.3. Types des valeurs

L'annotation `@ValueSource(TYPE = { VALEURS })` sert à spécifier les données à passer à la fonction de test.

Le *TYPE* est, entre autres, `ints`, `longs`, `floats`, `doubles`, `strings`, correspondant aux valeurs.

Pour fournir plusieurs paramètres ou des objets, c'est un peu plus compliqué.

1.5.4. Fournisseur de valeurs

Pour plusieurs paramètres ou de types objets, il est pratique d'utiliser un *générateur de valeurs*. C'est une méthode qui retourne une liste de n-uplets constituant les valeurs à fournir à la méthode de test.

On a deux choses. D'abord la méthode de test :

```
@ParameterizedTest
@MethodSource
void testFtOnAReturnsB(int paramA, string paramB) {
    ...
}
```

Il y a cette nouvelle annotation `@MethodSource` qui indique que les valeurs sont générées par une méthode portant exactement le même nom que le test, mais *statique* et surchargée différent.

Cette seconde fonction doit retourner une liste de n-uplets contenant les valeurs. Un n-uplet en JUnit5 est une instance de la classe `Arguments`. Cette classe possède un constructeur de type *fabrique* avec la méthode `of`.

```
// méthode qui génère les arguments pour testFtOnAReturnsB
private static Stream<Arguments> testFtOnAReturnsB() {
    return Stream.of(
        Arguments.of( 10, "ten"),
        Arguments.of(-23, "minus twenty three"),
        Arguments.of(  7, "seven")
    );
}
```

La classe `Stream` est une sorte de liste améliorée. On la remplit avec des instances de `Arguments` destinées à la méthode de test.

1.6. AssertJ et Truth

1.6.1. Insuffisances de JUnit

Avec ses trop peu nombreuses assertions, JUnit5 rend difficile des vérifications comme : « *tous les éléments de la liste doivent être des entiers positifs* », ou « *au moins l'un des éléments de la collection doit être différent de* `null` ».

Comment feriez-vous sans écrire un algorithme potentiellement faux ? Le comble serait de faire une erreur algorithmique dans une assertion.

C'est pour cela que des librairies tierces ont été proposées, *Hamcrest*, *AssertJ* et *Truth* notamment. *Hamcrest* est plutôt difficile à utiliser à cause de la syntaxe des *matchers* utilisant la généricité, très complexe en Java.

Et pour choisir entre *AssertJ* et *Truth*, voir cette discussion. Elles se ressemblent énormément.

1.6.2. Présentation rapide de Truth et AssertJ

Le principe, c'est d'écrire des sortes de phrases, ressemblant à de l'anglais, exprimant ce qu'il faut vérifier.

Quelques exemples *AssertJ*, presque identiques en *Truth* :

```
assertThat(result).isNotNull();
assertThat(result).isEqualTo(555);
assertThat(result).isGreaterThan(0);

assertThat(message).contains("jour");
assertThat(message).startsWith("bon");
assertThat(message).hasSizeBetween(5, 10);
assertThat(message).matches("b[a-z]*r");
```

Il y a une multitude de possibilités qui seront détaillées en TP.

1.6.3. Comparaisons entre AssertJ et Truth

- Documentation : la documentation de AssertJ est bien meilleure (explications, exemples) que celle de Truth (uniquement une FAQ et la JavaDoc).
- Programmation fluide (*fluent*) en *AssertJ*, impossible en *Truth*[1]

```
assertThat(colors)
    .isNotEmpty()
    .hasSize(3)
    .doesNotHaveDuplicates()
    .contains("white", "red", "blue")
    .startsWith("white")
    .endsWith("red")
    .containsSequence("white", "blue")
    .doesNotContain("black", "orange", null);
```

Certains tests pas possibles directement avec Truth :

- il n'y a aucun `null` dans la liste : **Truth** `assertThat(liste).doesNotContain(null);`
  AssertJ `assertThat(liste).doesNotContainNull();`
- il y a au moins un `null` dans la liste : **Truth** `assertThat(liste).contains(null);`
  AssertJ `assertThat(liste).containsNull();`
- il n'y a pas que des `null` dans la liste (pas avec Truth ⇒ JUnit5+Java) :
  J `assertTrue(liste.stream().anyMatch(n -> n != null));`
  A `assertThat(liste).anyMatch(n -> n != null);`
- il n'y a que des `null` dans la liste (pas avec Truth ⇒ JUnit5+Java) :
  J `assertTrue(liste.stream().allMatch(n -> n == null));`
  A `assertThat(liste).allMatch(n -> n == null);`

Mode d'emploi complet en TP...

1.6.4. C'est tout pour aujourd'hui

Cette présentation est finie. Rendez-vous en TD et TP pour mettre cela en pratique.

[1] car les méthodes retournent `void`, malgré les prétentions de la doc.

Semaine 2

XML

Dans ce cours :

- Norme XML
- Validation par un schéma
- Interrogation avec XPath

Pourquoi ?

De nombreux outils liés à la qualité logicielle utilisent XML, et c'est un format de stockage et d'échange de données qui propose des concepts à connaître : validation, interrogation, transformation.

2.1. Présentation rapide de la norme XML

2.1.1. XML ?

XML = Extensible Markup Language

Un fichier XML représente des informations structurées :

```xml
<?xml version="1.0" encoding="utf-8"?>
<!-- itinéraire fictif -->
<itineraire>
  <etape distance="0km">départ</etape>
  <etape distance="13km">tourner à droite</etape>
  <etape distance="22km">arrivée</etape>
</itineraire>
```

Cet exemple modélise un itinéraire composé d'étapes.

XML permet de choisir la représentation des données sans aucune contrainte. On choisit les balises et les attributs comme on le souhaite.

2.1.2. Arborescence d'éléments

Un document XML est composé de plusieurs parties :

- Entête de document précisant la version et l'encodage,
- Des règles optionnelles pour vérifier si le document est valide,
- Un arbre d'*éléments* basé sur un élément appelé *racine*
  - Un *élément* possède un *nom*, des *attributs* et un *contenu*

- Le contenu d'un élément peut être :
  - rien : élément vide noté `<nom attributs.../>`
  - du texte
  - d'autres éléments (les éléments enfants).
- Un élément non vide est délimité par une *balise ouvrante* et une *balise fermante*.
  - une balise ouvrante est notée `<nom attributs...>`
  - une balise fermante est notée `</nom>`

2.1.3. Exemple complet

Voici un document XML représentant une personne :

```
<?xml version="1.0" encoding="utf-8"?>
<personne nom="Nerzic" prenom="Pierre">
  <profession>
    enseignant
    <lieu>
      IUT
      <codepostal/>
      <ville>Lannion</ville>
    </lieu>
    informatique
  </profession>
</personne>
```

Les textes peuvent être délimités par des balises, ou non.

2.1.4. Représentation graphique

2.1.5. Explications

Le document XML représente un arbre composé de plusieurs types de nœuds (*node* en anglais) :

nœuds éléments ils sont associés aux balises `<element>`. Ce sont des nœuds qui peuvent avoir des enfants en dessous.

nœuds `#text` ils représentent le texte situé entre deux balises. Les nœuds texte sont des feuilles dans l'arbre.

nœuds attributs ils contiennent les attributs des éléments

Notez que différents textes peuvent être entrelacés avec des éléments. Voir le cas de `<lieu>` dans le contenu de `<profession>`. Il est possible de le faire, mais ce n'est pas forcément souhaitable.

D'autres types de nœuds existent mais ne seront pas présentés.

2.1.6. Vocabulaire

Soit cet arbre XML :

