# Exercices

Gerson Sunyé

2025-01-13

# Table des matières

# Chapitre 1. Cours magistraux

## 1.1. Cours magistraux

*Main Topics*

- Introduction to Software Construction and Evolution [https://gl.univ-nantes.io/construction/slides/introduction.html]
- Software Evolution [https://gl.univ-nantes.io/construction/slides/evolution.html]

*Methods*

- Agile Software Development [https://gl.univ-nantes.io/construction/slides/agile.html]
- Continuous Integration [https://gl.univ-nantes.io/construction/slides/ci.html]

*Techniques*

- Coding Techniques [https://gl.univ-nantes.io/construction/slides/coding.html]
- Design Patterns [https://gl.univ-nantes.io/construction/slides/patterns.html]
  - Singleton [https://patterns.univ-nantes.io//singleton.html]
  - Decorator [https://patterns.univ-nantes.io//decorator.html]
  - Abstract Factory [https://patterns.univ-nantes.io//abstract-factory.html]
  - Flyweight [https://patterns.univ-nantes.io//flyweight.html]
  - Iterator [https://patterns.univ-nantes.io//iterator.html]
  - Factory Method [https://patterns.univ-nantes.io//factory-method.html]
- Mapping Designs to Code - Part I [https://gl.univ-nantes.io/construction/slides/mapping.html]
- Mapping Designs to Code - Part II [https://gl.univ-nantes.io/construction/slides/behavior.html]
- Build Automation [https://gl.univ-nantes.io/construction/slides/build.html]
- Code Smells [https://gl.univ-nantes.io/construction/slides/smells.html]
- Refactorings [https://gl.univ-nantes.io/construction/slides/refactorings.html]
- Software Testing Basics [https://gl.univ-nantes.io/construction/slides/test.html]

## 1.2. Organisation

| | Heures | Séances |
|---|---|---|
| Cours Magistraux | 14 | 10,5 |
| Travaux dirigés | 16 | 12 |
| Travaux pratiques | 12 | 9 |

## 1.3. Modalités de contrôle continu

Cette UE correspond à 5 ECTS.

*Tableau 1. Première session*

| | |
|---|---|
| Examen | 50% |
| Contrôle continu écrit | 25% |
| Contrôle continu pratique | 25% |

*Tableau 2. Deuxième session*

| | |
|---|---|
| Examen | 60% |
| Contrôle continu écrit | 20% |

| | |
|---|---|
| Contrôle continu pratique | 20% |

## 1.4. Références


- P. Bourque and R.E. Fairley, eds., Guide to the Software Engineering Body of Knowledge (SWEBOK) [https://www.swebok.org], Version 3.0, IEEE Computer Society, 2014.
- «Code Complete» [https://www.microsoftpressstore.com/store/code-complete-9780735619678]. Steve McConnell. 2nd edition. Microsoft Press.
- «JUnitTest Infected: Programmers Love Writing Tests». Erich Gamma and Kent Beck. Java Report: Volume 3, Number 7. July 1998.
- «Extreme programming explained: embrace change. Kent Beck. Addison-Wesley, 1999.
- «Test-Driven Development: By Example». Kent Beck. Addison-Wesley, 2002.
- «Test-Driven Development: A Practical Guide». David Astels. Prentice Hall, 2003.
- «Head First Design Patterns: A Brain-Friendly Guide». Bert Bates, Kathy Sierra, Eric Freeman, Elisabeth Robson. O'Reilly Media, June 2009.
- The Unified Modeling Language [https://www.uml-diagrams.org]
- Introduction to the Diagrams of UML 2.X [http://www.agilemodeling.com/essays/umlDiagrams.htm]
- JUnit [https://junit.org]

# Chapitre 2. Exercices de TD

## 2.1. Introduction

### 2.1.1. Environnements de développement

- Pour la programmation Web en TypeScript, utilisez Visual Studio Code [https://code.visualstudio.com]
- Pour la programmation Java, utilisez IntelliJ IDEA [https://www.jetbrains.com/idea/]
- The Unified Modeling Language [https://www.uml-diagrams.org]

## 2.2. Objets mutables et immuables

### 2.2.1. Introduction

Les objets **`mutables`** sont des objets qui peuvent être modifiés après leur création, contrairement aux objets **`immuables`**, qui eux, ne peuvent pas être modifiés. En Java, la plupart des objets sont mutables, à quelques exceptions près, comme les classes `URL` et `File` et les *classes-enveloppe* : `Integer`, `Byte`, `Float`, `Char`, etc.

Les *Classes-enveloppe* ou *Wrappers*, sont des classes qui encapsulent des données de type primitif, pour les utiliser comme des objets ordinaires.

Par exemple, la classe `String` et immuable, contrairement à `StringBuilder`, qui elle est mutable Considérons l'extrait Java suivant :

```
1 String message = "In Java, String objects are ";
2 message = message + "immutable";
3
4 StringBuilder builder = new StringBuilder();
5 builder.append("StringBuilder objects are ")
6     .append("mutable");
```

La variable `message` sera d'abord assignée à un objet (ligne 1), puis à un autre objet (ligne 2). Après la deuxième affectation, le premier objet `String` sera perdu, mais restera en mémoire jusqu'à ce que le ramasse-miettes fasse son travail et libère la mémoire.

De son côté, la variable `builder` ne sera assignée qu'une seule fois (ligne 4), mais son contenu sera modifié après chaque appel de la méthode `append()`.

Les objets mutables ont des méthodes de modification, telles que : `set()`, `append()`, `add()`, etc.

#### Différences entre les objets mutables et immuables

| `Mutables` | `Immuables` |
|---|---|
| peuvent prendre n'importe quelle valeur ou n'importe quel état sans créer de nouvel objet. | ne peuvent pas être modifiés. |
| fournissent des méthodes de modification pour changer leur contenu. | ne fournissent aucune méthode de modification. |
| peuvent ou non être sûrs pour les threads. | sont sûrs pour les threads. |
| ne doivent pas être utilisés en tant que clés dans un `Map` : peuvent créer un comportement inattendu. | |
| ont tendance à utiliser moins de mémoire que les objets immuables. | |

#### Écrire des classes immuables

*Règles d'écriture des classes immuables :*

- La classe doit être déclarée `final` : elle ne doit pas être étendue.
- Les champs non transitoires doivent être `private` et `final` : ils ne peuvent pas être modifiés après l'initialisation.

#### Écrire des classes mutables

Il existe une règle simple pour écrire des classes mutables : elles doivent avoir des méthodes permettant de modifier les valeurs des champs, de façon directe ou indirecte.

### 2.2.2. Exercice : Écrire une classe mutable

Écrivez une classe mutable appelée `MutableIntWrapper`, qui implémente l'interface `MutableInt`, présentée ci-dessous. Cette classe doit implémenter des opérations arithmétiques (addition, soustraction, etc.), ainsi que des opérations d'incrémentation et de décrémentation.

Voici un exemple d'utilisation de la classe `MutableIntWrapper` :

```
package fr.unantes.sce.examples;

import fr.unantes.sce.mutable.MutableInt;
import fr.unantes.sce.mutable.MutableIntWrapper;

public class MutableIntExample {
    public static void main(String[] args) {
        MutableInt eight = new MutableIntWrapper(8);
        MutableInt fifteen = new MutableIntWrapper(15);
        eight.add(fifteen);
        assert eight.get() == 23;
    }
}
```

L'interface à implémenter est la suivante :

*Listing 1. L'interface* `MutableInt`

```
package fr.unantes.sce.mutable;

/**
 * An interface representing a mutable integer value.
 */
public interface MutableInt {

    /**
     * Gets the current value of the mutable integer.
     * @return The current value of the mutable integer.
     */
    int get();

    /**
     * Sets a new value for the mutable integer.
     * @param newValue The new value to set for the mutable integer.
     */
    void set(int newValue);

    /**
     * Adds the value of the other mutable integer to the current value of this mutable integer.
     * @param other The other mutable integer to add.
     * @return The current mutable integer after adding the other mutable integer.
     */
    MutableInt add(MutableInt other);
```

```
    /**
     * Subtracts the value of the other mutable integer from the current value of this mutable integer.
     * @param other The other mutable integer to subtract.
     * @return The current mutable integer after subtracting the other mutable integer.
     */
    MutableInt subtract(MutableInt other);

    /**
     * Multiplies the value of the other mutable integer with the current value of this mutable integer.
     * @param other The other mutable integer to multiply.
     * @return The current mutable integer after multiplying the other mutable integer.
     */
    MutableInt multiply(MutableInt other);

    /**
     * Divides the value of the current mutable integer by the value of the other mutable integer.
     * @param other The other mutable integer to divide by.
     * @return The current mutable integer after dividing by the other mutable integer.
     */
    MutableInt divide(MutableInt other);

    /**
     * Increments the value of the mutable integer by 1.
     */
    void increment();

    /**
     * Increments the value of the mutable integer by 1 and returns the updated value.
     * @return The updated value after incrementing.
     */
    int incrementAndGet();

    /**
     * Returns the current value of the mutable integer and then increments it by 1.
     * @return The current value of the mutable integer before incrementing.
     */
    int getAndIncrement();

    /**
     * Decrements the value of the mutable integer by 1.
     */
    void decrement();

    /**
     * Decrements the value of the mutable integer by 1 and returns the updated value.
     * @return The updated value after decrementing.
     */
    int decrementAndGet();

    /**
     * Returns the current value of the mutable integer and then decrements it by 1.
     * @return The current value of the mutable integer before decrementing.
     */
    int getAndDecrement();
}
```

### 2.2.3. Exercice : Écrire une classe immuable

1. Implémentez une classe nommée `ImmutableInt`, qui implémente des méthodes arithmétiques de base.
2. Utilisez l'interface `Immutable`, présentée ci-dessous, pour guider votre implémentation .
3. Une fois la classe implémentée, suggérez des solutions pour éviter :
    a. que les méthodes soient appelées avec un argument nul
    b. un dépassement de capacité du champ entier de la classe

*Listing 2. L'interface* `Immutable`

```java
package fr.unantes.sce.immutable;

public interface Immutable<T> {
    Immutable<T> add(Immutable<T> other);

    Immutable<T> subtract(Immutable<T> other);

    Immutable<T> multiply(Immutable<T> other);

    Immutable<T> divide(Immutable<T> other);

    T value();
}
```

Voici un exemple d'utilisation de la classe `ImmutableInt` :

*Listing 3. Exemple d'utilisation de la classe* `ImmutableInt`

```java
package fr.unantes.sce.examples;

import fr.unantes.sce.immutable.Immutable;
import fr.unantes.sce.immutable.ImmutableInt;

public class ImmutableIntExample {
    public static void main(String[] args) {
        Immutable<Integer> one = new ImmutableInt(1);
        Immutable<Integer> twenty = new ImmutableInt(20);
        Immutable<Integer> result = one.add(twenty);

        assert result.value() == 21;
        assert one.value() == 1;
    }
}
```

### 2.2.4. Exercice : Optimisation de l'instantiation de ImmutableInt

Modifier l'implémentation de la classe `ImutableInt` pour la faire mettre en cache les valeurs les plus utilisées (de 0 à 255) et empêcher l'instanciation de plus d'une instance par valeur.

Pour cela, vous allez appliquer le patron de conception Poids mouche [https://patterns.univ-nantes.io//flyweight.html].

*Étapes*

1. Créez un champ statique de type tableau d'entiers, de taille 256.
2. Initialisez ce champ avec des entiers immuables de 0 à 255.
3. Créez une méthode statique nommée `valueOf(int)`, qui vérifie si la valeur de l'argument est dans le cache. Si oui, elle retourne la valeur du cache. Si non, elle crée une nouvelle instance.
    a. La méthode `valueOf(int)` joue le rôle d'une méthode fabrique [https://patterns.univ-nantes.io//factory-method.html]
4. Rendez inaccessible le seul constructeur de la classe

## 2.3. Implémentation de la conception pilotée par le domaine

Pendant la construction du logiciel, le passage du modèles de conception au code source est essentielle. Dans ce chapitre, nous allons utiliser les patrons de conception, ainsi que des principes de conception introduits pendant les cours pour proposer l'implémentation des modèles de conception du domaine.

Un modèle de conception du domaine représente des objets du **`monde réel`** dans un domaine donnée, en opposition aux objets dits **`techniques`**, comme `Window`, `NetworkSocket`, `MouseEvent`, etc.

La figure Figure 1 est un modèle de conception du domaine pour le jeu de plateau Risk.

Commençons par la classe `Game`, qui est l'objet racine du modèle. Il est relié à la classe `Player`, puisqu'un joueur joue à une partie (*a player plays a game*).

Notez le triangle pour le sens de lecture de l'association. Notez aussi qu'une association possède au moins deux rôles : un à chaque extrémité. Chaque rôle a un nom et une multiplicité.

Selon la multiplicité du rôle de l'association entre ces deux classes, il peut y avoir entre deux et six (`[2..6]`) joueurs qui jouent à une partie. L'association entre `Player` et `Territory` précise qu'un joueur peut occuper entre 0 et 42 territoires.

L'association entre `Continent` et `Territory` représente la composition des territoires des différents continents, soit 4 (Amérique du sud et Océanie), 6 (Afrique), 7 (Europe), 9 (Amérique du nord) ou 12 (Asie).

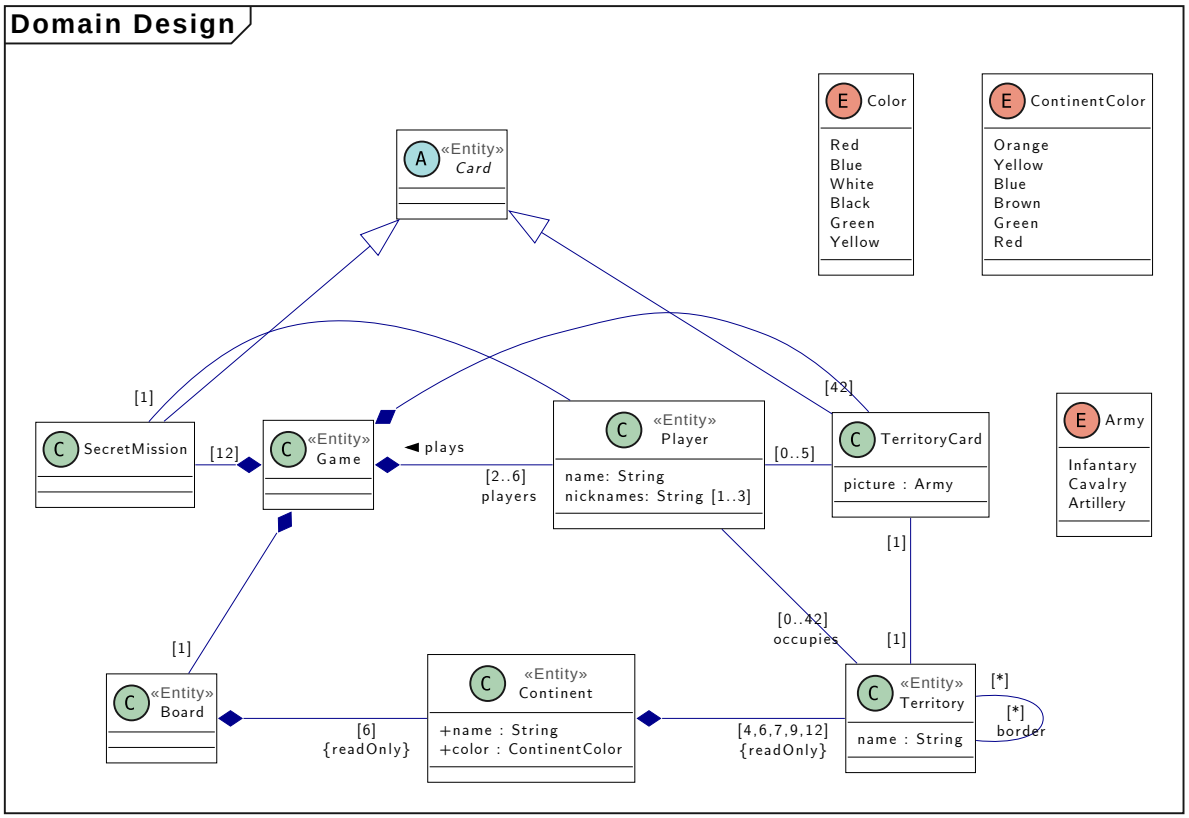



*Figure 1. Modèle de conception du jeu Risk*

### 2.3.1. Correspondance de types

Le langage UML dispose d'un ensemble restreint de types primitifs : `String`, `Integer`, `Real`, `Boolean` et `UnlimitedNatural`. Pour implémenter un modèle de conception UML dans un langage de programmation, nous devons établir une correspondance entre les types UML et ceux du langage cible (Java, Kotlin, C#, Python, etc.).

1. Tout d'abord, complétez le tableau suivant pour proposer une correspondance entre les types UML et ceux de Java, pour les attributs monovalués.

*Tableau 3. Types d'attributs monovalués*

| UML | Java |
|---|---|
| `String` | |
| `Integer` | |

| UML | Java |
|---|---|
| Real | |
| Boolean | |
| UnlimitedNatural | |

1. Maintenant, complétez le tableau suivant, pour proposer une table de correspondance similaire, mais pour des attributs à valeurs multiples, ou multivalués. Utilisez les classes et les interfaces fournies par le Java Collection Framework (JCF).

*Tableau 4. Types d'attributs multivalués*

| UML | Java |
|---|---|
| AnyType [*] {unique, ordered} | |
| AnyType [*] {unique, unordered} | |
| AnyType [*] {nonunique, ordered} | |
| AnyType [*] {nonunique, unordered} | |

### 2.3.2. Identifiants uniques

Un des principes de la programmation à objets est l'unicité des objets : chaque objet possède un identifiant unique qui le distingue des autres objets. En Java, comme dans d'autres langages à objets, c'est l'adresse mémoire d'un objet qui lui sert d'identifiant unique. Ainsi, si deux objets ont la même adresse mémoire, ils sont identiques.

Cette approche fonctionne correctement, pourvu que l'objet reste dans le même espace mémoire et dans le même contexte d'exécution. Si l'objet est enregistré sur une base de données ou envoyé, par le réseau, sur un autre espace mémoire, son adresse mémoire change et son identifiant n'est plus le même.

Contrairement aux objets techniques, dont le cycle de vie est lié à celui de l'exécution du logiciel, les objets du domaine doivent continuer à exister et rester cohérents d'une exécution à l'autre. Ils sont sujets à être stockés dans des bases de données et aussi envoyés à travers le réseau pour être utilisés par un autre logiciel, dans un espace mémoire différent.

En conséquence, il est naturel d'ajouter un identifiant unique aux objets du domaine. Différentes approches sont possibles : un nombre entier, une chaine de caractères, un identifiant unique universel, UUID [1], etc.

C'est ce que nous allons faire ici. Nous allons implémenter une classe nommée Any, qui servira de super classe commune à toutes les classes ayant besoin d'un identifiant. Voici la représentation de cette classe en UML :

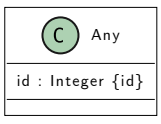



*Figure 2. La classe Any*

#### Exercice : Identifiants simples

1. Implémentez la classe Any en Java.
2. Utilisez un champ de type long pour représenter l'identifiant unique.
3. Implémentez un *accesseur* pour le champ id.

Un *Accesseur* ou *Getter* est une méthode qui renvoie la valeur d'un champ et ne fait rien de plus.

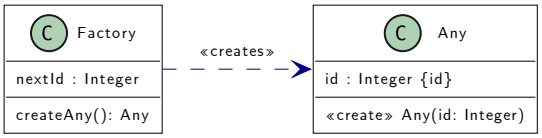
### Exercice : Utilisation d'une méthode fabrique

La solution précédente est très simple et efficace. Cependant, elle peut poser problème dans certaines situations, où l'on souhaite contrôler la valeur de l'identifiant.

Par exemple, supposez que vous souhaitez restaurer un objet stocké dans un fichier. Cet objet possède déjà un identifiant (par exemple, l'entier 42), comment créer une instance de `Any` et lui affecter un identifiant ?

On peut évidemment créer un deuxième constructeur et passer l'identifiant comme argument, mais dans ce cas, comment assurer l'unicité des identifiants ?

La solution est d'utiliser un patron de conception appelé « Méthode Fabrique ». Le diagramme de classes suivant décrit cette solution :




- Notez le lien unidirectionnel pointillé entre `Factory` et `Any`. Il représente une *dépendance* en UML. Ici, une dépendance de création/instantiation.
- Notez aussi que l'opération `Any()` est marquée avec `«create»`. Cela veut dire qu'il s'agit d'un constructeur.

1. Modifiez la classe `Any` de l'exercice précédent pour lui ajouter un constructeur contenant un seul paramètre, son identifiant.
2. Implémentez la classe `Factory`.

### Exercice : Utilisation d'un Singleton

La solution précédente pose aussi problème, car la classe Factory peut avoir plusieurs instances et par conséquent, plusieurs valeurs de `nextId`. Nous devons empêcher que la classe `Factory` soit instanciée plusieurs fois.

Pour cela, nous allons utiliser le patron de conception « Singleton », pour assurer que la classe `Factory` n'aura qu'une seule instance. Le diagramme de classes suivant décrit cette solution :


Le symbole `-` devant une propriété (attribut ou opération) indique que la visibilité est privée. Les symboles `+`, `~` et `#` représentent les visibilités publique, paquet et protégée, respectivement.

1. Modifiez la classe `Factory` de l'exercice précédent pour en faire un Singleton.

## 2.3.3. Implémentation d'Attributs Monovalués

Dans cet exercice, nous allons utiliser deux approches différentes pour implémenter des attributs :

1. Avec des accesseurs.
2. Avec des classes-enveloppes.

### Approche simple: utilisation d'accesseurs

Pour implémenter la classe `Continent` et ses attributs, utilisez la stratégie d'implémentation Getter/Setter introduite pendant les cours.

1. Tout d'abord, déclarez la classe Java et ses champs. N'oubliez pas d'en faire une sous-classe de la classe `Any`.
2. Déclarez ensuite les accesseurs et les modifieurs (getters et setters) des champs. N'oubliez pas de respecter la visibilité des attributs.
3. Finalement, décidez comment traiter les erreurs, par exemple les arguments dont la valeur est `null`.

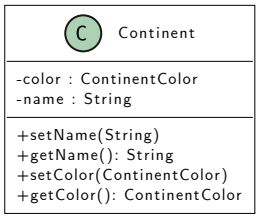



*Figure 3. Continent: Diagramme de classes d'implémentation*

### Avec des classes mutables

Maintenant, nous allons utiliser la stratégie d'implémentation "Classes-Enveloppes" introduite pendant les cours pour implémenter la classe `Continent`. L'idée ici c'est d'utiliser des classes mutables pour implémenter les attributs.

1. Commencez par implémenter une classe-enveloppe pour le type `String`
2. Cette classe doit être capable de :
   a. Stocker la valeur par défaut d'un attribut
   b. Dire si la valeur a été affectée (méthode booléenne `isSet()`)
   c. Revenir à la valeur initiale (méthode `reset()`)
3. Ensuite, ajoutez à la classe Continent un champ dont le type est `StringWrapper`, et un accesseur pour ce champ.

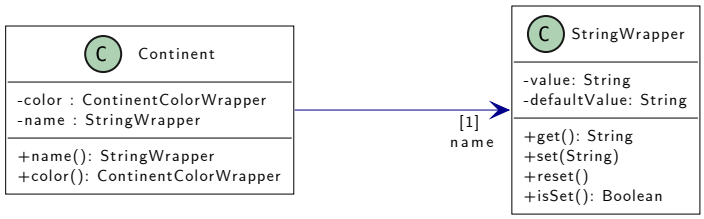



*Figure 4. Continent: Diagramme de classes d'implémentation*

### Comparaison

Analysez les deux solutions. A votre avis :

1. Laquelle demande le plus de lignes de code ?
2. Laquelle utilise le plus de mémoire ?
3. Laquelle est la plus simple à utiliser ?

### 2.3.4. Implémentation d'Attributs Multivalués

Les attributs multivalués sont des attributs dont la multiplicité maximale est supérieure à 1. Le nombre maximal d'éléments peut être non contraint (multiplicité `[*]`) ou contraint (multiplicité de `[n]`, où `n` est un nombre naturel supérieur à `1`).

La classe `Player` possède deux attributs, `name` et `nickname`. Le premier est monovalué (multiplicité `1`) et le deuxième, multivalué. Par la suite, vous allez implémenter ces deux attributs.

#### Exercice: utilisez le patron Décorateur pour implémenter une liste

Tout d'abord, appliquez le patron "Décorateur" pour implémenter une liste de taille maximale limitée. Comme le décorateur doit implémenter toutes les méthodes de l'interface `List`, vous aurez besoin de consulter la spécification [https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/List.html] de cette interface.

#### Exercice: implémentation de la classe Player

Maintenant, implémentez la classe `Player` et ses deux attributs. Utilisez des accesseurs pour ces deux attributs.

### 2.3.5. Implémentation d'Association Bidirectionnelles

Les associations bidirectionnelles sont des associations navigables dans les deux sens. Par exemple, la figure ci-dessous montre une association bidirectionnelle entre les classes `Player` et `Game`. Cette association est navigable à partir des instances de `Player` vers une instance de `Game`, mais aussi à partir des instances de `Game` vers des instances de `Player`.

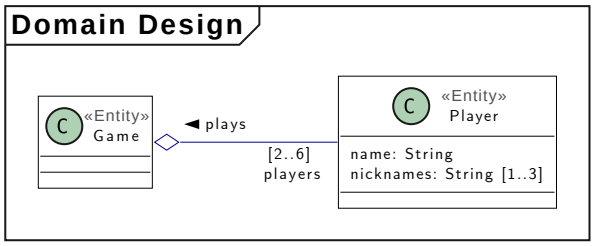



*Figure 5. Une association bidirectionnelle entre les classes `Game` et `Player`*

Les associations bidirectionnelles imposent une contrainte, appelée *intégrité référentielle* : si une instance `g` de `Game` est liée à une instance `p` de `Player`, alors cette instance `p` est aussi liée à l'instance `g`.