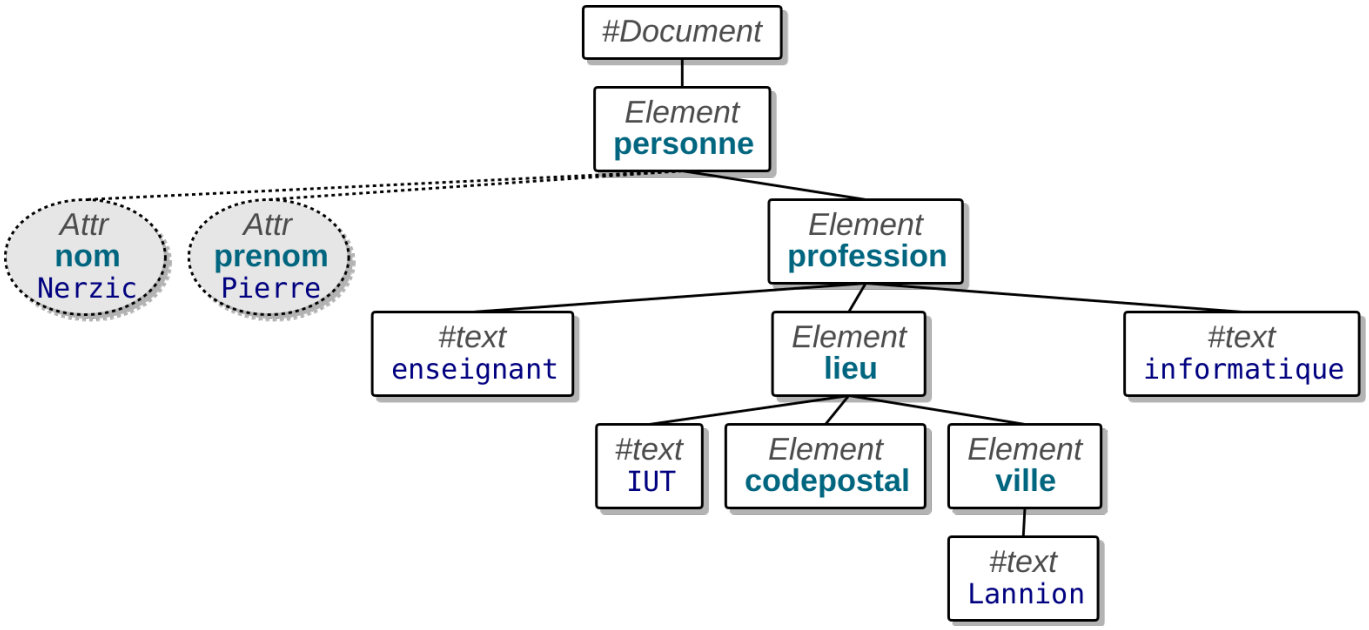

Figure 5: Arbre XML

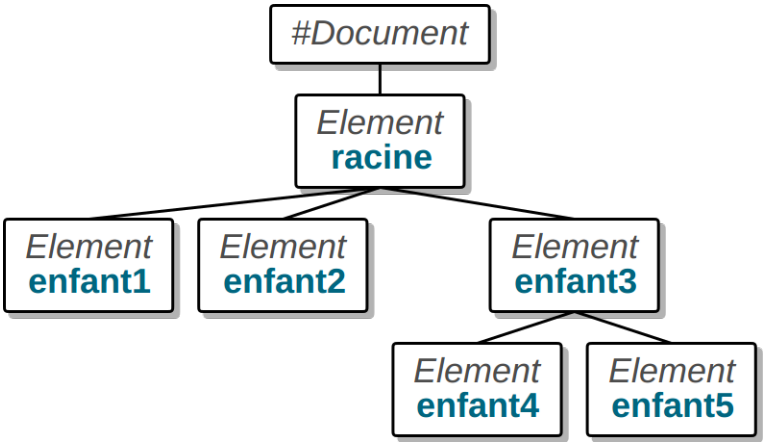

Figure 6: Nœuds parent, enfant et cousins

2.1.7. Vocabulaire (suite)

Voici comment on désigne les différents nœuds les uns par rapport aux autres :

- `<racine>` est le nœud **parent** du nœud **enfant** (*child*) `<enfant3>`, lui-même parent de `<enfant4>` et `<enfant5>`,
- `<racine>`, `<enfant3>` sont des nœuds **ancêtres** (*ancestors*) de `<enfant4>` et `<enfant5>`,
- `<enfant4>` et `<enfant5>` sont des **descendants** (*descendants*) de `<racine>` et `<enfant3>`,
- `<enfant1>` est un nœud **frère** (*sibling* = fratrie) de `<enfant2>` et réciproquement.

2.1.8. Attributs

Les attributs caractérisent un élément. Ce sont des couples `nom="valeur"` ou `nom='valeur'`. Ils sont placés dans la balise ouvrante.

```
<?xml version="1.0" encoding="utf-8"?>
<meuble id="765" type='table'>
  <prix monnaie='€'>74,99</prix>
  <dimensions unites="cm" longueur="120" largeur="80"/>
  <description langue='fr'>Belle petite table</description>
</meuble>
```

Remarques :

- Il n'y a pas d'ordre entre les attributs d'un élément,
- Un attribut ne peut être présent qu'une fois.

2.1.9. Texte

Les textes font partie du contenu des éléments et sont vus comme des nœuds enfants.

```
<?xml version="1.0" encoding="utf-8"?>
<racine>
  texte1
  <enfant>texte2</enfant>
  texte3
</racine>
```

Il faut bien comprendre que tous les fragments de texte situés entre les balises, y compris les espaces et les retours à la ligne sont considérés comme faisant partie du même texte. Dans un programme, il faut penser à nettoyer les textes extraits.

2.1.10. Arbre correspondant

Voici l'arbre correspondant à l'exemple précédent. Notez les retours à la ligne et espaces présents dans les textes sauf `texte2`.

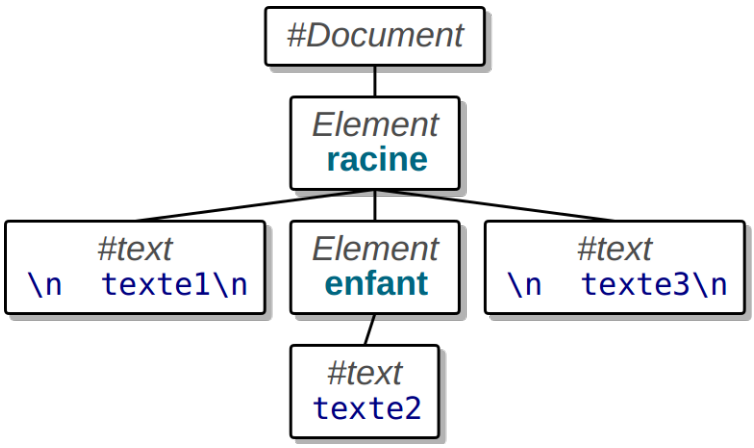

Figure 7: Arbre XML

2.1.11. Noms des éléments

Les noms des éléments peuvent employer de nombreux caractères Unicode (correspondant au codage déclaré dans le prologue) mais pas les signes de ponctuation.

Le caractère « : » permet de séparer le nom en deux parties, préfixe et *nom local*. Le tout s'appelle *nom qualifié*. Par exemple `iut:departement` est un nom qualifié préfixé par `iut`.

$$\textit{nom qualifié} = \textit{préfixe} : \textit{nom local}$$

Le préfixe permet de définir un *espace de nommage* (*namespace*).

2.1.12. Espaces de nommage

Un espace de nommage définit une famille de noms afin d'éviter les confusions entre des éléments qui auraient le même nom mais pas le même sens. Cela arrive quand le document XML modélise les informations de plusieurs domaines.

Voici un exemple dans le domaine de la vente de meubles. Le document modélise une table (avec 4 pieds) et aussi un tableau HTML pour afficher ses dimensions. On voit la confusion.

```
<meuble id="765">
  <table prix="74,99€">acajou</table>
  <table border="1">
    <tr><th>longueur</th><th>largeur</th></tr>
    <tr><td>120cm</td><td>80cm</td></tr>
  </table>
</meuble>
```

2.1.13. Définition d'un espace de nommage

On doit choisir un URI (désignation d'une ressource internet), un URL ou un URN, pour identifier l'espace de nommage.

Un URL a cette syntaxe : `schéma:[//[user:passwd@]hôte[:port]][/]chemin[?requête][#fragment]`, par exemple `http://en.wikipedia.org/wiki/Uniform_Resource_Identifier#Syntax`.

Les URN sont au format `urn:NID:NSS`, ex: `urn:iutlan:xmlsem1`

Ensuite on rajoute un attribut spécial à la racine du document :

```
<?xml version="1.0" encoding="utf-8"?>
<racine xmlns:PREFIXE="URI">
    <PREFIXE:balise>...</PREFIXE:balise>
</racine>
```

2.1.14. Exemple revu

Voici l'exemple précédent, avec deux *namespaces*, un URN pour les meubles et un URL pour HTML :

```
<?xml version="1.0" encoding="utf-8"?>
<meuble:meuble id="765"
        xmlns:meuble="urn:iutlan:meubles"
        xmlns:html="http://www.w3.org">
  <meuble:table prix="74,99€">acajou</meuble:table>
  <html:table border="1">
    <html:tr><html:th>longueur</html:th>...</html:tr>
    <html:tr><html:td>120cm</html:td>...</html:tr>
  </html:table>
</meuble:meuble>
```

Notez le préfixe également appliqué à la racine du document.

2.1.15. Namespace par défaut

Lorsqu'un élément définit un attribut `xmlns="URI"`, alors lui-même et ses descendants sont placés dans ce namespace, sans préfixe.

```
<?xml version="1.0" encoding="utf-8"?>
<book xmlns="http://docbook.org/ns/docbook">
   <title>Livre très simple</title>
</book>
```

Autre exemple :

```
<meuble id="765" xmlns="urn:iutlan:meubles">
  <table prix="74,99€">acajou</table>
  <table border="1" xmlns="http://www.w3.org">
    <tr><th>longueur</th>...</tr>
    <tr><td>120cm</td>...</tr>
  </table>
</meuble>
```

2.1.16. Namespace par défaut, attributs et valeurs

Définir un *namespace* par défaut associe seulement les éléments à ce *namespace*, mais pas leurs attributs.

Les attributs qui n'ont pas de préfixe n'ont pas de *namespace*. L'interprétation des attributs dépend de l'élément dans lequel ils se trouvent. Voir cette réponse sur StackOverflow.

Cependant, pour clarifier la situation, on rajoute de préférence un préfixe devant les attributs. Par exemple, `android:layout_width`. Avec le préfixe, il n'y a plus d'ambiguïté possible : l'attribut appartient au *namespace* concerné.

On retrouve le même problème avec les valeurs de certains attributs, voir les schémas XML plus loin.

2.2. Validité d'un document

2.2.1. Introduction

Il y a deux niveaux de correction pour un document XML :

- Un document XML **bien formé** (*well formed*) respecte les règles syntaxiques d'écriture XML : écriture des balises, imbrication des éléments, entités, etc. C'est la plus facile des vérifications.
- Un document **valide** respecte des règles supplémentaires sur les noms, les attributs et l'organisation des éléments.

La validation est cruciale pour une entreprise, p.ex. une banque, qui gère des transactions représentées en XML. S'il y a des erreurs dans les documents, cela peut compromettre l'entreprise. Il vaut mieux être capable de refuser un document invalide plutôt qu'essayer de le traiter et pâtir des erreurs qu'il contient.