#### Implémentation de l'association entre `Game` et `Player`

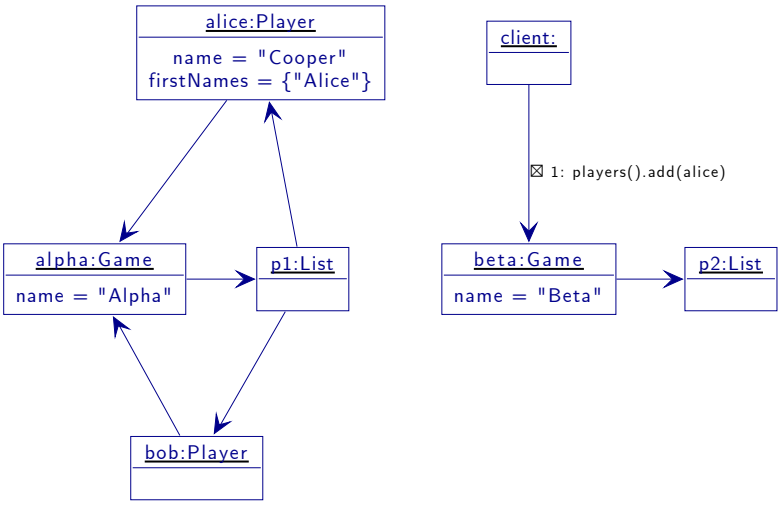
L'implémentation de cette association est plutôt complexe, car elle impose de maintenir la cohérence des deux côtés de l'association, ce qui implique l'envoi de plusieurs messages entre les différents objets.

Pour comprendre l'interaction entre les objets, vous allez utiliser des diagrammes de communication [https://www.uml-diagrams.org/communication-diagrams.html], pour illustrer les deux cas suivants :

1. L'ajout d'un joueur à une autre partie.
2. L'affection d'une partie à un joueur.

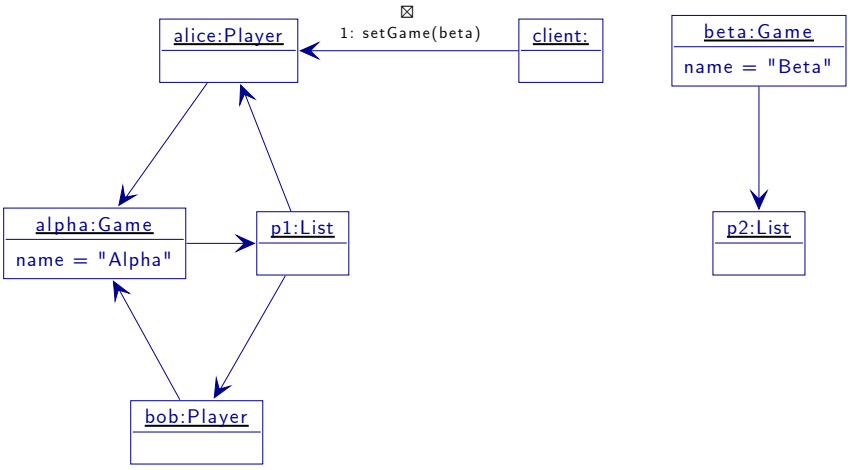
*Ajout d'un joueur à une autre partie*

1. Commencez par illustrer l'interaction lorsqu'une classe client demande à une instance de `Game`, `beta`, d'ajouter à la liste de joueurs une instance de `Player`, `alice`, qui participe déjà à une autre partie, `alpha`.
2. Les messages sont numérotés pour indiquer l'ordre d'envoi.
3. Utilisez des flèches pour indiquer le sens de l'envoi des messages.
4. Utilisez des croix pour indiquer la destruction d'un lien ou d'une instance, et des couleurs différentes pour indiquer les nouveaux liens ou instances créés.




*Affectation d'une partie à un joueur*

1. Maintenant, illustrez l'interaction lorsqu'une classe client demande à une instance de `Player` d'affecter une nouvelle instance de `Game`.




### Implémentation de l'association entre `SecretMission` et `Player`

Contrairement à l'association entre `Player` et `Game`, les deux rôles de l'association entre `SecretMission` et `Player` sont mono-valuées. La conséquence est que les liens entre les objets sont des références Java, qui peuvent être nulles : il faudra tester si le lien est nul avant de l'utiliser.

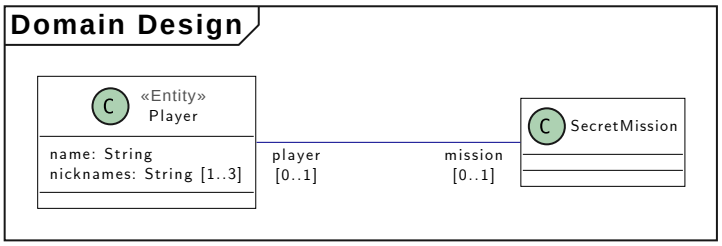



*Figure 6. L'association entre `Player` et `SecretMission`*

1. Utilisez le diagramme de communication ci-dessous pour illustrer l'affectation d'un joueur à une mission.


*Figure 7. Affectation d'un joueur à une mission*

1. Utilisez le diagramme de communication ci-dessous pour illustrer l'affectation d'une mission à un joueur.

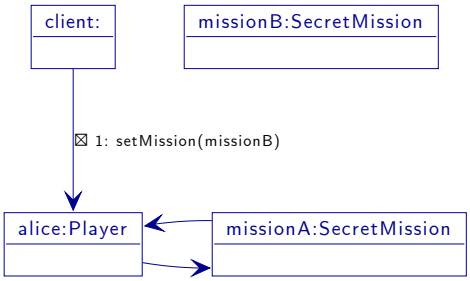



*Figure 8. Affectation d'une mission à un joueur*

## 2.4. Opérations de Refactoring

### 2.4.1. Opérations de refactoring sur les champs

Les opérations de *refactoring* sont des opérations de transformation de code source, qui ne modifient pas leur comportement visible. Pour assurer que ces opérations préservent le comportement, leur application est limitée par des *pré-conditions*.

Par exemple, pour renommer une classe, on doit d'abord vérifier qu'aucune autre classe porte son nouveau nom, car dans le cas contraire, la transformation introduirait une erreur.

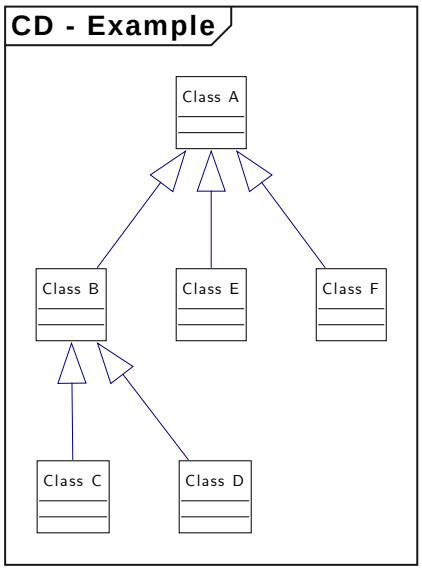



*Figure 9. Diagramme de classes générique*

Énumérez les préconditions nécessaires à l'exécution des opérations de *refactoring* suivantes. Utilisez le diagramme de classes présenté précédemment comme base de réflexion. Supposez que les opérations s'appliquent

à la classe `Class B`. Cette classe a une super-classe, `Class A`, deux sous-classes, `Class C` et `Class D`, et deux classes-sœurs, `Class E` et `Class F`.

Utilisez la sémantique du langage Java comme référence.

#### Ajout d'un champ d'instance à une classe

- `addInstanceField(c : Class, f: Field)`

#### Ajout d'un champ statique à une classe

- `addStaticField(c: Class, f: Field)`

#### Renommage d'un champ

- `rename(f: Field, name: String)`

#### Suppression d'un champ

- `delete(c: Class, f: Field)`

#### Transfert d'un champ à la classe mère

- `pullUp(c: Class, f: Field)`

#### Transfert d'un champ aux classes filles

- `pushDown(c: Class, f: Field)`

### 2.4.2. Opérations de refactoring sur les méthodes

#### Transfert d'une méthode à la classe mère

- `pullUpMethod(c: Class, m: Method)`

#### Transfert d'une méthode aux classes filles

- `pushDownMethod(c: Class, m: Method)`

#### Transfert d'une méthode à une autre classe

- `moveMethod(c: Class, m: Method, d: Class)`

## 2.5. Amélioration de code

### 2.5.1. Méthode composée

Le patron "Méthode composée" (Composed Method [https://c2.com/ppr/wiki/WikiPagesAboutRefactoring/ComposedMethod.html]) permet de produire des méthodes qui communiquent efficacement ce qu'elles font (leur comportement) et comment elles le font.

> Divide your program into methods that perform one identifiable task. Keep all of the operations in a method at the same level of abstraction.
>
> This will naturally result in programs with many small methods, each a few lines long.»
>
> — Kent Beck

Une méthode composée se compose d'appels à des méthodes bien nommées qui se situent toutes au même

niveau de détail. Suivez les instructions ci-dessous pour simplifier la méthode `add()`

*Listing 4. La classe ArrayList (Kerievsky:2004)*

```
1 package fr.unantes.sce.refactorings;
2
3 public class ArrayList {
4     private Object[] elements = new Object[10];
5     private boolean readOnly;
6     private int size = 0;
7
8     public void add(Object child) {
9         if (!readOnly) {
10            int newSize = size + 1;
11            if (newSize > elements.length) {
12                Object[] newElements = new Object[elements.length + 10];
13                System.arraycopy(elements, 0, newElements, 0, size);
14                elements = newElements;
15            }
16            elements[size] = child;
17            size++;
18        }
19    }
20
21    public void setReadOnly(boolean readOnly) {
22        this.readOnly = readOnly;
23    }
24
25    public boolean isReadOnly() {
26        return readOnly;
27    }
28 }
```

#### Exercice : gardes

1. Transformez la vérification de la variable `readOnly` en une garde.

#### Exercice : méthode addElement()

1. Appliquez l'opération de refactoring "Extraction de méthode" aux lignes 18-19 et créez la méthode `void addElement(object)`.

#### Exercice : variable explicative

1. Appliquez l'opération de refactoring "Extraire une constante" pour remplacer le nombre magique "10" en introduisant une "variable explicative" appelée `GROWTH_INCREMENT`.

#### Exercice méthode atCapacity()

1. Appliquez l'opération "Inline variable" à `newSize`, puis appliquez à nouveau l'opération "Extraire méthode" au code qui vérifie si le tableau d'éléments a atteint sa capacité et doit être agrandi, et créez la méthode `atCapacity()`.

#### Exercice méthode grow()

1. Enfin, appliquez l'opération de refactoring "Extraire une méthode" au code qui augmente la taille du tableau d'éléments, en créant la méthode `grow()`.

### 2.5.2. Enchaînement de constructeurs

En Java, les classes peuvent avoir plusieurs constructeurs : c'est normal, car il peut y avoir différentes façons d'instancier les objets d'une même classe. Cependant, lorsque le code est dupliqué dans deux constructeurs ou plus, des problèmes de maintenance se posent.

Considérons les trois constructeurs de la classe Loan [tds:amelioration:::Kerievsky:2004], présentés dans le Listing Listing 5, dont le code est dupliqué. Nous allons utiliser l'opération de refactoring appelée "Enchaînement de constructeurs", dont le but est de supprimer la duplication dans les constructeurs et les faire s'appeler entre eux. Tout d'abord, nous analysons ces constructeurs pour déterminer lequel est le "constructeur qui fait tout", celui qui gère tous les détails de la construction.

Il semble que ce soit le constructeur 3, puisque faire en sorte que les constructeurs 1 et 2 appellent le 3 peut être réalisé avec un minimum de travail.

*Listing 5. Les constructeurs de la classe Loan*

```
1 public Loan(float notional, float outstanding, int rating, Date expiry) {
2    this.strategy = new TermROC();
3    this.notional = notional;
4    this.outstanding = outstanding;
5    this.rating = rating;
6    this.expiry = expiry;
7 }
8
9 public Loan(float notional, float outstanding, int rating, Date expiry, Date maturity) {
10   this.strategy = new RevolvingTermROC();
11   this.notional = notional;
12   this.outstanding = outstanding;
13   this.rating = rating;
14   this.expiry = expiry;
15   this.maturity = maturity;
16 }
17
18 public Loan(CapitalStrategy strategy, float notional, float outstanding, int rating, Date expiry, Date
   maturity) {
19   this.strategy = strategy;
20   this.notional = notional;
21   this.outstanding = outstanding;
22   this.rating = rating;
23   this.expiry = expiry;
24   this.maturity = maturity;
25 }
```

**Exercice : simplifier le premier constructeur**

1. Modifiez le code du premier constructeur pour le faire appeler le troisième

**Exercice : simplifier le deuxième constructeur**

1. Maintenant, modifiez le deuxième constructeur pour le faire appeler le troisième.

### 2.5.3. Remplacer les constructeurs par des méthodes fabrique

L'objectif de l'opération de refactoring "Remplacer les constructeurs par des méthodes fabrique" est de remplacer les constructeurs par des méthodes fabrique qui clarifient l'intention du constructeur et renvoient des instances d'objets.

Les méthodes fabrique présentent au moins deux avantages sur les constructeurs :

1. Elles peuvent avoir des noms différents et donc communiquer l'intention de manière efficace.
2. Elles peuvent avoir le même nombre de paramètres.

Nous allons appliquer ce refactoring pour améliorer les constructeurs de la classe Loan. Considérons une autre version de la classe Loan, présentée dans le Listing Listing 6.