2.2.2. Processus de validation

D'abord, il faut disposer d'un fichier contenant des règles. Il existe plusieurs langages pour faire cela : DTD, Schemas XML, RelaxNG et Schematron. Ces langages modélisent des règles de validité plus ou moins précises et d'une manière plus ou moins lisible.

On ne verra pas les DTD, des antiquités, mais les schémas XML.

Chaque document XML est comparé à ce fichier de règles, à l'aide d'un outil de validation : `xmlstarlet`, `xmllint`...

La validation indique :

- soit le document est valide, conforme aux règles,
- soit il contient des erreurs comme : tel attribut de tel élément contient une valeur interdite par telle contrainte, il manque tel sous-élément ou attribut dans tel élément, etc.

2.2.3. Schémas XML

Les Schémas XML sont une norme W3C pour spécifier le contenu d'un document XML.

Pour se faire une idée, voici un exemple de schéma :

```xml
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="reference" type="ElemReference" />
  <xs:complexType name="ElemReference">
    <xs:sequence>
      <xs:element name="titre" type="xs:string" />
      <xs:element name="auteur" type="xs:string" />
      <xs:element name="ISBN" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

2.2.4. Validation d'un document XML par un schéma

Quand on possède un schéma (un fichier `.xsd`) et un document XML, on peut valider le document par une de ces commandes Unix :

- `xmllint --schema schema.xsd --noout document.xml`
- `xmlstarlet val --xsd schema.xsd -e document.xml`

Soit ça passe, soit il y a une erreur de validation : le document ne respecte pas le schéma.

C'est un peu le même principe qu'un test unitaire, mais cela concerne la structure et le contenu d'un document XML.

2.2.5. Principes généraux des Schémas XML

Un schéma permet de définir des éléments, leurs attributs et leurs contenus, avec une notion de typage forte.

Avec un schéma, il faut définir des **types de données** :

- la nature des données : chaîne, nombre, date, etc.
- les contraintes qui portent dessus : domaine de définition, valeurs possibles, expression régulière, etc.

Et avec ces types, on définit les éléments :

- noms et types des attributs
- sous-éléments possibles avec leurs répétitions, les alternatives...

2.2.6. Structure générale d'un schéma

Un schéma est contenu dans un arbre XML de racine `<schema>`. Le contenu du schéma définit les éléments qu'on peut trouver dans le document. Voici un squelette de schéma :

```xml
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="itineraire" type="ElemItineraire" />
  ... définition du type ElemItineraire ...
</xs:schema>
```

Ce schéma valide le document partiel suivant :

```
<?xml version="1.0" encoding="utf-8"?>
<itineraire>
    ...
</itineraire>
```

2.2.7. Définition d'éléments

Un élément <*nom*>*contenu*</*nom*> du document est défini par un élément `<xs:element name="`*nom*`" type="`*TypeContenu*`">` dans le schéma.

Dans l'exemple suivant, le type est `xs:string`, un texte quelconque, donc l'élément peut/doit contenir du texte :

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="message" type="xs:string"/>
</xs:schema>
```

Ce schéma valide le document suivant :

```
<?xml version="1.0" encoding="utf-8"?>
<message>Tout va bien !</message>
```

2.2.8. Définition de types de données

L'exemple précédent indique que l'élément `<message>` doit avoir un contenu de type `xs:string`, c'est à dire du texte. Ce type est un « type simple ». Il y a de nombreux types simples prédéfinis, dont :

- chaîne :
  - `xs:string` est le type le plus général
  - `xs:token` vérifie que c'est une chaîne nettoyée des sauts de lignes et espaces d'indentation
- date et heure :
  - `xs:date` correspond à une chaîne au format `AAAA-MM-JJ`
  - `xs:time` correspond à `HH:MM:SS.s`
  - `xs:datetime` valide `AAAA-MM-JJTHH:MM:SS`, on doit mettre un `T` entre la date et l'heure.

2.2.9. Types de données (suite)

- nombres :
  - `xs:float`, `xs:decimal` valident des nombres réels
  - `xs:integer` valide des entiers
  - il y a de nombreuses variantes comme `xs:nonNegativeInteger`, `xs:positiveInteger`...
- autres :
  - `xs:ID` pour une chaîne identifiante, `xs:IDREF` pour une référence à une telle chaîne
  - `xs:boolean` permet de n'accepter que `true`, `false`, `1` et `0` comme valeurs dans le document.
  - `xs:base64Binary` et `xs:hexBinary` pour des données binaires.
  - `xs:anyURI` pour valider des URI (URL ou URN).

2.2.10. Restrictions sur les types

Lorsque les types ne sont pas suffisamment contraints et risquent de laisser passer des données fausses, on peut rajouter des contraintes. Elles sont appelées *facettes* (*facets*).

Dans ce cas, on doit définir un type `simpleType` et lui ajouter des restrictions, comme dans cet exemple :

```
<xs:element name="temperature" type="TypeTemperature" />

<xs:simpleType name="TypeTemperature">
  <xs:restriction base="xs:decimal">
    <xs:minInclusive value="-30"/>
    <xs:maxInclusive value="+40.0"/>
  </xs:restriction>
</xs:simpleType>
```

2.2.11. Définition de restrictions

La structure d'une restriction est :

```
<xs:restriction base="type de base">
    <xs:CONTRAINTE value="PARAMETRE"/>
    ...
</xs:restriction>
```

Par exemple, un type « numéro de sécurité sociale » :

```
<xs:simpleType name="TypeNumeroSecu">
  <xs:restriction base="xs:string">
    <xs:whiteSpace value="collapse"/>
    <xs:pattern value="[12][0-9]{12}( [0-9]{2})?"/>
  </xs:restriction>
</xs:simpleType>
```

Les contraintes qu'on peut mettre dépendent du type de données.

2.2.12. Restriction sur `string`

Ces restrictions (*facettes*) s'appliquent au type `string` :

- longueur : `length`, `maxLength`, `minLength`. Ces contraintes vérifient que la valeur possède la bonne longueur.
- `whiteSpace` indique ce qu'on doit faire avec les caractères espaces, tabulations et retours à la ligne éventuellement présents dans les données, pour vérifier les facettes :
  - `value="preserve"` : on les garde tels quels
  - `value="replace"` : on les remplace par des espaces
  - `value="collapse"` : on les supprime tous (à utiliser avec un motif).

2.2.13. Restrictions sur `string` (suite)

- expression régulière étendue (egrep) avec `pattern` :

```
<xs:simpleType name="TypeTemperature">
  <xs:restriction base="xs:string">
    <xs:pattern value="[-+]?[1-9][0-9]?°C"/>
    <xs:whiteSpace value="collapse"/>
  </xs:restriction>
</xs:simpleType>
```

- liste des valeurs possibles avec `enumeration` :

```
<xs:simpleType name="TypeFreinsVélo">
  <xs:restriction base="xs:string">
    <xs:enumeration value="disque"/>
    <xs:enumeration value="patins"/>
    <xs:enumeration value="rétropédalage"/>
  </xs:restriction>
</xs:simpleType>
```

2.2.14. Restrictions sur les dates et nombres

Les dates et nombres possèdent quelques contraintes sur la valeur exprimée :

- bornes inférieure et supérieure :
  - `minExclusive` et `minInclusive`
  - `maxExclusive` et `maxInclusive`
- nombre de chiffres :
  - `totalDigits` : vérifie le nombre de chiffres total (partie entière et fractionnaire, sans compter le point décimal)
  - `fractionDigits` : vérifie le nombre de chiffres dans la partie fractionnaire.
- Les facettes `pattern` et `enumeration` sont utilisables aussi.

2.2.15. Types à alternatives

Comment valider une donnée qui pourrait être de plusieurs types possibles, par exemple, valider les deux premiers éléments et refuser le troisième :

```
<couleur>Chartreuse</couleur>
<couleur>#7FFF00</couleur>
<couleur>02 96 46 93 00</couleur>
```

Si on déclare l'élément `<couleur>` comme contenant n'importe quelle valeur chaîne, comme ceci :

```
<xs:element name="couleur" type="xs:string"/>
```

on ne pourra rien vérifier.

2.2.16. Types à alternatives (suite)

Alors on crée un « type à alternatives » qui est équivalent à plusieurs possibilités. Attention, ce n'est pas comme déclarer une énumération de *valeurs possibles*. Ici, on parle de *types possibles*.

Pour exprimer qu'un type peut correspondre à plusieurs autres types, il faut le définir en tant que `<union>` et mettre les différents types possibles dans l'attribut `memberTypes` :

```
<xs:simpleType name="TYPE_ALTERNATIF">
  <xs:union memberTypes="TYPE1 TYPE2 ..."/>
</xs:simpleType>
```

Les types possibles sont séparés par un espace.

2.2.17. Exemple de type à alternatives

Voici un exemple pour les couleurs :

```
<xs:simpleType name="TypeCouleurs">
  <xs:union memberTypes="TypeCouleursNom TypeCouleursHex"/>
</xs:simpleType>

<xs:simpleType name="TypeCouleursNom">
  <xs:restriction base="xs:string">
    <xs:pattern value="([A-Z][a-z]+)+"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="TypeCouleursHex">
  <xs:restriction base="xs:string">
    <xs:pattern value="#[0-9A-F]{6}"/>
  </xs:restriction>
</xs:simpleType>
```

2.2.18. Contenu d'éléments

On revient maintenant sur les éléments. Nous avons vu comment définir un élément contenant un texte, un nombre, etc. :

```
<xs:element name="NOM" type="TYPE"/>
```

Ça définit une balise `<NOM>` pouvant contenir des données du type indiqué par `TYPE` :

```
<?xml version="1.0" encoding="utf-8"?>
<NOM>données correspondant à TYPE</NOM>
```

Comment définir un élément dont le contenu peut être d'autres éléments, ainsi que des attributs ? En fait, c'est la même chose que les `simpleType`, sauf que le type est « complexe ». Un type complexe peut contenir des sous-éléments et des attributs.

2.2.19. Exemple de type complexe

Pour modéliser un élément `<personne>` ayant deux éléments enfants `<prénom>` et `<nom>`, il suffit d'écrire ceci :

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="personne" type="ElemPersonne"/>
  <xs:complexType name="ElemPersonne">
    <xs:sequence>
      <xs:element name="prénom" type="xs:string"/>
      <xs:element name="nom" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

La structure `<xs:sequence>` contient une liste d'éléments qui doivent se trouver dans le document à valider, dans l'ordre.

2.2.20. Contenu d'un type complexe

Un `<xs:complexType>` peut contenir trois sortes d'enfants :

```
<xs:complexType name="ElemPersonne">
  <xs:sequence> ou <xs:choice> ou <xs:all>...
</xs:complexType>
```

Les enfants peuvent être :

- `<xs:sequence>`éléments...`</xs:sequence>` : ces éléments doivent arriver dans cet ordre
- `<xs:choice>`éléments...`</xs:choice>` : le document à valider doit contenir l'un des éléments
- `<xs:all>`éléments...`</xs:all>` : le document à valider doit contenir ces éléments une et une seule fois chacun, et dans n'importe quel ordre.

2.2.21. Exemple de choix

Pour représenter une limite temporelle, par exemple la date de fin d'une garantie, soit on mettra un élément `<date_fin>` soit un élément `<durée>` :

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="limite" type="ElemLimiteTemps"/>
  <xs:complexType name="ElemLimiteTemps">
    <xs:choice>
      <xs:element name="date_fin" type="xs:date"/>
      <xs:element name="durée" type="xs:positiveInteger"/>
    </xs:choice>
  </xs:complexType>
</xs:schema>
```

2.2.22. Exemple avec `all`

Ici les éléments `<nom>`, `<prénom>` peuvent être dans n'importe quel ordre :

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="personne" type="ElemPersonne"/>
  <xs:complexType name="ElemPersonne">
    <xs:all>
      <xs:element name="prénom" type="xs:string"/>
      <xs:element name="nom"    type="xs:string"/>
    </xs:all>
  </xs:complexType>
</xs:schema>
```

2.2.23. Imbrication de structures

On peut imbriquer plusieurs structures pour définir des éléments à suivre et en option :

```
<xs:complexType name="ElemPersonne">
  <xs:sequence>
    <xs:element name="prénom" type="xs:string"/>
    <xs:element name="nom" type="xs:string"/>
    <xs:choice>
      <xs:element name="age" type="xs:string"/>
      <xs:element name="date_naiss" type="xs:date"/>
    </xs:choice>
  </xs:sequence>
</xs:complexType>
```

Par contre, on ne peut pas faire de mélange avec `<xs:all>`.

2.2.24. Nombre de répétitions

Dans le cas des structures `<xs:sequence>` et `<xs:all>`, il est possible de spécifier un nombre de répétition pour chaque sous-élément.

```
<xs:complexType name="ElemPersonne">
  <xs:sequence>
    <xs:element name="prénom" type="xs:string"
                minOccurs="1" maxOccurs="2"/>
    <xs:element name="nom" type="xs:string"
                minOccurs="0" maxOccurs="1"/>
  </xs:sequence>
</xs:complexType>
```

Par défaut, les nombres de répétitions min et max sont 1. Pour enlever une limite sur le nombre maximal, il faut écrire `maxOccurs="unbounded"`.

2.2.25. Définition d'attributs

Les attributs se déclarent dans un `<complexType>` :

```
<xs:complexType name="ElemPersonne">
    ...
    <attribute name="NOM" type="TYPE" OPTIONS/>
</xs:complexType>
```

nom le nom de l'attribut
type le type de l'attribut, ex: `string` pour un attribut quelconque
options mettre `use="required"` si l'attribut est obligatoire, mettre `default="valeur"` s'il y a une valeur par défaut.

2.2.26. Cas spéciaux

Plusieurs situations sont assez particulières et peuvent sembler très compliquées :

- éléments vides sans ou avec attributs
- éléments textes sans ou avec attributs
- éléments avec enfants sans ou avec attributs
- éléments avec textes et enfants sans ou avec attributs

Voici comment elles sont modélisées avec des Schémas XML.

2.2.27. Élément vide sans attribut

C'est le cas le plus simple :

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="test" type="ElemTest"/>

  <xs:complexType name="ElemTest"/>

</xs:schema>
```

Appliqué au document entier, ça permet de ne valider que :

```
<?xml version="1.0" encoding="utf-8"?>
<test/>
```

2.2.28. Élément vide avec attribut

On rajoute un attribut obligatoire :

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="test" type="ElemTest"/>

  <xs:complexType name="ElemTest">
    <attribute name="att" type="xs:string" use="required"/>
  </xs:complexType>

</xs:schema>
```

Appliqué au document entier, ça permet de ne valider que :

```
<?xml version="1.0" encoding="utf-8"?>
<test att="ok"/>
```

2.2.29. Élément texte sans attribut

Il suffit de définir l'élément en tant que `simpleType` avec un `<xs:restriction>` pour définir son contenu :

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="test" type="ElemTest"/>

  <xs:simpleType name="ElemTest">
    <xs:restriction base="xs:integer"/>
  </xs:simpleType>
</xs:schema>
```

Appliqué au document entier, ça permet de ne valider que :

```
<?xml version="1.0" encoding="utf-8"?>
<test>123</test>
```

2.2.30. Élément texte avec attribut

On doit faire ainsi :

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="test" type="ElemTest"/>
  <xs:complexType name="ElemTest">
    <xs:simpleContent>
      <xs:extension base="xs:integer">
        <xs:attribute name="att" type="xs:string"/>
```

```
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:schema>
```

Appliqué au document entier, ça permet de ne valider que :

```
<?xml version="1.0" encoding="utf-8"?>
<test att="ok">456</test>
```

2.2.31. Éléments enfants sans attribut

C'est comme précédemment, par exemple une séquence :

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="test" type="ElemTest"/>
  <xs:complexType name="ElemTest">
    <xs:sequence>
      <xs:element name="test1"/>
      <xs:element name="test2" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Appliqué au document entier, ça permet de ne valider que :

```
<?xml version="1.0" encoding="utf-8"?>
<test><test1/><test2>texte</test2></test>
```

2.2.32. Éléments enfants avec attribut

Pour valider des attributs sur l'élément parent :

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="test" type="ElemTest"/>
  <xs:complexType name="ElemTest">
    <xs:sequence>
      <xs:element name="test1"/>
      <xs:element name="test2" type="xs:string"/>
    </xs:sequence>
    <attribute name="att" type="xs:string"/>
  </xs:complexType>
</xs:schema>
```

Appliqué au document entier, ça permet de ne valider que :

```xml
<?xml version="1.0" encoding="utf-8"?>
<test att="ok"><test1/><test2>texte</test2></test>
```

2.2.33. Éléments enfants avec texte mélangé

Rajouter l'attribut `mixed="true"` à `<complexType>` :

```xml
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="test" type="ElemTest"/>
  <xs:complexType name="ElemTest" mixed="true">
    <xs:sequence>
      <xs:element name="test1"/>
      <xs:element name="test2" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Appliqué au document entier, ça permet de ne valider que :

```xml
<?xml version="1.0" encoding="utf-8"?>
<test>texte<test1/>texte<test2>texte2</test2>texte</test>
```

2.3. XPath

2.3.1. Présentation rapide

XPath est un mécanisme (syntaxe + fonctions) permettant d'extraire des informations d'un document XML. Par exemple, dans le document messages.xml dont voici un extrait,

```xml
<messages>
  <message numero="1" date="2024-01-01">
    <dest>promo2024</dest>
    <dest bcc="oui">Pierre Nerzic</dest>
    <contenu>Bonne année !</contenu>
  </message>
  ...
```

extraire le contenu du message n°4 s'écrit ainsi en XPath :

```
/messages/message[@numero=4]/contenu
```

2.3.2. Évaluation d'une expression XPath

Pour évaluer une expression en ligne de commande, il y a :

- `xmlstarlet sel --template --value-of` *expression document.xml*
- `xmllint --xpath` *expression document.xml*

Pour les navigateurs, je vous fournis ce formulaire. Étudiez son source. C'est basé sur une `XMLHttpRequest` et la méthode `evaluate` sur le document XML qu'on reçoit.

2.3.3. Cadre général

XPath sert à extraire des informations dans un arbre XML à l'aide d'une sorte de chemin.

Soit ce document XML :

```xml
<?xml version="1.0" ...?>
<messages>
  <message numero="4">
    <dest bcc="oui">promo2023</dest>
    <contenu>Salut</contenu>
  </message>
</messages>
```

NB: le document complet contient tous les messages.

Voici son arbre XML :

2.3.4. Chemin dans l'arbre du document

Le but d'XPath est d'aller chercher les informations voulues dans le document XML. Ex: quels sont les destinataires du message n°2 ?

Cela se fait à l'aide d'un chemin d'accès qui ressemble beaucoup à un nom complet Unix, mais avec des conditions écrites entre `[]`.