*Listing 6. Tous les constructeurs de la classe Loan*

```
1 public class Loan {
2
```

```
    double commitment;
    double outstanding;
    int riskRating;
    Date maturity;
    Date expiry;
    CapitalStrategy capitalStrategy;

    public Loan(double commitment, int riskRating, Date maturity) {
        this(commitment, 0.00, riskRating, maturity, null); // Term Loan
    }

    public Loan(double commitment, int riskRating, Date maturity, Date expiry) {
       this(commitment, 0.00, riskRating, maturity, expiry);
    }

    public Loan(double commitment, double outstanding, int riskRating, Date maturity, Date expiry) {
        this(null, commitment, outstanding, riskRating, maturity, expiry);
    }

    public Loan(CapitalStrategy capitalStrategy, double commitment, int riskRating, Date maturity, Date
expiry) {
        this(capitalStrategy, commitment, 0.00, riskRating, maturity, expiry);
    }

    public Loan(CapitalStrategy capitalStrategy, double commitment, double outstanding, int riskRating, Date
maturity,
            Date expiry) {
        this.commitment = commitment;
        this.outstanding = outstanding;
        this.riskRating = riskRating;
        this.maturity = maturity;
        this.expiry = expiry;
        this.capitalStrategy = capitalStrategy;

        if (capitalStrategy == null) {
            if (expiry == null)
                this.capitalStrategy = new CapitalStrategyTermLoan();
            else if (maturity == null)
                this.capitalStrategy = new CapitalStrategyRevolver();
            else
                this.capitalStrategy = new CapitalStrategyRCTL();
        }
    }
}
```

Pour appliquer ce refactoring, nous devons trouver un extrait de code qui appelle l'un de ces constructeurs. Par exemple, dans un cas de test :

```
public class CapitalCalculationTests {
    // ...
    public void testTermLoanNoPayments() {
      //...
      Loan termLoan = new Loan(commitment, riskRating, maturity);
      //...
   }
}
```

### Exercice : extraction d'une méthode statique

Tout d'abord, appliquez le refactoring "Extraire Méthode" à un **appel** du constructeur pour produire une méthode publique et statique appelée `createTermLoan()`.

### Exercice : déplacement de la méthode

Ensuite, appliquez le refactoring "Déplacer méthode" sur la méthode fabrique `createTermLoan()`, pour la déplacer vers la classe `Loan`.

#### Exercice : Inline Method

Après cette étape, nous devons trouver tous les appelants du constructeur et les mettre à jour pour appeler `createTermLoan()`. Puisque la méthode `createTermLoan()` est maintenant le seul appelant du constructeur, nous pouvons appliquer le refactoring "Inline Method" à cet appel de constructeur.

#### Exercice : les autres constructeurs

Répétez les mêmes opérations pour les autres constructeurs de la classe `Loan`, afin de créer des méthodes fabrique supplémentaires.

#### Exercice : dernière étape

Dernière étape, comme les constructeurs ne sont utilisés que par les méthodes fabrique, ils peuvent devenir privés.

#### Références


- [tds:amelioration:::Kerievsky:2004] Kerievsky, Joshua. Refactoring to patterns. Pearson Deutschland GmbH, 2005.


## 2.6. Patrons de conception

### 2.6.1. Double Dispatch

Considérez les classes `Graphic`, `Shape` et `Display`, dont le code est listé ci-dessous :

*Listing 7. Graphic.java*

```
package fr.unantes.sce.patterns.dispatch;

public class Graphic {
    @Override
    public String toString() {
        return "Graphic";
    }
}
```

*Listing 8. Shape.java*

```
package fr.unantes.sce.patterns.dispatch;

public class Shape extends Graphic {
    @Override
    public String toString() {
        return "Shape";
    }
}
```

*Listing 9. Display.java*

```
package fr.unantes.sce.patterns.dispatch;

public class Display {

    void display(Graphic graphic) {
        System.out.println("Displaying a Graphic");
    }

    void display(Shape shape) {
        System.out.println("Displaying a Shape");
    }
```

```
    public static void main(String[] args) {
        Graphic ga = new Graphic();
        Graphic gb = new Shape();
        Shape s = new Shape();
        Display d = new Display();

        d.display(ga);
        d.display(gb);
        d.display(s);
    }
}
```

> Qu'est-ce qui sera affiché lorsque la méthode `Display::main()` sera exécutée ? Pourquoi ?

L'expédition multiple (*Multiple Dispatch*) est une caractéristique propre à certains langages de programmation, tels que C#, Groovy ou Julia. Elle permet de déterminer la méthode à invoquer, en fonction du type dynamique (à l'exécution) de l'objet récepteur et de ceux des arguments.

La double expédition est une forme spéciale de multiple expédition, qui détermine la méthode à invoquer en fonction des types dynamiques du récepteur et d'un argument. Cependant, la plupart des langages à objets typés, tels que Java et C++, déterminent la méthode à invoquer en fonction du type **`dynamique`** du récepteur et des types **`statiques`** des arguments.

Supposons que vous deviez gérer des portefeuilles contenant de l'argent dans différentes devises : Euro (€), Dollar ($), Livre (£), etc. Ajouter de l'argent à un portefeuille est un problème, car vous ne pouvez additionner que de l'argent d'une même devise.

Lorsque vous additionnez de l'argent provenant de différentes devises, vous devez créer un porte-monnaie d'argent contenant les deux montants. Par exemple, 15€ + 5€ = 20€, mais 15€ + 5£ = {15€, 5£}.

Vous pouvez également additionner de l'argent à un porte-monnaie et additionner deux porte-monnaies. Par exemple, 15€ + {15€, 5£} = {30€, 5£} et {5€, 5£} + {10€, 10£} = {15€, 15£}.

En résumé, il y a 3 additions différentes :

1. Argent + Argent
2. Argent + Porte-monnaie (ou vice-versa)
3. Porte-monnaie + Porte-monnaie

Par conséquent, la complexité de la mise en œuvre concerne la gestion des différentes combinaisons possibles d'argent et de porte-monnaie.

Une manière élégante de résoudre ce problème consiste à utiliser le patron de conception *Double Dispatch*. L'idée derrière ce patron est d'utiliser un appel supplémentaire pour découvrir le type dynamique de l'argument. Par exemple, considérons l'interface `Money` et ses implémentations, `MoneyBag` et `SingleMoney` :

*Listing 10. L'interface Money*

```
interface Money {
    Money add(Money amount);
}

class MoneyBag implements Money {
    public Money add(Money amount) {
    //...
    }
}
```

Les implémentations de la méthode `add()` n'ont pas accès au type dynamique du paramètre `amount`, qui peut être soit `SingleMoney` soit `MoneyBag`. Une manière simple de trouver le type dynamique de l'argument est d'utiliser l'opérateur `instanceof`, mais cette solution est rarement une bonne idée.

**Le patron Double Dispatch à la rescousse** : Une autre solution consiste à utiliser l'argument comme récepteur d'un nouvel appel de méthode et laisser le polymorphisme faire sa magie :

```
class MoneyBag implements Money {}
    public Money add(Money amount) {
        amount.addMoneyBag(this);
    }
}
```

La solution est simple, puisque nous connaissons le type dynamique de l'objet courant (`this`), `MoneyBag`. Nous utilisons un deuxième appel de méthode, cette fois en utilisant l'argument `amount` comme récepteur de l'appel.

> Quelle implémentation de la méthode `addMoneyBag()` sera invoquée, `SingleMoney::addMoneyBag()` ou `MoneyBag::addMoneyBag()` ?

La réponse est simple : Java utilise le type dynamique du récepteur, si `amount` est une instance de `SingleMoney`, il invoquera la première, et si `amount` est une instance de `SingleMoney`, il appellera la seconde.

Néanmoins, cette solution présente un **inconvénient important**, l'interface `Money` doit spécifier les méthodes `addSingleMoney(SingleMoney)` et `addMoneyBag(MoneyBag)`. En d'autres termes, l'interface dépend de ses implémentations, ce qui est une **mauvaise pratique**.

#### Exercice : L'interface Money

1. Créez une interface nommée `Money`, contenant 3 méthodes : `add()`, `addSingleMoney()` et `addMoneyBag()`.

#### Exercice : La classe SingleMoney

Ensuite, créez une classe **immuable** nommée `SingleMoney` qui implémente l'interface `Money` et ses trois méthodes. Cette classe doit posséder deux champs, représentant leur montant ainsi que le nom de la devise utilisé.

1. Pour commencer, ajoutez deux constructeurs à cette classe, permettant de l'instancier à partir d'un montant et d'une devise et aussi à partir de deux instances de la classe `SingleMoney`.
2. Ensuite, implémentez les trois méthodes d'addition.

#### Exercice : La classe MoneyBag

Enfin, créez une classe **immuable** nommée `MoneyBag` qui implémente l'interface `Money` et ses trois méthodes. Cette classe doit posséder un champ, représentant un multi-ensemble de monnaies simples.

1. Pour commencer, ajoutez trois constructeurs à cette classe, permettant de l'instancier à partir de
   a. Deux monnaies simples
   b. Un porte-monnaie et une monnaie simple
   c. Deux portemonnaies
2. Ensuite, implémentez les trois méthodes d'addition.

---

### 2.6.2. État

La machine d'états présentée ci-dessous est attachée à la classe `Connection`. Ce diagramme nous permet de savoir que la classe possède **au moins** cinq opérations : `connect()`, `disconnect()`, `setNotAvailable()`, `setAvailable()` et `setFreeForChat()`. Chacune de ces opérations résultent en un changement d'état.

Dans cet exercice, nous allons utiliser le patron de conception "État" pour implémenter la classe `Connection`.

Nous allons implémenter seulement le comportement correspondant aux changements d'état.

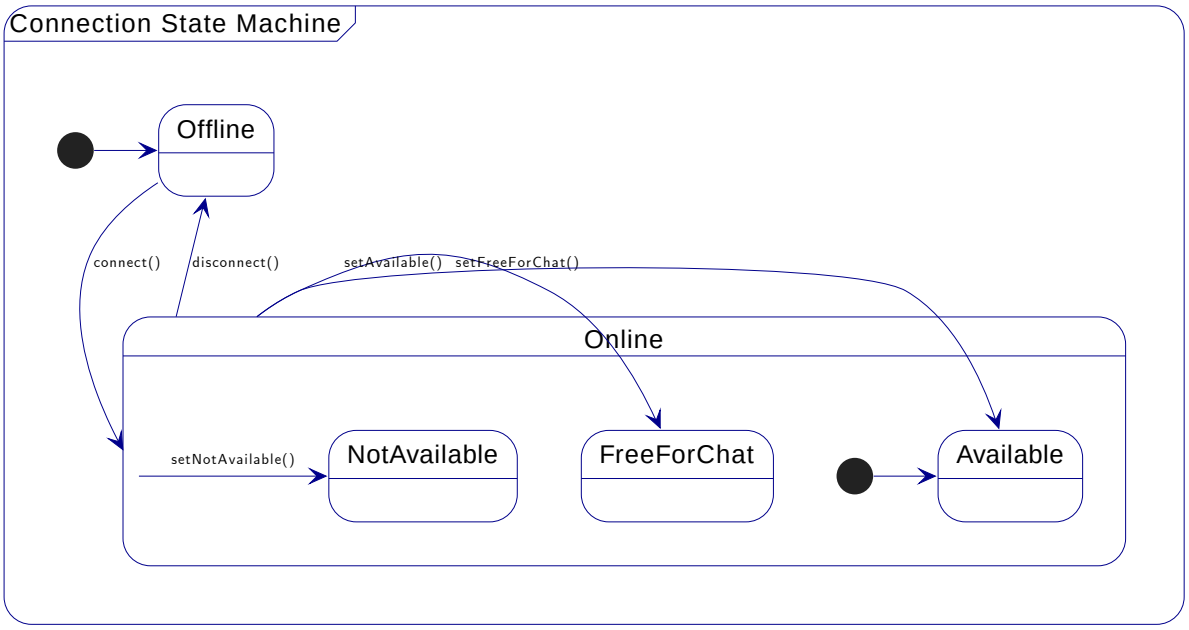



*Figure 10. Diagramme État-Transition de la classe Connection*

**Exercice : Diagramme de classes**

1. Pour commencer, dessinez un diagramme de classes UML représentant votre solution
2. Implémentez la classe `Connection` et ses méthodes.
3. Maintenant, proposez une interface pour la classe `ConnectionState`. Elle doit contenir toutes les méthodes qui dépendent de l'état de la connexion.
4. Enfin, implémentez les classes état, ainsi que leurs méthodes.

### 2.6.3. Itérateur

En Java, la façon traditionnelle de parcourir une collection d'objets est d'utiliser des **`itérateurs externes`**. On les appelle *externes*, car ils ne sont pas contrôlés par la collection, mais par la classe cliente souhaitant la parcourir. Ils sont aussi *robustes*, parce qu'ils sont capables de détecter des modifications dans la collection durant le parcours et d'arrêter l'itération.

Une autre façon de parcourir les collections, c'est d'utiliser les **`itérateurs internes`**. Dans ce cas, c'est la collection elle-même qui contrôle l'itération. Par exemple, l'interface `Collection` propose les méthodes `removeIf()` et `forEach()`, qui utilisent des itérateurs internes. La première supprime tous les éléments de la collection qui satisfont à un prédicat donné. La deuxième effectue une *action* pour chaque élément de la collection, jusqu'à ce que tous les éléments aient été traités (ou que l'action lance une exception).

**Exercice : itérateurs internes**

Nous souhaitons ajouter 4 méthodes à l'interface `Collection` de Java :

- `select()` : sélectionne les élements de la collection qui satisfont à un prédicat passé en paramètre ;
- `forAll()` : retourne vrai si tous les éléments satisfont à un prédicat donné, faux autrement ;
- `forEach()` : exécute une action pour chaque élément de la collection ;
- `collect()` : exécute une action pour chaque élément de la collection et ajoute le résultat à une nouvelle collection, de même taille, mais potentiellement de type différent.