Exemple :

```
/messages/message[@numero="2"]/dest
```

C'est un peu comme un nom complet absolu dans Unix. Cependant, il y a énormément plus de possibilités pour écrire ces chemins et d'autre part, les chemins peuvent retourner plusieurs résultats.

2.3.5. Réponses multiples

Un point très important est qu'une expression XPath peut retourner plusieurs réponses. En effet, contrairement à Unix, un élément parent peut contenir plusieurs exemplaires du même élément enfant. Le chemin `/messages/message/dest` sélectionne 4 élements.

2.3.6. Structure d'une expression XPath simple

Une expression XPath est une suite d'étapes séparées par des séparateurs : `[sep] étape1 sep étape2 sep étape3...` Les étapes sont les noms des éléments dans lesquels il faut aller successivement.

Cependant cela dépend du séparateur employé :

- `/` Ce séparateur se comporte comme dans Unix : s'il est mis au début du chemin, il représente le document entier ; s'il est mis entre les étapes, c'est un simple séparateur.
- `//` Ce séparateur signifie de sauter un nombre quelconque d'éléments quelconques. C'est un peu comme si on écrivait `/*/*/*.../` un nombre indéfini de fois, y compris 0. S'il est au début du chemin, cela signifie alors que la première étape est à chercher n'importe où dans l'arbre.

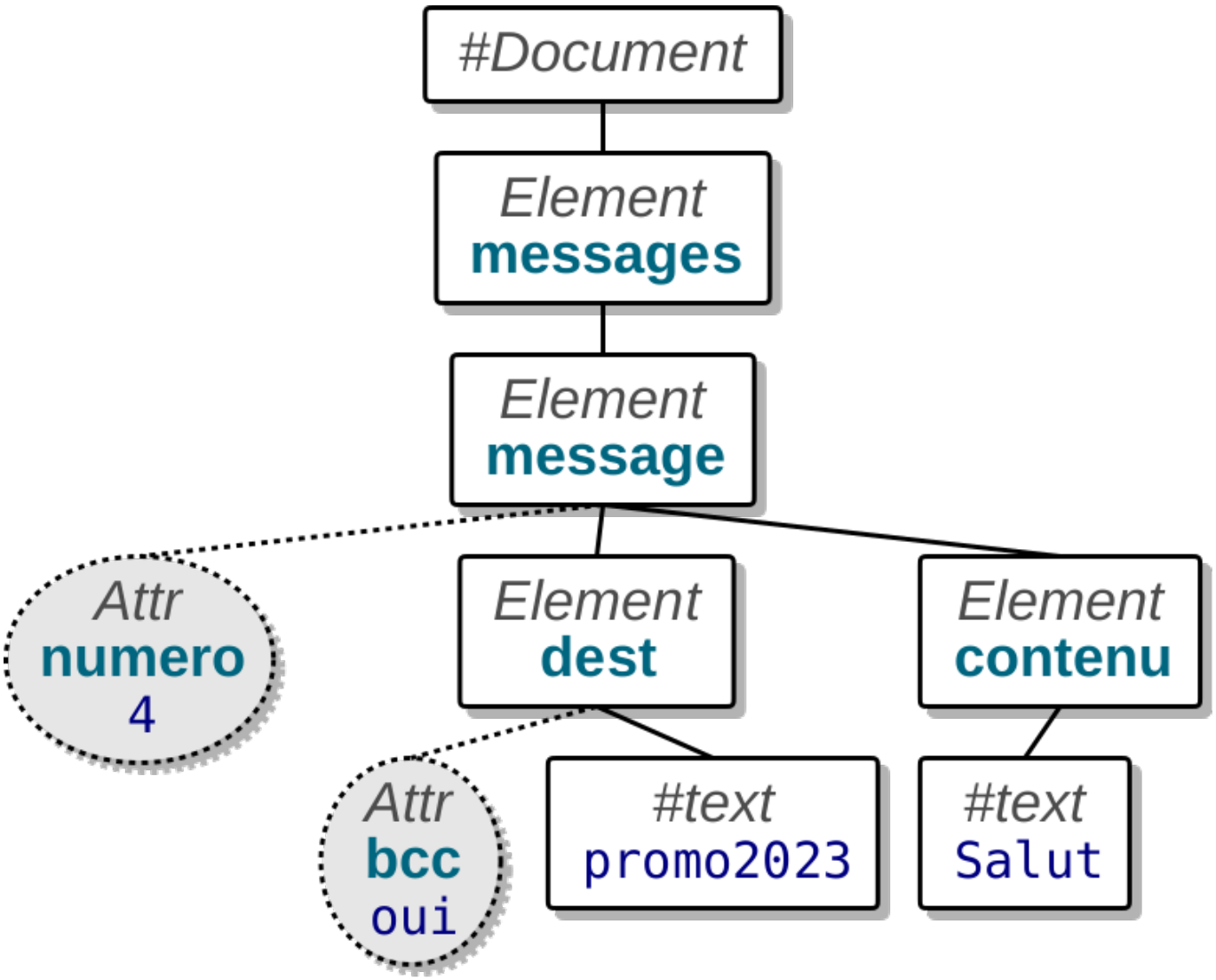

Figure 8: Arbre XML

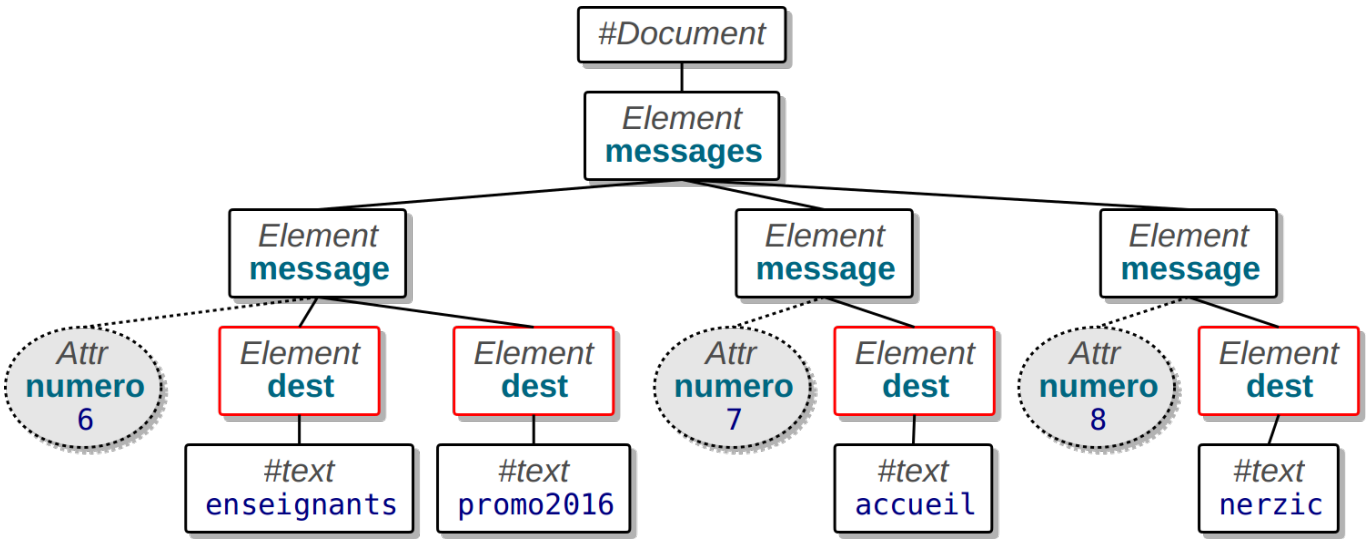

Figure 9: Arbre XML

2.3.7. Attributs des éléments

Pour désigner un attribut et non pas un sous-élément, on met un `@` devant le nom de l'attribut.

Exemples :

- `/messages/message/@numero` sélectionne tous les attributs nommés `numero` des éléments `<message>`.
- `//@numero` sélectionne tous les attributs portant ce nom sur n'importe quel élément du document.

NB: XPath retourne des nœuds de type Attribut sous la forme `nom="valeur"`. Pour avoir seulement la valeur, il faut écrire `string(chemin)`. Par exemple, `string(//message[dest="promo2024"]/@numero)`. Attention `string()` exige qu'il n'y ait qu'un seul attribut sélectionné par l'expression XPath.

2.3.8. Autres étapes d'un chemin

D'autres étapes peuvent être employées :

- `.` désigne le nœud courant,
- `..` désigne le nœud parent,
- `*` désigne tous les éléments de ce niveau. C'est plus restreint que `//`.

Exemples :

- `/messages/*/@numero` sélectionne tous les attributs nommés `numero` des éléments situés sous `<messages>`
- `//contenu[../@numero=3]` sélectionne les éléments `<contenu>` tels que leur élément parent a un attribut appelé `numero` et valant 3.

2.3.9. Conditions sur les étapes

Les conditions sur les étapes (éléments et attributs) sont appelées *prédicats*. Un prédicat se met entre `[...]` juste après l'élément dont il filtre l'un des enfants. On peut enchaîner plusieurs prédicats.

Exemple :

- `//message[@numero=5]/contenu` sélectionne les `<contenu>` des éléments `<message>` dont l'attribut `numero` vaut 5.
- `//message[contenu=""]/@numero` sélectionne les attributs `numero` des `<message>` ayant un sous-élement `<contenu>` vide.
- `//message[dest="iut"][@date="2024-01-01"]` sélectionne les éléments ayant un sous-élément `<dest>` valant « iut » et un attribut date valant « 2024-01-01 ».

Attention à bien lier le prédicat à l'élément concerné :

- `//message[./@numero="3"]` sélectionne seulement les messages ayant un attribut `numero` valant 3.
- `//message[//@numero="3"]` sélectionne *tous* les messages, si jamais on trouve quelque part un attribut `numero` valant 3

Dans `/element[condition]`, il faut que la condition concerne seulement l'élément.

2.3.10. Syntaxe des prédicats

Pour écrire les prédicats, XPath propose ces opérateurs un peu différents de ceux du C :

- arithmétique : `+ - * div mod` (et non pas `/` et `%`)
- comparaisons : `< <= = != >= >` (et non pas `==`)
- logique : `and or not(condition)`

Exemples :

- `//message[not(@numero < 5 or @numero >= 9)]` sélectionne les `<message>` dont l'attribut `numero` est entre 5 et 8.
- `//message[@numero mod 5 = 0]` sélectionne les `<message>` dont l'attribut `numero` est un multiple de 5.

À part les conditions basées sur des comparaisons, il y a aussi :

- la notation `[`*index*`]` sélectionne l'élément ayant cet index (1 à n) dans la liste de son parent,
- le prédicat `[`*enfant*`]` est vrai si l'élément contient cet enfant.

Exemples :

- `//message[7]/contenu` sélectionne le `<contenu>` du 7e élément `<message>` du document.
- `//message[contenu and not(@date)]` sélectionne les `<message>` qui ont un `<contenu>` mais pas d'attribut date.

2.3.11. Fonctions XPath

XPath possède de très nombreuses fonctions, dont :

- Fonctions sur les éléments :
  - `string(s)` retourne le texte de l'expression `s`
  - `position()` retourne l'index de l'élément dans son parent (premier = n°1)
  - `last()` retourne le n° du dernier élément dans son parent

Exemples :

- `string(/messages/message[2]/contenu)` retourne le contenu du 2e message.
- `/messages/message[position()<=3]` sélectionne les 3 premiers éléments `<message>` du document.

- `//dest[position()>last()-3]` sélectionne les `<dest>` qui sont parmi les trois derniers enfants de leur parent.

2.3.12. Fonctions XPath (suite)

Une fonction est particulièrement utile : `count(expression)`. Elle compte le nombre de nœuds XML (élément, attributs, textes...) sélectionnés par l'expression. On l'utilise dans des conditions.

Attention, count() compte les nœuds sélectionnés, chacun dans son parent *séparément*. Ça conduit à faire des erreurs si on croit que `count` peut regrouper différents comptages.

Exemples :

- `//message[count(dest)>2]` retourne les éléments `<message>` ayant plus de deux enfants `<dest>`.
- `//message[count(//dest)>2]` retourne tous les éléments `<message>` s'il y a plus de deux éléments `<dest>` quelque part dans le document.

2.3.13. Fonctions XPath (suite)

- Fonctions sur les chaînes :
  - `string-length(s)` retourne la longueur de la chaîne `s`
  - `concat(s1, s2,...)` concatène les chaînes passées
  - `substring(s,deb,lng)` retourne `lng` caractères de `s` à partir du n°`deb` (premier = 1)
  - `contains(s1, s2)` vrai si `s1` contient `s2`
  - `starts-with(s1,s2)` et `ends-with(s1,s2)`
  - `matches(s, motif)` vrai si `s` correspond au motif

Exemple :

- `//message[string-length(contenu)<=15 and not(starts-with(dest, "promo"))]/@numero` retourne les numéros des messages dont le contenu ne fait pas plus de 15 caractères et aucun destinataire ne commence par « promo ».

2.3.14. Fonctions XPath (suite)

- Fonctions mathématiques :
  - `abs(nb)`, `ceiling(nb)`, `floor(nb)`, `round(nb)`
- Fonctions sur les dates et heures :
  - `year-from-dateTime(dt)`, `month-from-dateTime(dt)`, `day-from-dateTime(dt)`, `hours-from-dateTime(dt)`, `minutes-from-dateTime(dt)`, `seconds-from-dateTime(dt)`
  - `year-from-date(d)`, `month-from-date(d)`, `day-from-date(d)`
  - `hours-from-time(t)`, `minutes-from-time(t)`, `seconds-from-time(t)`

2.3.15. Retour sur les composants d'un chemin

Un chemin XPath est constitué de `[sep] étape1 sep étape2 sep étape3...` Chaque étape est soit le nom d'un élément, soit `@` et le nom d'un attribut ; chacune suivie éventuellement d'un prédicat entre crochets :

```
/racine/element1[filtre1]/.../@attribut[filtre3]
```

Les étapes sont appelées *sélecteurs*. À la fin d'une expression, on peut employer des sélecteurs spéciaux comme :

`text()` sélectionne tous les textes sous l'élément courant, y compris dans tous ses descendants.
`node()` sélectionne tous les nœuds enfants de l'élément.

Exemple :

- `/messages/message/contenu/text()`

2.3.16. Axes

XPath permet de rajouter encore une « décoration » sur chaque étape, la *direction* dans laquelle aller à partir de l'étape courante. Cette direction est appelée *axe*. Cela donne la syntaxe :

```
/racine/axe1::element1[filtre1]/axe2::element2[filtre2]/...
```

Par défaut, on descend toujours vers les enfants du nœud courant au niveau de chaque étape. Cet axe s'appelle `child`.

Exemple, ces deux syntaxes signifient la même chose :

- `/messages/message/@numero`
- `/messages/child::message/attribute::numero`

D'autres axes existent.

2.3.17. Axes

L'axe définit vers quels nœuds aller à chaque étape. Voici quelques axes utiles à connaître parmi ceux qui existent :

`child::` parcourir les nœuds enfants du contexte ; c'est l'axe utilisé par défaut.
`descendant::` parcourir tous les nœuds enfants et petit-enfants ; ça revient un peu à utiliser `//`.
`parent::` parcourir le nœud parent du contexte ; ça revient à utiliser `..` mais avec un test sur le parent voulu.
`ancestor::` parcourir tous les nœuds parent et grand-parents.
`preceding-sibling::` parcourir tous les nœuds frères précédents
`following-sibling::` parcourir tous les nœuds frères suivants
`attribute::` parcourir les nœuds attributs du contexte ; c'est l'axe par défaut pour une étape commençant par un `@`.

2.3.18. Exemples de chemins avec axes

- `/messages/message[last()]/child::contenu` retourne le contenu du dernier message du document.
- `/messages/message[@numero=7]/descendant::dest` sélectionne tous les nœuds situés sous le message n°7.
- `//message[@numero=5]/preceding-sibling::message` sélectionne les messages situés avant le n°5.
- `//dest[@bcc="oui"]/parent::node()` sélectionne le nœud parent d'un élément `<dest>` dont l'attribut `bcc` vaut oui.
- `//message[3]/attribute::numero` retourne l'attribut numéro du 3e message présent dans le document.

2.3.19. C'est tout pour aujourd'hui

Cette présentation est finie. Rendez-vous avec le TP5 et les suivants pour mettre cela en pratique.

Semaine 3

XML

Dans ce cours :

- Tests fonctionnels
  - Robot Framework et Selenium
- Intégration continue
  - Pipelines GitLab
- Bibliographie :
  - partie tests fonctionnels : horustest.io C'est une entreprise qui propose une solution clé en mains, HorusTest, pour définir des tests fonctionnels sur des logiciels Web. Leur blog, écrit par Stéphanie Binet, est remarquable.

3.1. Tests fonctionnels

3.1.1. Tests fonctionnels ?

Ce sont des tests du logiciel dans sa globalité, tel que le verra l'utilisateur.

Le but est de vérifier les fonctionnalités du logiciel par rapport aux spécifications, cahier des charges, cas d'utilisation...

Cela consiste en l'exécution de **scénarios d'utilisation** : par exemple pour un logiciel Web, ouvrir le logiciel par son URL, cliquer sur des boutons et des menus, remplir un formulaire, etc. Les tests simulent un utilisateur dans son usage du logiciel.

Les scénarios sont composés d'actions, chacune étant accompagnée de vérifications validant les exigences du client.

3.1.2. Buts des tests fonctionnels

Garantir la qualité du logiciel :

- vérifier que les exigences client sont réalisées,
- détecter des défauts au niveau de ce que verra utilisateur,
- éviter des régressions sur des fonctionnalités existantes,
- évaluer les performances,
- en retour, améliorer les phases du développement en tenant compte des priorités sur les besoins utilisateur.

3.1.3. Positionnement des tests fonctionnels parmi les tests

Les tests fonctionnels sont, par exemple, effectués par une sorte d'automate qui effectue des actions (clics, saisies, vérifications) sur un navigateur internet. Toutes ces actions ne sont pas instantanées : délais réseau et temps de calculs du navigateur pour mettre à jour l'interface.

Les tests fonctionnels sont donc très lents, comparés aux tests unitaires et d'intégration. Ils sont donc effectués après ces derniers, uniquement s'ils sont réussis.

Les tests fonctionnels sont difficiles à mettre en place dans un projet Agile, et encore plus dans l'approche *Test Driven Dev*, car les spécifications définitives ne sont pas disponibles.

3.1.4. Avantages des tests fonctionnels automatisés

Par rapport à des tests réalisés par des humains, les tests fonctionnels automatiques sont :

- plus rapides, en particulier parce qu'ils sont parallélisables,
- reproductibles,
- adaptables à différents jeux de données,
- planifiables au moment souhaité, par exemple la nuit.

Cercle vertueux : en augmentant le nombre de tests fonctionnels automatiques, cela augmente la couverture du code et des fonctionnalités, donc ça améliore la qualité du logiciel, et en conséquence, ça augmente aussi la confiance de l'équipe et du client.