*Travail à faire*

1. Proposez la signature des 4 méthodes ;
2. Ensuite, proposez une façon d'étendre les classes qui implémentent l'interface `Collection` avec ces méthodes ;
3. Enfin, implémentez les 4 méthodes et donnez des exemples d'utilisation de ces méthodes.

[1] Universally Unique IDentifier est en informatique un système permettant à des systèmes distribués d'identifier de façon unique une information sans coordination centrale importante

# Chapitre 3. Exercices de TP

## 3.1. Travaux pratiques

### 3.1.1. Séances

| Séance | Sujet |
|---|---|
| 1 et 2 | 1. Section 3.2<br>2. Section 3.3<br>3. Section 3.4 |
| 2 | 1. La classe `Interval` |
| 3 | 1. Implémentation d'attributs UML |
| 4 et 5 | 1. Implémentation d'associations UML |
| 6 à 9 | Projet |

## 3.2. Préparation

### 3.2.1. Environnement de développement

- Pour la programmation Java, utilisez IntelliJ IDEA [https://www.jetbrains.com/idea/], qui est installé sur les machines du CIE.

### 3.2.2. Maven à la Faculté des Sciences

- La configuration du pare-feu de réseau de la Faculté des Sciences pose quelques problèmes au fonctionnement de Maven.
- Le premier c'est que tout accès au réseau doit passer par le serveur proxy et Maven ne le sait pas.
- Pour le configurer, vous devez créer un fichier appelé `settings.xml` dans le répertoire `~/.m2`:

```
touch ~/.m2/settings.xml
vim ~/.m2/settings.xml
```

Ajoutez ensuite les lignes suivantes:

```
<settings>
  <proxies>
    <proxy>
      <active>true</active>
      <protocol>http</protocol>
      <host>proxy.ensinfo.sciences.univ-nantes.prive</host>
      <port>3128</port>
    </proxy>
  </proxies>
</settings>
```

En complément, sur certaines machines, Maven n'arrive pas à retrouver son dossier d'installation et ne marche pas correctement. Dans ce cas, vous devez affecter la variable système `M2_HOME`:

```
export M2_HOME=/usr/local/opt/maven/
```

## 3.3. Tutoriel Maven

### 3.3.1. Introduction

- Après avoir développé avec succès une Calculatrice en Java, vous souhaitez partager votre succès rayonnant avec tout le monde.
- Et vous ne souhaitez pas vous arrêter là, vous souhaitez aussi que d'autres développeurs participent à votre projet et contribuent à son amélioration.
- Vous avez de la chance, la fondation Apache a créé Maven [https://maven.apache.org], un outil de gestion de la construction de logiciels.
- Maven vous aidera à compiler et tester votre logiciel, mais aussi à le distribuer et à le rendre simple à comprendre par d'autres développeurs.

#### Convention avant la configuration

- L'organisation d'un projet Maven respecte toujours la même convention : l'emplacement du code source, des tests, des ressources, etc. est toujours le même.
- Cette convention a deux conséquences :
  a. La prise en main d'un nouveau projet est plus simple : le nouveau développeur connaît d'emblée la structure du projet et ne perdra pas de temps à chercher l'emplacement des fichiers sources, des tests, etc.
  b. Il n'est pas nécessaire de configurer Maven avant de l'utiliser. Il trouvera tout seul le code source et les tests unitaires de votre projet.
- Les conventions proposées par Maven sont très complètes. En voici quelques exemples :

| Dossier | Contenu |
|---|---|
| `src/main/java` | Fichiers source Java |
| `src/test/java` | Tests unitaires JUnit ou TestNG. |
| `src/test/resources` | Ressources utilisées pendant les tests. |
| `target` | Fichiers générés pendant la construction du logiciel. |

#### Identification et création d'un projet

- Tout projet Maven possède un identifiant unique, composé de trois éléments :
  a. un identifiant de groupe,
  b. un identifiant de l'artefact et
  c. sa version.
- Cet identifiant permet de créer des dépendances entre projets, qui sont gérées par Maven.
- Par exemple, si votre projet utilise JUnit, vous n'avez pas besoin de télécharger une archive Java et de l'ajouter au projet, mais simplement de déclarer que votre projet a besoin de JUnit, identifié par le groupe `junit`, l'artefact `junit` et la version `4.12`.
- Ainsi, avant de créer votre projet, vous devez l'identifier. L'identifiant est souvent celui de l'organisation à laquelle vous appartenez.
- Par exemple, `fr.unantes`. Ensuite, l'identifiant de l'artefact est son nom, par exemple, `calculatrice`.

Une fois que votre projet est identifié, vous pouvez le créer par la commande Unix suivante :

```
 mvn archetype:generate -DarchetypeVersion=1.4 -DgroupId=fr.unantes -DartifactId
=calculatrice -Dversion=1.0-SNAPSHOT -DinteractiveMode=false
```

- Cette commande créera un dossier appelé `calculatrice`, contenant différents dossiers vides et un fichier XML de configuration et d'information du projet, nommé `pom.xml`, appelé couramment un fichier POM.
- Notez que cette commande spécifie aussi un «archétype», c'est à dire, la structure de projet à être construite.
- L'archétype `generate` est un des plus simples et vous posera quelques questions durant la création.
- Le contenu de ce fichier doit ressembler à ceci :

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>fr.unantes</groupId>
  <artifactId>calculatrice</artifactId>
  <version>1.0-SNAPSHOT</version>

  <name>calculatrice</name>
  <!-- FIXME change it to the project's website -->
  <url>http://www.example.com</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.7</maven.compiler.source>
    <maven.compiler.target>1.7</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <pluginManagement><!-- lock down plugins versions to avoid using Maven defaults (may be moved to parent pom)
-->
      <plugins>
        <!-- clean lifecycle, see https://maven.apache.org/ref/current/maven-
core/lifecycles.html#clean_Lifecycle -->
        <plugin>
          <artifactId>maven-clean-plugin</artifactId>
          <version>3.1.0</version>
        </plugin>
        <!-- default lifecycle, jar packaging: see https://maven.apache.org/ref/current/maven-core/default-
bindings.html#Plugin_bindings_for_jar_packaging -->
        <plugin>
          <artifactId>maven-resources-plugin</artifactId>
          <version>3.0.2</version>
        </plugin>
        <plugin>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>3.8.0</version>
        </plugin>
        <plugin>
          <artifactId>maven-surefire-plugin</artifactId>
          <version>2.22.1</version>
        </plugin>
        <plugin>
          <artifactId>maven-jar-plugin</artifactId>
          <version>3.0.2</version>
        </plugin>
        <plugin>
          <artifactId>maven-install-plugin</artifactId>
          <version>2.5.2</version>
        </plugin>
        <plugin>
          <artifactId>maven-deploy-plugin</artifactId>
```

```
          <version>2.8.2</version>
        </plugin>
        <!-- site lifecycle, see https://maven.apache.org/ref/current/maven-core/lifecycles.html#site_Lifecycle
-->
        <plugin>
          <artifactId>maven-site-plugin</artifactId>
          <version>3.7.1</version>
        </plugin>
        <plugin>
          <artifactId>maven-project-info-reports-plugin</artifactId>
          <version>3.0.0</version>
        </plugin>
      </plugins>
    </pluginManagement>
  </build>
</project>
```

- Comme vous pouvez le constater, ce fichier contient l'identifiant du projet, une propriété (le type d'encodage) et une seule dépendance, vers la version `4.11` de JUnit.
- Le fichier POM contient aussi une liste de plugins sous la balise `<pluginManagement>`.
  - C'est une manière d'indiquer à Maven les versions souhaitées des ses plugins et éviter qu'il utilise des versions très anciennes.
- Il existe d'autres moyens de créer un projet Maven, par exemple, directement dans Netbeans, VS Code ou dans IntelliJ IDEA.
- Il existe aussi d'autres archétypes, proposés par la fondation Apache ou par d'autres contributeurs.
- Vous pouvez aussi modifier un projet existant et le transformer en projet Maven.
  - Pour cela, il suffit de créer ou copier un fichier POM et de respecter la structure de dossiers de Maven.

### Dépendances

- Comme expliqué dans l'introduction, un projet Maven ne contient pas les archives Java des artefacts qu'il utilise.
- Par exemple, la dépendance à Junit 4.11, ajouté par défaut dans le fichier POM, dit à Maven de télécharger l'archive dans un référentiel local.
  - Plus précisément, dans le dossier `~/.m2/repository/junit/junit/4.11/`.
  - Cet artefact n'est pas spécifique à votre projet, il peut être utilisé par d'autres projets Maven.
- Si vous souhaitez utiliser une version plus récente de JUnit, il suffit de mettre à jour cette dépendance :

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>5.11.3</version>
    <scope>test</scope>
</dependency>
```

- Maven se chargera de télécharger cette archive lorsque ça sera nécessaire. Sauf si elle a déjà été téléchargée.
- En fait, avant de télécharger une dépendance, Maven vérifie si elle n'est pas présente dans le référentiel local et si c'est le cas, il ne la téléchargera pas une deuxième fois.
- Notez que la dépendance spécifie aussi la portée (*scope*) de l'artefact.
- Dans ce cas précis, il n'est utilisé que pendant les tests unitaires :
  - En termes Java, il n'est pas ajouté à la variable `CLASSPATH` pendant la compilation, mais le sera pendant les tests unitaires.
- Si vous souhaitez utiliser d'autres artefacts, il suffit de les ajouter au fichier POM.
- Si par exemple, si vous souhaitez utiliser les annotations proposées par la JSR 308 [http://www.oracle.com/technetwork/articles/java/ma14-architect-annotations-2177655.html], il suffit de chercher son identifiant (utilisez par

exemple le site MVNRepository [https://mvnrepository.com/]) et de l'ajouter au fichier POM :

```
<dependency>
    <groupId>net.java.loci</groupId>
    <artifactId>jsr308-all</artifactId>
    <version>1.1.2</version>
</dependency>
```

### Production

- Comme dans tout outil de production depuis plus de 40 ans (Make a été créé en 1977), la production suit une séquence spécifique de phases.
- Par exemple, pour créer une archive Java, vous devez d'abord tester votre code. Et pour tester le code, vous devez d'abord le compiler.
- Maven n'échappe pas à la règle, la production suit une séquence précise de phases, appelées cycle de vie [https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html].
- Au début, vous devez connaître 3 phases :

| Phase | Description |
|---|---|
| `compile` | Compile le code Java, après avoir téléchargé les dépendances nécessaires. |
| `test` | Exécute les tests unitaires du projet, après l'avoir compilé. |
| `package` | Crée une archive Java du projet, après l'avoir testé. |

- D'autres phases sont disponibles, par exemple pour nettoyer le dossier `target` (`mvn clean`) ou générer le site web du projet (`mvn site`).

### Configuration des plugins

- Chaque phase de production (compilation, test, etc.) est associée à un plugin.
- Dans la plupart des cas, vous n'avez pas besoin de les configurer, les valeurs par défaut sont suffisantes et marchent dans la plupart des cas.
- Dans les autres cas, il est possible de configurer précisément chaque plugin à l'intérieur du fichier POM.

Par exemple, si vous souhaitez que votre code soit compilé par la version 9 de Java (en ligne de commande, `javac -source 1.9 -target 1.9`), vous pouvez ajouter les balises suivantes au fichier POM :

```
<project>
  [...]
  <build>
    [...]
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.7.0</version>
        <configuration>
          <source>1.9</source>
          <target>1.9</target>
        </configuration>
      </plugin>
    </plugins>
    [...]
  </build>
  [...]
</project>
```

### 3.3.2. Maven, La suite

Maintenant que vous avez créé votre premier projet Maven, vous vous voyez confrontés à un scénario un peu différent. Deux de vos amis ont développé une application capable de gérer des agendas et de les synchroniser avec *Google Calendar*.

Vous trouvez leur application géniale, mais vous ne savez pas comment l'intégrer à votre projet.

En effet, bien que sympathiques, vos amis sont fainéants : ils ne vous ont laissé que le code source [https://gitlab.univ-nantes.fr/sunye-g/dates] et un fichier `README` succinct.

Deux options se présentent à vous :

1. copier leur code dans votre projet ou
2. le gérer comme un projet indépendant et utiliser Maven pour le transformer en un artefact réutilisable.

Comme vous faites souvent de bons choix (et accessoirement, vous avez remarqué que ceci est un tutoriel sur Maven et non sur la copie de fichiers), vous avez opté pour la deuxième option.

Vous allez élitiser [1] leur projet.

Avant de commencer, créez une copie locale du projet :

```
git clone https://gitlab.univ-nantes.fr/sunye-g/dates.git && cd dates
```

#### Configuration du projet

- Notre première étape sera de créer le modèle objet du projet, le fichier `pom.xml`.
- Cette fois, nous n'allons pas utiliser un archétype pour créer le squelette du projet, car le projet existe déjà, nous allons tout simplement créer le fichier manuellement.
- Bien évidemment, vous pouvez copier le modèle d'un autre projet et modifier ensuite les identifiants de groupe et d'artefact.

1. Créez un fichier appelé `pom.xml` à la racine du projet **`Dates`** et ajoutez-y les balises XML suivantes :

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://maven.apache.org/POM/4.0.0"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>fr.univnantes.student</groupId>
    <artifactId>dates</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>
    </properties>
</project>
```

Vous pouvez désormais compiler le projet avec Maven :

```
mvn compile
```

- Comme vous pouvez constater, la compilation est un succès.
- Cela n'est pas dû à la qualité du code de vos amis, mais tout simplement parce que le compilateur Java n'a pas trouvé de code à compiler.

### Réorganisation des dossiers

- Il ne s'agit pas d'une erreur, dans le cadre d'un projet Maven, le compilateur Java cherche le code source dans le dossier `src/main/java`, exclusivement.

Il est bien sûr possible de dire à Maven de chercher le code source ailleurs, mais ce choix va à l'encontre d'un des principes de base de Maven :

> La **`convention`** plutôt que la **`configuration`**.

- Nous allons donc respecter ce principe et déplacer le code source à sa place appropriée.

Utilisez les commandes *shell* suivantes pour réorganiser le projet :

```
mkdir -p src/main/java
mv src/univ src/main/java
```

Compilez à nouveau le projet avec Maven :

```
mvn compile
```

- Cette fois, le compilateur trouve plusieurs erreurs, ce qui est normal.
- Il a trouvé le code source, mais n'arrive pas à trouver les artefacts logiciels utilisés par ce code.

### Configuration des dépendances

- Le fichier `README.adoc` nous donne une piste importante : le code source utilise les artefacts **`MiG Layout`** et **`Google Data`**.
- La bonne nouvelle c'est que ces deux artefacts sont disponibles sur Maven Central.
- Cependant, pour les retrouver Maven aura besoin de leurs identifiants, composés de trois parties :
  a. leurs identifiants de groupe (`groupId`),
  b. leurs identifiants d'artefact (`artifactId`) et
  c. leurs versions (`version`).

- Utilisez le site MVNRepository [https://mvnrepository.com] pour retrouver les dépendances et ajoutez-les au fichier `pom.xml`.

Pour gagner du temps, utilisez les mots-clés `com.google.gdata` et `miglayout`.

- A la fin, vous devez avoir une balise appelée `<dependencies>`, qui a comme mère directe la balise `<project>` et comme filles, deux balises `<dependency>` :

```
<project>
```

```
    <!-- ... -->
    <dependencies>
        <dependency>
            <groupId>com.google.gdata</groupId>
            <artifactId>core</artifactId>
            <version>1.47.1</version>
        </dependency>
        <dependency>
            <groupId>com.miglayout</groupId>
            <artifactId>miglayout</artifactId>
            <version>3.7.4</version>
        </dependency>
    </dependencies>
    <!-- ... -->
</project>
```

Recompilez le projet à l'aide de Maven avec `mvn compile`.

- Cette fois, la compilation réussit avec quelques avertissements : le code utilise des méthodes qui ont été dépréciées.
- Ces avertissements sont importants, car ils aideront les développeurs à mettre à jour le code pour garder la compatibilité avec les versions futures des artefacts utilisés.
- Nous allons exploiter cette information plus tard.

### Documentation du projet

- Maintenant que le projet compile (presque) correctement, nous allons générer un site web contenant la documentation technique du projet.
- Par défaut, Maven produit un rapport assez riche, que nous allons améliorer par la suite.
- Nous allons commencer par configurer le *plugin* responsable de la phase `site` de la construction.

Ajoutez la balise suivante au fichier `pom.xml` :

```
<project>
    <!-- (...) -->
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-site-plugin</artifactId>
                <version>3.12.1</version>
            </plugin>
        </plugins>
    </build>
</project>
```

- En réalité, on ne configure pas vraiment le *plug-in* `maven-site-plugin` ici, on se contente de préciser à Maven qu'il doit utiliser une version récente du *plug-in* : en l'occurrence, la `3.12.1`.
- Cette précision est nécessaire, car elle empêchera Maven d'utiliser une version plus ancienne de ce *plug-in*.

Exécutez maintenant la phase `site` grâce à la commande suivante :

```
mvn site
```

- Le *plug-in* génère les pages HTML dans le dossier `target/site/`.

Utilisez un navigateur pour ouvrir le fichier `target/site/index.html` et découvrir le rapport généré.

- Parmi d'autres informations, vous trouverez les *plug-ins* utilisés par Maven, ainsi que les dépendances (directes et transitives) du projet.

### Génération de la Javadoc

- Vous avez probablement remarqué qu'à la racine du projet, il y a un dossier appelé `javadoc` qui contient la Javadoc générée par vos amis.
- Ils ont pensé que c'était une bonne idée d'ajouter **`tous`** les fichiers générés au gestionnaire de versions, plutôt qu'un simple script capable de les générer.
- Ce n'est pas le cas.
- L'idée est mauvaise, mais vous pouvez les remercier : comme vous n'aurez pas le temps de faire toutes les erreurs possibles en une seule vie, vous pouvez apprendre des erreurs des autres.
- Ou, pour citer Confucius :

> L'homme sage apprend de ses erreurs, l'homme plus sage apprend des erreurs des autres.
>
> — Confucius

- Vous avez compris, vous pouvez utiliser Maven pour générer automatiquement la Javadoc pendant le processus de *build*.
- En effet, Maven est capable d'intégrer d'autres rapports au site généré et c'est ce que nous allons faire avec la Javadoc.

Ajoutez les balises suivantes au fichier `pom.xml` :

```
<project>
    <!-- (...) -->
    <reporting>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-javadoc-plugin</artifactId>
                <version>3.5.0</version>
                <configuration>
                    <additionalJOption>-Xdoclint:none</additionalJOption>
                </configuration>
            </plugin>
        </plugins>
    </reporting>
</project>
```

- La balise `<reporting>` nous permet de configurer les rapports qui apparaîtront dans le site web généré, plus précisément dans la section **`Project Reports`**.
- Le *plug-in* `maven-javadoc-plugin` permet plusieurs options [https://maven.apache.org/plugins/maven-javadoc-plugin/javadoc-mojo.html].
- Ici, on lui dit de ne pas arrêter la génération lorsqu'il rencontre des erreurs de syntaxe dans les commentaires.
- Bien évidemment, il n'est pas nécessaire de dire que vos amis apprécient particulièrement ce type d'erreur.

Exécutez à nouveau la commande `mvn site` et rechargez la page `index.html`.

- Le site généré contient désormais la Javadoc.
- Remarquez que la Javadoc générée contient des liens hypertexte vers les classes de la JDK, comme `Object` ou `ActionListener`.

### Évaluation de la qualité du code source

- La Javadoc est certes une bonne source d'informations pour ceux qui souhaitent utiliser le code fait par vos amis, mais elle ne donne aucune information sur la qualité de ce code.
- En effet, toujours fainéants, vos amis n'ont écrit aucun test, ni unitaire, ni d'intégration.
- Vous ne savez ni si le code est fiable ni s'il est maintenable.
- Heureusement pour vous, plusieurs outils d'analyse statique de code Java existent et pourront vous aider à estimer la qualité du code source et à l'améliorer.
- Par exemple :
- PMD [https://pmd.github.io],
- Checkstyle [https://checkstyle.sourceforge.io],
- Spotbugs [https://spotbugs.github.io],
- Error Prone [http://errorprone.info],
- The Checker Framework [https://checkerframework.org/]
- BlockHound [https://github.com/reactor/BlockHound] et quelques autres.
- Ces outils peuvent s'intégrer aux rapports produits par Maven, grâce à des *plug-ins* spécifiques.
- Pour évaluer le code de vos amis, nous allons en utiliser deux, qui permettent à Maven d'utiliser PMD et Spotbugs.

Ajoutez les balises suivantes à l'intérieur de la balise `<reporting>` de votre fichier `pom.xml` :

```
<reporting>
    <plugins>
        <!-- (...) -->
        <plugin>
            <groupId>com.github.spotbugs</groupId>
            <artifactId>spotbugs-maven-plugin</artifactId>
            <version>4.8.3.0</version>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-pmd-plugin</artifactId>
            <version>3.21.2</version>
        </plugin>
    </plugins>
</reporting>
```

Exécutez à nouveau la commande `mvn site` et rechargez la page `index.html`.

- Trois nouveaux rapports s'affichent à côté de celui de la Javadoc : SpotBugs, PMD et CPD.
- Ce dernier, *Copy Paste Detector*, affiche les morceaux de code dupliqués du projet.
- Comme vous pouvez constater, le format des rapports est similaire, il s'agit d'un tableau contenant le nom de la règle (ou *bug* dans SpotBugs), la classe et les lignes de code où la règle n'a pas été respectée.
- Un inconvénient de ces rapports est qu'il n'est pas possible de les utiliser pour accéder directement au code source.
- Un autre *plug-in* de Maven peut résoudre cet inconvénient : JXR [https://maven.apache.org/jxr/maven-jxr-plugin/].
- En effet, JXR permet la création de références croisées entre les rapports et le code source.
- Son utilisation se fait en deux étapes : (i) ajout du *plug-in* à la génération du site et (ii) configuration des autres *plug-ins*.

Après ces modifications, la balise `<reporting>` doit ressembler à :

```
<reporting>
```

```
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-javadoc-plugin</artifactId>
            <version>3.5.0</version>
            <configuration>
                <additionalJOption>-Xdoclint:none</additionalJOption>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-jxr-plugin</artifactId>
            <version>3.1.1</version>
        </plugin>
        <plugin>
            <groupId>com.github.spotbugs</groupId>
            <artifactId>spotbugs-maven-plugin</artifactId>
            <version>4.8.3.0</version>
            <configuration>
                <linkXref>true</linkXref>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-pmd-plugin</artifactId>
            <version>3.21.2</version>
            <configuration>
                <linkXref>true</linkXref>
            </configuration>
        </plugin>
    </plugins>
</reporting>
```

### 3.3.3. Conclusion

- Né au sein du projet Jakarta de la fondation Apache, Maven est devenu un outil incontournable de construction automatique de projets Java, créant les bases de l'intégration continue.
- Actuellement, Maven a un concurrent de poids, Gradle [https://gradle.org], dont le fichier de configuration utilise un format plus lisible que le XML des fichiers POM. Si Gradle n'est pas encore aussi populaire que Maven, il a le potentiel de le devenir.
- La bonne nouvelle c'est que Gradle respecte les conventions établies par Maven et sait utiliser les référentiels local et distant des projets Maven.
- On peut passer d'un projet Maven à un projet Gradle (et vice-versa) sans beaucoup d'effort.
- Maven n'est pas seulement utile lors de la création d'un nouveau projet, il l'est aussi pour les projets existants.
- L'adoption des conventions Maven entraîne des efforts supplémentaires, mais qui ne présente que des avantages.
- En effet, elles ne sont pas exclusives aux projets Maven, elles sont aussi respectées par d'autres outils de production, comme Gradle [https://gradle.org] ou Bazel [https://bazel.build].
- Respecter ces conventions c'est rendre le code plus accessible aux autres développeurs.
- Enfin, les outils d'analyse statique de code fournissent des informations très intéressantes sur le code source.
- Mais ces informations ne doivent pas être considérées individuellement, mais dans leur globalité et dans le contexte d'un projet.
- Ces outils sont personnalisables et peuvent s'adapter aux règles de chaque projet.

### 3.3.4. Après le TP

1. Convertissez certaines classes du projet `dates` en Kotlin et configurez Maven pour compiler les sources Kotlin et Java

Les sources Kotlin doivent êtres placées dans `src/main/kotlin/`

1. Trouvez la commande Maven qui liste toutes les dépendances d'un projet
2. Remplacez la dépendance à Guava version `13.0.1` par une version plus récente

1. Trouvez et configurez un plugin Maven capable de générer un ficher SBOM (*Software Bill Of Materials*) à la fin de la production

1. Ajoutez JUnit Jupiter à vos dépendances
2. Écrivez au moins 1 test unitaire (en utilisation JUnit)
3. Configurez le plugin `surefire` pour exécuter vos tests unitaires
4. Utilisez Jacoco et PITest pour évaluer la qualité de vos tests unitaires

## 3.4. Tutoriel JUnit

### 3.4.1. Introduction

Ernest Hemingway suggère que pour devenir un être humain accompli, une personne doit réaliser 5 tâches :

1. Planter un arbre,
2. Combattre un taureau,
3. Écrire une nouvelle,
4. Élever un enfant
5. Développer sa propre classe Date[2] en Java.

Si vous voulez accomplir la dernière, vous aurez sans doute besoin de tester unitairement les méthodes de cette classe. Et rien de plus adapté pour écrire des tests unitaires en Java que JUnit [http://junit.org/] dans sa version la plus récente.

### 3.4.2. La classe Date

1. Commencez par ouvrir le projet "Software Construction Labs", que vous avez cloné précédemment.
2. Trouvez ensuite le module `date`
3. Ce module n'a qu'une seule classe : `Date`

La classe `Date` que nous allons tester a un constructeur et deux méthodes : `isLeapYear(int)` et `int dayOfWeek()`. Cette classe n'est valable que pour les dates postérieures à l'année 1582 et en France.

Pourquoi seulement à partir de cette date ? Parce que l'année 1582 est un peu spéciale : elle a duré 341 jours en raison de l'adoption du calendrier grégorien par Henri III le 9 décembre (jour qui précéda le 20 décembre 1582). La réforme a été mise en place en France quelques mois après l'Espagne, le Portugal et la Pologne, mais quelques années avant le Royaume Uni, qui a attendu jusqu'à 1752 pour le faire, et la Russie, qui ne l'a adoptée qu'en 1918. Pendant plusieurs années, les pays européens n'avaient pas le même calendrier. Cette confusion a servi de contexte au roman «Le pendule de Foucault» de Umberto Eco, où des sociétés secrètes se sont perdues après un rendez-vous manqué pendant ces années.

Mais revenons à la méthode `isLeapYear(int)`. Une première méthode de test pourrait être écrite comme suit :