3.1.5. Remarques sur les tests fonctionnels automatisés

- Il est souhaitable que le test fonctionnel automatique se comporte comme l'utilisateur final humain. Ça peut être compliqué, étant donné l'étendue des possibilités sur les interfaces actuelles : clics souris, raccourcis clavier, etc.
- Les tests automatisés font des contrôles à chaque étape : présence/visibilité de tel ou tel élément sur l'écran, mais il est difficile d'avoir la même intelligence qu'un humain pour analyser ce qui est affiché, par exemple les positions et tailles relatives des éléments, la vitesse d'affichage, etc.
- Il est quasiment impossible de tout prévoir, c'est à dire de programmer toutes les actions et vérifications possibles/nécessaires pour valider le cahier des charges. On doit se limiter à ce qui est le plus pertinent pour obtenir la qualité voulue.

3.1.6. Limitations des tests fonctionnels automatisés

- Un système automatique de tests ne peut pas avoir l'esprit inventif des humains pour essayer des choses imprévues, et trouver des bugs ($\Rightarrow$ les béta testeurs sont des humains).
- Les tests sont souvent complexes et difficiles à maintenir. Quand le logiciel évolue, il faut du temps pour adapter les tests.
- Les tests fonctionnels peuvent demander des ressources coûteuses et/ou complexes, ex : un serveur Web réel avec un SGBD. Certains fournisseurs d'accès internet offrent des services dématérialisés pour cela.
- Contrairement aux tests unitaires, les tests fonctionnels vérifient simultanément un grand nombre d'assertions. Il peut être très difficile de caractériser un bug lors d'un échec de test.

3.1.7. Construction des tests fonctionnels

Il existe des outils pour construire un scénario de test sans programmation. Cela consiste à enregistrer des actions manuellement, puis à les faire rejouer automatiquement, par exemple des *macros*.

Le problème est que ces scénarios sont généralement trop rigides, et n'incluent pas des vérifications à chaque étape.

La suite du cours présente un outil appelé **Robot Framework** qui ne demande que quelques notions de programmation. C'est parce que les tests fonctionnels sont généralement spécifiés par les experts fonctionnels et métier, comme le Chef de Produit, qui sont moins dans la technique que les développeurs/euses.

3.2. Robot Framework

3.2.1. Présentation de Robot Framework

À l'origine, en 2005, c'était la proposition d'un chercheur, Pekka Klärck (LinkedIn), d'exprimer des tests fonctionnels à l'aide de mots-clés. C'est à dire qu'au lieu d'écrire un programme, comme on le fait avec JUnit, on écrit des spécifications avec un langage basé sur des phrases simplifiées.

Voici un exemple :

```
*** Test Cases ***
Test Chercher Robot Framework sur Wikipedia
    Go To           https://www.wikipedia.fr/
    Input Text      id=search     Robot Framework
    Click Element   //img[@id="search-icon"]
    Wait Until Location Is  https://www.wikipedia.fr/wiki/Robot_Framework
    Capture Page Screenshot
```

3.2.2. Principes généraux

Robot Framework se présente sous la forme d'une commande shell (programmée en Python), accompagnée de librairies pour gérer les navigateurs, les bases de données, etc.

La commande shell s'appelle `robot` et elle demande un nom de script de test en paramètre. Un script robot contient des directives indiquant des actions ou des vérifications à faire :

- ouvrir tel URL sur un navigateur
- vérifier que tel élément est présent
- cliquer sur tel élément
- vérifier que la base de données contient tel enregistrement,
- vérifier qu'on est maintenant sur tel URL
- etc.

3.2.3. Syntaxe générale des scripts robot

Un script de test doit avoir une mise en page très particulière. À première vue, ça ressemble à du texte mélangé à un fichier CSV, c'est à dire des chaînes séparées par des tabulations, et indentées comme en Python. Ces chaînes sont appelées « mots-clés » et peuvent avoir des paramètres.

```
NOM DU TEST
    MOT CLÉ 1    PARAM 1    PARAM 2
    MOT CLÉ 3    PARAM 4
    ...
```

Les majuscules/minuscules ne comptent pas mais il faut **impérativement** qu'il y ait au moins **deux espaces** entre deux mots-clés. Raison : les mots-clés peuvent être écrits avec plusieurs mots séparés par un seul espace.

3.2.4. Exemple

Voici un exemple de script, pour observer cette syntaxe :

```
*** Settings ***
Documentation   Test du tuto CodeIgniter
Library         SeleniumLibrary
Library         DatabaseLibrary

*** Test Cases ***
Endpoint test6 doit afficher la liste des produits
    Go To                       http://localhost:8000/test6
    Wait Until Page Contains    Voici la liste des produits :
    Page Should Contain Link    /test6/nouveau
    Capture Page Screenshot
```

`Go To`, `Wait Until Page Contains`... sont des « mots-clés ».

3.2.5. Sections d'un script

Un script robot est composé de plusieurs sections :

- initialisation générale : importation de librairies, commentaires

  `*** Settings ***`

- suite de tests

  `*** Test Cases ***`

- définition de mots-clés, de variables...

  ```
  *** Keywords ***
  *** Variables ***
  ```

3.2.6. Section `Settings`

Cette section sert à importer des librairies, c'est à dire des collections de mots-clés et variables permettant d'effectuer certaines tâches, et configurer les tests.

```
*** Settings ***
Library  SeleniumLibrary
Test Setup     Open Browser   http://localhost:8000/  firefox
Test Teardown  Close Browser
```

- `Library` fait charger une librairie de mots-clés et fonctions internes, voir transparent 51.
- `Test Setup` définit l'initialisation de chaque test quand elle est identique d'un test à l'autre (comme `@BeforeEach` de JUnit)
- `Test Setup` définit la clôture commune de chaque test (comme `@AfterEach`)

3.2.7. Section `Test Cases`

C'est là qu'on place les scénarios de test. Tous les tests doivent être présentés ainsi, indentés comme en Python :

```
*** Test Cases ***
Nom du test
    Mot clé     premier param     2e param      ...
    ...
```

Les paramètres dépendent des mots-clés. Voir transparent 51.

Il peut également y avoir des directives juste devant le mot-clé :

- `[Setup]`, `[Teardown]` voir transparent 51,
- `[Timeout]` *durée* spécifie un temps maximal, voir la doc,
- `[Documentation] texte` pour des informations.

3.2.8. Variante de syntaxe pour les tests

Robot Framework offre une autre syntaxe, permettant de mettre en valeur le patron AAA. Ce sont des mots-clés `Given`, `When`, `Then` et `And`. `Given` remplace *Arrange*, `When` remplace *Act*, et `Then` remplace *Assert*.

```
Test Wikipedia's Robot Framework page
    Given Open Browser  https://www.wikipedia.fr/  firefox
    When Input Text     id=search     Robot Framework
    And Click Element   //img[@id="search-icon"]
    Then Page Should Contain    est un framework de test
```

Le mot-clé `And` permet de continuer comme le précédent mot-clé.

3.2.9. Section `Variables`

Cette section permet de définir des variables qui seront accessibles dans tout le script. Il y a trois syntaxes :

- `${NOM}` pour une variable contenant une simple chaîne.
- `@{NOM}` pour une variable contenant une liste. Les éléments doivent être séparés d'au moins deux espaces.

- `&{NOM}` pour un dictionnaire de paires clé-valeur.

Voici un exemple :

```
*** Variables ***
${URL} =      http://localhost:8000/
@{WINSIZE} =  1280   1024
&{TEST1} =    user=uti1   pass=uti1pw   vrainom=Pierre
```

Le signe = après le nom de la variable est facultatif.

3.2.10. Utilisation des variables

- Pour les variables simples, il suffit d'écrire `${NOM}`
- Pour une liste, `@{NOM}` vaut toute la liste, et la notation `@{NOM}[ind]` fonctionne exactement comme en Python.
- Pour un dictionnaire, `&{NOM.clé}` et `&{NOM}[clé]` retournent la valeur associée.

```
*** Test Cases ***
Check login
    Open Browser      ${URL}/main.html    headlessfirefox
    Set Window Size   @{WINSIZE}
    Input Text        //input[@name="user"]  &{TEST1.user}
    Input Text        //input[@name="pass"]  &{TEST1.pass}
    Click Button      id=Login
    Wait Until Page Contains    Bonjour &{TEST1}[vrainom]
```

3.2.11. Section `Keywords`

Pour définir de nouveaux mots-clés utilisables dans les tests :

```
*** Keywords ***
Start Services
    ${PHP} =  Start Process   /usr/bin/php  -S  ${HOST}:${PORT}
    Set Suite Variable  ${PHP}

Stop Services
    Close Browser
    Terminate Process   ${PHP}
```

Le mot-clé `Set Suite Variable` rend la variable globale dans la suite de tests. On peut alors l'employer dans l'autre mot-clé.

Les mots-clés `Start Services` et `Stop Services` pourront être employé dans les tests, ou bien au niveau de la suite de tests.

Une suite de test est un ensemble de tests qui partagent la même initialisation/terminaison. Il est préférable de :

- Définir l'initialisation et la terminaison en tant que mots-clés, comme dans le transparent précédent
- Indiquer que le démarrage et l'arrêt de l'ensemble des tests se font avec ces mots-clés :

```
*** Settings ***
Suite Setup     Start Services
Suite Teardown  Stop Services
```

- `Suite Setup` est comme `@BeforeAll` de JUnit, une initialisation commune exécutée une seule fois, au tout début,
- `Suite Teardown` est comme `@AfterAll` faite une fois à la fin.

3.2.12. Démarrage ou terminaison spécifique

S'il y a un mot-clé spécifique pour le démarrage ou l'arrêt d'un test, on le fait précéder par :

- `[Setup]` pour un mot-clé qui initialise le test,
- `[Teardown]` pour ce qui termine le test même s'il a échoué, et s'il y a plusieurs `[Teardown]`, ils sont tous faits, même si certains échouent.

Exemple :

```
*** Test Cases ***
Test de la page d'accueil de Google
    [Setup]     Log   test de base sur la page google.fr
    Open Browser      https://www.google.fr
    Title Should Be   Google
    [Teardown]  Close Browser
```

3.2.13. Librairies

Il y a un grand nombre de mots clés pour toutes sortes d'interactions avec le navigateur : actions sur la page Web et vérifications sur le contenu.

Ces mots-clés sont définis dans des bibliothèques (*library* en anglais).

On inclut celles dont on a besoin dans la section `Settings`.

Exemple :

```
*** Settings ***
Library   Collections
Library   SeleniumLibrary
Library   JSONLibrary
```

3.2.14. Librairie `SeleniumLibrary`

Cette librairie sert à interagir avec un navigateur. Sa documentation est sur cette page. Elle définit de très nombreux mots-clés, 173 au total, pour :

- contrôler le navigateur : Open Browser, Close Browser. En général, on utilise un navigateur sans interface graphique visible, par exemple `headlessfirefox`.
- naviguer sur des URLs : Go To, Go Back, Wait Until...
- vérifier le contenu affiché : Page Should Contain..., Element Should..., Get... avec de très nombreuses variantes
- cliquer sur un élément : Click..., Double Click Element, Submit Form
- saisir des valeurs dans des *input* : Input...
- cocher des checkbox et radio : Select...

3.2.15. Désignation des éléments

Un point important est la désignation de l'élément du DOM HTML qu'on souhaite manipuler, par exemple un `<button>` pour cliquer dessus. Selenium appelle ça un *locator* (documentation).

```
Click Element   id=btn-submit
```

Il y a deux syntaxes :

- le mode implicite (par défaut) qui dépend du mot-clé, par exemple pour le mot-clé `Click Button`, Selenium recherche l'attribut `id`, `name` ou `value` qui correspond au paramètre.
- le mode explicite signalé par un préfixe :
  - `id:truc`, `name:truc`, `class:truc`
  - `css:selecteur` désigne l'élément qui possède ce sélecteur
  - `xpath:chemin` désigne l'élément qui correspond au chemin. Ce mode est aussi implicite, car le préfixe `xpath:` est optionnel.

3.2.16. Librairie `DatabaseLibrary`

Elle sert à interagir avec un SGBD. Sa documentation est sur cette page. Elle définit 15 mots clés pour :

- gérer une connexion `Connect To Database` *paramètres*, `Disconnect From Database`
- exécuter un script SQL `Execute Sql Script` *nomscript* ce qui permet de remplir une base avec des valeurs prédéfinies
- vérifier la présence d'une table `Table Must Exist` *nomtable*
- compter les n-uplets sélectionnés par une requête `Row Count Is Equal To X` *select nb*
- supprimer des n-uplets `Delete All Rows From Table` *nomtable*

3.2.17. Librairie `Process`

Elle sert à lancer des processus de type « démon », par exemple un serveur Web. Sa documentation est sur cette page.

Elle définit 15 mots-clés, dont ceux-ci :

- lancer un processus Run Process qui attend la fin du processus, et Start Process qui le lance en arrière-plan. Ce mot-clé retourne le PID du processus, à mettre dans une variable,
- arrêter un processus, Terminate Process à qui on fournit le PID,
- vérifier l'état d'un processus Is Process Running, Process Should Be Running.

3.2.18. Exécution des tests

Les scripts doivent porter l'extension `.robot`

- exécuter un seul script

  ```
  robot script.robot
  ```

- exécuter tous les scripts d'un dossier

  ```
  robot dossier
  ```

La commande crée plusieurs fichiers dans un dossier « `reports` » :

- un bilan dans un fichier `report.html`
- des compte-rendus `output.xml` et des `*.log`
- des copies d'écran par le mot-clé `Capture Page/Element Screenshot`

3.3. Intégration continue et livraison continue

3.3.1. Intégration continue ?

Cela consiste à systématiquement appliquer les suites de tests et de couverture lors de chaque enregistrement (*commit*) du code source.

Dans une équipe comprenant plusieurs développeurs/euses, chacun/e travaille sur des parties du projet. L'intégration consiste à regrouper les différents travaux et faire en sorte qu'ils soient compatibles et qu'ils passent les tests.

Le logiciel Git permet de travailler relativement séparément puis de regrouper les contributions. Les logiciels Web GitHub et GitLab possèdent un dispositif, « CI/CD » (*Continuous Integration/Continuous Deployment*), pour déclencher automatiquement les suites de tests à chaque *commit+push*.

3.3.2. Déroulement de l'intégration continue (CI)

- Il faut avoir un système de dépôt de code source avec versions et qui gère l'intégration continue.
- On y définit ce qui s'appelle un **pipeline de test**, une succession d'étapes devant réussir :
  1. compilation du code source,
  2. application de tous les tests unitaires, d'intégration et fonctionnels,
  3. mesure de la qualité
  4. génération des livrables pour les clients (installateur Windows, paquet APK ou Debian, archive, etc.).

Chaque étape peut échouer. Une sorte de tableau de bord permet d'informer les utilisateurs.

3.3.3. Livraison et déploiement continus ?

C'est la suite de l'intégration continue. La livraison continue concerne l'exécution du logiciel dans un environnement réaliste, par exemple sur des machines virtuelles, similaires aux machines des clients, afin de faire des tests fonctionnels en situation et des mesures de performance.

Par exemple, on vérifie que le logiciel s'installe bien et tourne bien sur toutes les versions de Windows, avec des PC plus ou moins performants, etc.

La livraison continue débouche sur le déploiement continu : la mise à disposition du logiciel pour les clients.

Ce cours ne pourra pas traiter ces aspects, faute de temps.

3.3.4. Intégration continue dans GitLab

En TP, nous utiliserons GitLab qui intègre tous les outils utiles : git et un ordonnanceur d'intégration continue.

NB: nous allons utiliser seulement un tout petit sous-ensemble des possibilités. C'est parce qu'il n'est pas possible de vous rendre administrateur des projets. Vous serez obligés de rester dans un tout petit cadre, mais qui permet quand même de se faire une idée générale.

Pour l'intégration continue (CI/CD), le principe est de définir un *pipeline* à l'aide d'un script dédié.

3.3.5. Définition d'un pipeline

Créer un fichier appelé `.gitlab-ci.yml` à la racine du projet et contenant par exemple ceci :

```
RobotFramework tests:
  tags:
    - robotframework
  stage: test
  variables:
    OUTDIR: /usr/local/reports
  script:
    - mkdir -p $OUTDIR
    - robot --outputdir $OUTDIR Tests
```

Ce script fait passer les tests Robot Framework à votre projet. Ces tests sont présents dans un dossier appelé `Tests` à la racine du projet. Les résultats vont dans `/usr/local/reports`.

3.3.6. Exécution d'un pipeline

Voici ce qu'on voit quand un job s'exécute, le même résultat qu'avec Robot Framework :

3.3.7. Tableau de bord des pipelines

Voici une partie de l'interface montrant un pipeline exécuté avec succès :

L'exécution du pipeline a été déclenché par un *commit* appelé « import initial ».

Nous verrons en TP ce qui se passe quand un pipeline échoue, et comment les développeurs sont invités à corriger le problème.

3.3.8. Fin

C'est tout pour aujourd'hui. Rendez-vous en TP pour la mise en pratique.

```
35 ==============================================================================
36 Tests.TestMainHTML
37 ==============================================================================
38 Check main.html                                                       | PASS |
39 ------------------------------------------------------------------------------
40 Tests.TestMainHTML                                                   | PASS |
41 1 test, 1 passed, 0 failed
42 ==============================================================================
43 Tests                                                                | PASS |
44 3 tests, 3 passed, 0 failed
45 ==============================================================================
46 Output:  /usr/local/reports/root/projet1/4f6e2370/output.xml
47 Log:     /usr/local/reports/root/projet1/4f6e2370/log.html
48 Report:  /usr/local/reports/root/projet1/4f6e2370/report.html
50 Job succeeded
```

Figure 10: Détail d'un job

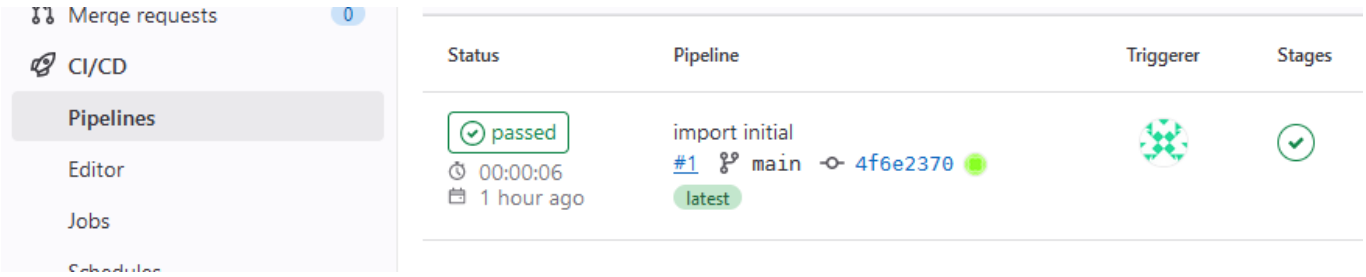

Figure 11: Pipelines