```
@Test
void testTwoThousandIsALeapYear() {
    assertThat(Date.isLeapYear(2000)).isTrue();
```

```
}
```

1. Retrouvez la classe de test `DateTest`, qui se trouve dans le dossier `src/test/java`
2. Ajoutez la méthode de test `testTwoThousandIsALeapYear()` à cette classe.

Bien que capable de détecter 1 erreur de codage (si l'année 2000 n'est pas considérée bissextile), cette méthode de test est insuffisante pour détecter d'autres erreurs. À défaut de pouvoir la tester exhaustivement, on peut utiliser l'analyse partitionnelle [https://www.baeldung.com/cs/software-testing-equivalence-partitioning] pour établir des classes d'équivalence et proposer des données de test efficaces. Par exemple :

- Années bissextiles : {1900, 1940, 1996, 2000, 2004}
- Années non bissextiles : {1958, 1962, 1970, 1994, 2002}

Certaines années sont parfois considérées comme bissextiles, alors que ce n'est pas le cas. Par exemple, les années 1700, 1800 et 1900 ne sont pas bissextiles (parce que divisibles par 100), alors que 1600 et 2000 le sont (car divisibles par 400). Cela nous permet de trouver 2 autres classes d'équivalence :

- Années bissextiles inhabituelles : {1600, 2000}
- Années non bissextiles inhabituelles : {1700, 1800, 1900}

Pour utiliser ces différentes données de test, nous allons améliorer notre première méthode de test et la rendre paramétrable.

### 3.4.3. Tests paramétrés

En JUnit 5, les méthodes de test peuvent avoir des paramètres. Dans ce cas, les méthodes doivent être ornées par l'annotation `@ParameterizedTest`, comme suit :

```
@DisplayName("Valid leap years")
@ParameterizedTest
@ValueSource(ints = {1904, 1940, 1996, 2000, 2004})
void testLeapYear(int year) {
    assertThat(Date.isLeapYear(year)).isTrue();
}
```

Cette méthode de test est aussi ornée par l'annotation `@ValueSource` qui dans ce cas, permet de spécifier 5 entiers qui serviront de données de test. JUnit exécutera cette méthode 5 fois, une pour chaque donnée de test.

1. Ajoutez maintenant la méthode ` testLeapYear()` à votre classe de test
2. Exécutez les tests: menu:Run[Run 'DateTest.java']

Notez que l'annotation `@DisplayName` permet de spécifier un nom lisible à la méthode de test. Notez aussi l'utilisation de AssertJ [http://joel-costigliola.github.io/assertj/index.html], une des bibliothèques compatibles avec JUnit qui simplifie l'écriture d'assertions. Hamcrest [http://hamcrest.org] et Truth [http://google.github.io/truth/] sont aussi compatibles avec JUnit.

Nous utilisons ces mêmes annotations pour écrire deux autres méthodes de test pour les années inhabituelles :

```
@DisplayName("Unusual valid leap years")
@ParameterizedTest(name = "year: {0}")
@ValueSource(ints = {1600,2000})
void testUnusualLeapYear(int year) {
    assertThat(Date.isLeapYear(year)).isTrue();
}

@DisplayName("Unusual invalid leap years")
@ParameterizedTest(name = "year: {0}")
@ValueSource(ints = {1700,1800,1900})
```

```
    void testUnusualNotLeapYear(int year) {
        assertThat(Date.isLeapYear(year)).isFalse();
    }
```

1. Ajoutez ces deux nouvelles méthodes à la classe de test

### 3.4.4. Méthodes fournisseuses de données

Une alternative au passage de valeurs par l'annotation `@ValueSource` est d'appeler une méthode qui retourne un `Stream` de données de test. La méthode doit être définie comme `static` et son type de retour doit être compatible avec celui du paramètre de la méthode de test. La liaison entre la méthode de test et la méthode fournisseuse de données se fait par l'annotation `@MethodSource`, qui spécifie le nom de la méthode appelée :

```
    @DisplayName("Invalid leap years")
    @ParameterizedTest
    @MethodSource("fiveTimesWorldChampion")
    void testNotLeapYear(int year) {
        assertThat(Date.isLeapYear(year)).isFalse();
    }

    static IntStream fiveTimesWorldChampion() {
        return IntStream.of(1958,1962,1970,1994,2002);
    }
```

1. Ajoutez cette nouvelle méthode de test à la classe `DateTest`
2. Exécutez à nouveau les tests: menu:Run[Run 'DateTest.java']

Pour tester le constructeur de la classe date, `Date(int,int,int)` nous allons à nouveau appliquer l'analyse partitionnelle pour établir deux classes d'équivalence :

- Dates valides : {10/10/2000, 23/2/1999}
- Dates invalides : {10/10/1000, 33/2/1999, 1/13/1999, 0/2/1999, 10/0/1999}

Comme le constructeur de la classe testée prend 3 paramètres, nous allons définir une méthode de test avec 3 paramètres également :

```
    @DisplayName("Constructor: valid dates")
    @ParameterizedTest
    @MethodSource("validDates")
    void testConstructorValidDates(int d, int m, int y) {
        var date = new Date(d,m,y);
        assertThat(date.year()).isEqualTo(y);
        assertThat(date.month()).isEqualTo(m);
        assertThat(date.day()).isEqualTo(d);
    }
```

Quand la méthode de test déclare plusieurs paramètres, les données fournies doivent être du type `Argument`. Argument est un n-uplet, où le type de chaque composant correspond au type de chaque paramètre de la méthode de test. Dans notre exemple, les trois composants sont de type entier :

```
    static Stream<Arguments> validDates() {
        return Stream.of(
                Arguments.of(10, 10, 2000),
                Arguments.of(23, 2, 1999)
        );
    }
```

1. Ajoutez les méthodes `validDates()` et `testConstructorValidDates()`
2. Exécutez à nouveau les tests unitaires

Une troisième et dernière façon de passer des données à une méthode de test consiste à utiliser l'annotation `@CsvSource`, qui permet de spécifier un ensemble de chaînes de caractères, où chaque chaîne contient des arguments au format CSV (séparés par des virgules) :

```
@DisplayName("Constructor: invalid dates")
@ParameterizedTest
@CsvSource({"10,10,1000","33,2,1999","1,13,1999","0,2,1999","10,0,1999"})
void testConstructorInvalidDates(int d, int m, int y) {
    assertThrows(IllegalArgumentException.class, () -> new Date(d,m,y));
}
```

Notez que cette méthode de test utilise l'assertion `assertThrows()`, introduite dans la version 5 de JUnit et qui fait appel aux expressions lambda de Java 8. Son comportement est simple : elle attend qu'une exception du type passé en premier paramètre soit levée lors de l'évaluation de l'expression passée en deuxième paramètre.

1. Ajoutez cette nouvelle méthode de test à `DateTest` et exécutez les tests.
2. Si tout marche bien, vous avez fini, félicitations !

### 3.4.5. Conclusion

JUnit permet la séparation entre les méthodes de test et les données de tests :

- Les méthodes de test peuvent avoir des paramètres ;
- Les données de test sont décrites soit à l'intérieur d'une annotation, soit par une méthode externe, appelée *fournisseuse de données* ;
- Les méthodes de test sont appelées autant de fois qu'il y a de données de test.

Pour plus d'informations, lisez «https://openclassrooms.com/courses/les-tests-unitaires-en-java[Les tests unitaires en Java]».

## 3.5. Interval

### 3.5.1. Setup : Cloning the GitLab project

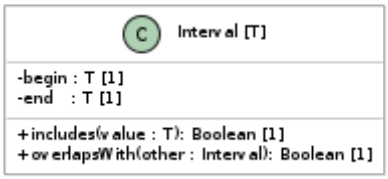
First of all, you clone copy the project containing all labs from the University GitLab. As you will see, all the labs are stored in different directories.

To clone the project, use the following git command:

```
git clone https://gitlab.univ-nantes.fr/gl/construction/labs.git
```

### 3.5.2. Class Implementation

Use the "Test-Driven Development" approach to implement the UML class `Interval`, its attributes and operations.




1. Open the Maven module named `interval`.
2. Part of the implementation is already done: a Java class named `Interval` containing
3. two methods is available, as well as several test cases.
4. There is only a partial implementation of the test cases: only the method comments and its signature . are available.
5. You have to write the body of the test methods.

Follow these steps to achieve the implementation of the method `Interval::includes()`:

1. Start writing the test cases for the method. Read the comments attentively.
2. Then, implement the method and check that all test pass.
3. Improve the quality of the code source: passing all tests is not enough, the code must be readable.
4. Improve the tests, if you estimate that some cases were not tested.

You can check the code coverage of your tests in IntelliJ! To do that, use the "Run Test with Coverage" option when executing your tests. All lines visited by the tests will be highlighted in green (in the editor margin), and lines not tested will be highlighted in red.

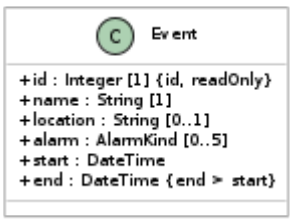
5. You are done for the method.

Now start again the above steps to implement the method `Interval::overlapsWith()`.

## 3.6. UML - Attributs

### 3.6.1. Attribute Implementation

Use Java to implement the UML Class `Event` and its attributes. Use the Getter/Setter implementation strategy introduced during the lectures.


Open the Maven module named `attributes`. Part of the implementation is already done: a Java enumeration named `AlarmKind`, a Java interface named `Event`, and several unit tests (see class `SimpleEventTest`). There is also a partial implementation of the class `SimpleEvent`.

*You have to complete this class:*

1. First, declare the needed Java fields. All fields must be private.
2. Then, implement the class constructor. Remember that some attributes are mandatory and others are not. The constructor must throw a `IllegalArgumentException` if the arguments are invalid.

3. After, implement the fields' accessors and modifiers (getters and setters).
4. Run all tests. Once all tests pass, you are finished.

## 3.7. UML - Associations

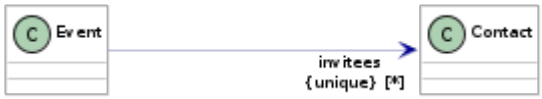
Use Java to implement three UML association between the class `fr.unantes.software.construction.associations.Event` and the classes `Contact`, `Task`, and `Calendar`. Use the *Wrapper* implementation strategy introduced during the lectures.

Open the Maven module named `associations`. Part of the implementation is already done:

- There are several unit tests to ensure the correctness of your implementation.
- There are also 1 java interface implementation for each UML class.
- There are also two java interfaces to help you implement the associations: `SingleReference` and `MultipleReference`.

### 3.7.1. Association between Event and Contact

Start with the simpler association. The association between `Event` and `Contact` is unidirectional, all you have to do is to implement the interface `MultipleReference`, in order to contain several contacts.




### 3.7.2. Association between Event and Task

This association is bidirectional and therefore harder to implement. Your code must ensure the handshake (referential integrity).

You will have to implement the interface `SingleReference` twice: from `Event` to `Task` and from `Task` to `Event`


- Start with the simpler case: a new event (`evt1`) and a new task (`task1`). Ensure that setting the event of `task1` to `evt1` is equivalent to setting the task of `evt1` to `task1`.
- Afterwards, take a harder case: a first event (`evt1`) is related to a task (`task1`) and a second event (`evt2`) is related to a second task (`task2`). Ensure that is you set the task of `evt1` to `task2` (or vice versa):
  - `evt2` will not be related to any task and
  - `taks1` will not be related to any event.

### 3.7.3. Association between Event and Calendar

This last association is also the most complex. The association between `Event` and `Calendar` is also bidirectional, but in this case, one of the references is multi-valued and must implement the interface `MultipleReference`.


### 3.7.4. Conclusion

Once all the tests pass, your are done: now you know how to implement UML associations.

Before packing up your things, let us have some thought: you implemented the same interfaces (`SingleReference` and `MultipleReference`) several times: is it possible to generalize your code and use the same class for different mono-valued references and the same class for different multi-valued references?

[1] Le verbe "maveniser" n'existant pas et "Maven" signifiant "expert" ou "connaisseur", je me permets cette localisation.

[2] Cette dernière suscite de vives controverses.

# Chapitre 4. Annales

## 4.1. QCM - Patrons de conception

### 4.1.1. Quel patron de conception devez-vous utiliser ?

**Lorsque un système doit être configuré avec une famille de produits parmi plusieurs.**

- ☐ Monteur
- ☐ Fabrique abstraite
- ☐ Méthode fabrique
- ☐ Composite

▼ *Solution*

Fabrique abstraite

Le patron de conception Fabrique abstraite doit être utilisé lorsqu'un système doit être configuré avec une ou plusieurs familles de produits. Ce modèle fournit une interface permettant de créer des familles d'objets apparentés ou dépendants sans spécifier leurs classes concrètes. Il permet au code client de créer des objets de types différents appartenant à la même famille, en garantissant que les objets créés sont compatibles et fonctionnent ensemble de manière transparente.

**Lorsque vous souhaitez utiliser une classe existante dont l'interface ne correspond pas à celle dont vous avez besoin.**

- ☐ Adaptateur
- ☐ Décorateur
- ☐ Mediateur
- ☐ Proxy

▼ *Solution*

Adaptateur

Le patron de conception "Adaptateur" doit être utilisé lorsque l'on souhaite utiliser une classe existante, mais que son interface ne correspond pas à celle dont vous avez besoin. Le patron Adaptateur vous permet de créer une nouvelle classe qui sert de pont entre la classe existante et l'interface souhaitée. Cette nouvelle classe, appelée adaptateur, convertit l'interface de la classe existante en l'interface attendue par le client. En utilisant le patron Adaptateur, vous pouvez réutiliser des classes existantes sans modifier leur code, ce qui favorise la réutilisation et la flexibilité du code.

**Lorsque vous souhaitez éviter un lien permanent entre une abstraction et son implémentation.**

Cela peut être le cas, par exemple, lorsque l'implémentation doit être sélectionnée ou changée au moment de l'exécution.

- ☐ Fabrique abstraite
- ☐ Stratégie
- ☐ Pont
- ☐ Méthode Template

▼ *Solution*

Pont

Le patron de conception Pont doit être utilisé lorsque vous souhaitez éviter un lien permanent entre une abstraction et son implémentation. Cela est utile lorsque l'implémentation doit être sélectionnée ou changée au moment de l'exécution. Le patron de conception Pont permet à l'abstraction et à l'implémentation de

varier indépendamment, ce qui permet plus de flexibilité et d'extensibilité dans le code.

Il sépare l'abstraction de son implémentation en créant un pont entre les deux, ce qui leur permet d'évoluer indépendamment. Ce patron favorise un couplage lâche et facilite la maintenance et la modification du code.

**Lorsque le processus de construction doit permettre différentes représentations de l'objet construit.**

- ☐ Singleton
- ☐ Composite
- ☐ Décorateur
- ☐ Monteur

▼ *Solution*

Monteur

Le patron de conception Monteur doit être utilisé lorsque le processus de construction doit permettre différentes représentations de l'objet construit. Le patron Monteur sépare la construction d'un objet de sa représentation, ce qui permet au même processus de construction de créer différentes représentations.

Ce patron est utile lorsqu'il existe plusieurs façons de construire un objet et que le processus de construction doit être flexible et indépendant de la représentation finale.

**Lorsque plusieurs objets peuvent traiter une demande et le gestionnaire n'est pas connu a priori. Le gestionnaire doit être déterminé automatiquement.**

- ☐ Observateur
- ☐ Chaine de responsabilité
- ☐ Décorateur
- ☐ Prototype

▼ *Solution*

Chaîne de responsabilités

Le patron de conception de la chaîne de responsabilité doit être utilisé dans une situation où plusieurs objets peuvent traiter une demande et où le gestionnaire n'est pas connu à l'avance.

Ce patron permet à plusieurs objets d'avoir une chance de traiter la demande, chaque objet de la chaîne ayant la possibilité de transmettre la demande à l'objet suivant de la chaîne.

Le gestionnaire est déterminé automatiquement au fur et à mesure que la demande traverse la chaîne jusqu'à ce qu'elle soit traitée ou qu'elle atteigne la fin de la chaîne.

**Lorsque vous souhaitez paramétrer les objets par une action à effectuer.**

- ☐ Commande
- ☐ Prototype
- ☐ Stratégie
- ☐ Monteur

▼ *Solution*

Commande

Le patron de conception Commande doit être utilisé lorsque vous souhaitez paramétrer des objets par une action à effectuer.

Ce patron dissocie l'émetteur d'une requête du récepteur, ce qui permet à plusieurs requêtes d'être traitées par différents récepteurs. Le patron de commande encapsule une requête sous forme d'objet, qui peut être passé en tant que paramètre, stocké et exécuté à un moment ultérieur.

Cela permet une flexibilité et une extensibilité dans le traitement de différentes actions ou opérations.

**Lorsque vous souhaitez représenter des hiérarchies d'objets en partie ou en totalité.**

- ☐ Composite
- ☐ Itérateur
- ☐ Fabrique abstraite
- ☐ Poids mouche

▼ *Solution*

Composite

Le patron de conception composite doit être utilisé lorsque vous souhaitez représenter des hiérarchies d'objets en partie ou en totalité. Ce patron vous permet de traiter des objets individuels et des groupes d'objets de manière uniforme, en créant une structure arborescente où les objets individuels et les groupes d'objets peuvent être traités comme un seul objet.

Ce patron est utile lorsque vous devez travailler avec des structures complexes qui peuvent être composées de parties plus petites et que vous souhaitez pouvoir effectuer des opérations à la fois sur les parties individuelles et sur l'ensemble de la structure.

**Lorsque vous souhaitez ajouter des compétences à des objets individuels de manière dynamique et transparente, c'est-à-dire sans affecter d'autres objets.**

- ☐ Visiteur
- ☐ Proxy
- ☐ Décorateur
- ☐ Memento

▼ *Solution*

Le patron de conception Décorateur doit être utilisé lorsque vous souhaitez ajouter des responsabilités à des objets individuels de manière dynamique et transparente, sans affecter les autres objets.

Le patron Décorateur vous permet d'ajouter de nouveaux comportements ou fonctionnalités à un objet en l'enveloppant d'une classe Décorateur. De cette manière, vous pouvez ajouter ou supprimer des responsabilités d'un objet au moment de l'exécution, sans modifier sa structure ni affecter les autres objets qui l'utilisent.

**Lorsque vous souhaitez fournir une interface simple à un sous-système complexe.**

- ☐ Adaptateur
- ☐ Façade
- ☐ Fabrique abstraite
- ☐ Composite

▼ *Solution*

Façade

Le patron de conception Façade doit être utilisé lorsque vous souhaitez fournir une interface simple à un sous-système complexe. Ce patron fournit une interface unifiée qui masque les complexités d'un sous-système et permet aux clients d'interagir avec lui de manière simplifiée.

En utilisant le patron Façade, les clients peuvent accéder au sous-système par le biais d'une interface unique, ce qui facilite son utilisation et sa compréhension.

**Lorsqu'une classe souhaite que ses sous-classes spécifient les objets qu'elle crée.**

- ☐ Pont
- ☐ Stratégie
- ☐ Monteur
- ☐ Méthode Fabrique

▼ *Solution*

Méthode fabrique

Le patron de conception Méthode Fabrique doit être utilisé lorsqu'une classe souhaite que ses sous-classes spécifient les objets qu'elle crée. Ce patron permet à une classe de différer l'instanciation d'un objet à ses sous-classes, leur permettant de décider de la classe concrète à instancier.

Cela favorise un couplage lâche entre la classe créatrice et les objets qu'elle crée, puisque la classe créatrice ne dépend que d'une interface abstraite ou d'une classe de base.

**Lorsqu'une application utilise un grand nombre d'objets et les coûts de stockage sont élevés en raison de la quantité d'objets.**

- ☐ Interpreter
- ☐ Prototype
- ☐ Poids mouche
- ☐ Décorateur

▼ *Solution*

Poids mouche

Lorsqu'une application utilise un grand nombre d'objets et que les coûts de stockage sont élevés en raison de la quantité d'objets, il convient d'utiliser le patron de conception Poids mouche. Le patron Poids mouche vise à minimiser l'utilisation de la mémoire en partageant les données communes entre plusieurs objets.

Il y parvient en séparant l'État intrinsèque (partagé entre les objets) de l'État extrinsèque (propre à chaque objet). Ce faisant, le patron Poids mouche réduit l'empreinte mémoire de l'application et améliore les performances.

**Lorsqu'il existe un langage à interpréter, et vous pouvez représenter les énoncés du langage sous forme d'arbres syntaxiques abstraits.**

- ☐ Interpreter
- ☐ Singleton
- ☐ Façade
- ☐ Composite

▼ *Solution*

Interpréteur

Le patron de conception Interpreter est utilisé lorsqu'il existe un langage à interpréter et que vous pouvez représenter les éléments du langage sous forme d'arbres syntaxiques abstraits.

Ce patron permet de définir une grammaire pour le langage et fournit un moyen d'évaluer ou d'interpréter les éléments du langage. Il sépare l'analyse des éléments du langage de leur exécution, ce qui facilite l'ajout de nouvelles expressions ou la modification de la grammaire du langage.

**Lorsque vous souhaitez accéder au contenu d'un objet agrégé sans exposer sa représentation interne.**

- ☐ Itérateur

- ☐ Composite
- ☐ Proxy
- ☐ Pont

▼ *Solution*

Itérateur

Le patron de conception Itérateur doit être utilisé lorsque vous souhaitez accéder au contenu d'un objet agrégé sans exposer sa représentation interne. Ce patron permet d'accéder aux éléments d'un objet agrégé de manière séquentielle sans exposer la structure sous-jacente de l'objet.

Il dissocie le client des détails de l'implémentation de l'objet agrégé, ce qui permet de parcourir et de manipuler facilement la collection.

**Lorsqu'un ensemble d'objets communiquent selon des modalités bien définies mais complexes. Les interdépendances qui en résultent ne sont pas structurées et sont difficiles à comprendre.**

- ☐ Façade
- ☐ Méthode Fabrique
- ☐ Template Method
- ☐ Médiateur

▼ *Solution*

Médiateur

Le patron de conception du médiateur doit être utilisé lorsqu'un ensemble d'objets communiquent de manière bien définie mais complexe, et que les interdépendances qui en résultent ne sont pas structurées et sont difficiles à comprendre.

Le patron du médiateur favorise le couplage lâche en encapsulant la logique de communication entre les objets dans un objet médiateur. Cela permet aux objets d'interagir entre eux indirectement par l'intermédiaire du médiateur, ce qui réduit les dépendances directes entre eux et rend le système plus facile à maintenir et à comprendre.

**Lorsqu'un instantané (une partie) de l'État d'un objet doit être sauvegardé afin qu'il puisse être restauré à cet État ultérieurement, et une interface directe permettant d'obtenir l'État exposerait les détails de l'implémentation et romprait l'encapsulation de l'objet.**

- ☐ État
- ☐ Memento
- ☐ Commande
- ☐ Observateur

▼ *Solution*

Memento

Le patron de conception Memento devrait être utilisé dans ce scénario. Le patron Memento permet de sauvegarder et de restaurer l'état d'un objet sans exposer les détails de son implémentation ni rompre l'encapsulation. Il permet de capturer l'état interne d'un objet et de le stocker à l'extérieur, de sorte qu'il puisse être restauré ultérieurement si nécessaire.

Ce patron est utile lorsqu'il est nécessaire de sauvegarder et de restaurer l'état d'un objet, tout en gardant ses détails d'implémentation cachés.

**Lorsqu'un objet doit être en mesure de notifier d'autres objets sans faire d'hypothèses sur l'identité de ces objets. En d'autres termes, ces objets ne doivent pas être étroitement couplés.**

- ☐ Visiteur

- ☐ Adaptateur
- ☐ Observateur
- ☐ Chaine de responsabilité

▼ *Solution*

Observateur

Le patron de conception Observateur doit être utilisé lorsqu'un objet doit notifier d'autres objets sans faire de suppositions sur l'identité de ces objets. Ce patron permet un couplage lâche entre les objets, car l'objet notifiant n'a pas besoin d'avoir une connaissance directe des objets récepteurs.

Au lieu de cela, l'objet notifiant tient à jour une liste d'observateurs et les informe lorsqu'un événement spécifique se produit. Cela permet au système d'être flexible et extensible, car de nouveaux observateurs peuvent être facilement ajoutés sans modifier l'objet de notification.

**Lorsqu'un système doit être indépendant de la manière dont ses produits sont créés, composés et représentés, et lorsque les classes à instancier sont spécifiées au moment de l'exécution, par exemple par chargement dynamique.**

- ☐ Prototype
- ☐ Façade
- ☐ Fabrique abstraite
- ☐ Poids mouche

▼ *Solution*

Prototype

Le patron de conception Prototype doit être utilisé lorsqu'un système doit être indépendant de la manière dont ses produits sont créés, composés et représentés. Il permet de créer des objets en clonant des objets existants, plutôt que de s'appuyer sur une instanciation explicite.

Ce patron est également utile lorsque les classes à instancier sont spécifiées au moment de l'exécution, par exemple par chargement dynamique.

**Lorsqu'il est nécessaire de disposer d'une référence à un objet plus polyvalente ou plus sophistiquée qu'un simple pointeur et d'envelopper un objet pour en contrôler l'accès.**

- ☐ Itérateur
- ☐ Singleton
- ☐ Stratégie
- ☐ Proxy

▼ *Solution*

Proxy

Le patron de conception Proxy est utilisé lorsqu'il est nécessaire de disposer d'une référence à un objet plus polyvalente ou plus sophistiquée qu'un simple pointeur. Il enveloppe un objet pour en contrôler l'accès, agissant comme un intermédiaire entre le client et l'objet. Le patron Proxy peut être utilisé pour ajouter des fonctionnalités supplémentaires à l'objet, telles que la fourniture d'un niveau de sécurité ou la mise en cache.

Il permet la mise en œuvre du chargement paresseux, où l'objet réel n'est créé que lorsqu'il est nécessaire.

**Lorsqu'il doit y avoir exactement une instance d'une classe et celle-ci doit être accessible aux clients à partir d'un point d'accès bien connu.**

- ☐ Pont
- ☐ Singleton
- ☐ Prototype

☐ Méthode Fabrique

▼ *Solution*

Singleton

Le patron de conception Singleton doit être utilisé lorsqu'il ne doit y avoir qu'une seule instance d'une classe et qu'elle doit être accessible aux clients à partir d'un point d'accès bien connu. Ce patron garantit qu'une seule instance de la classe est créée et fournit un point d'accès global à celle-ci.

Il est couramment utilisé dans les scénarios où il ne doit y avoir qu'une seule instance d'une connexion de base de données, d'un enregistreur ou d'un gestionnaire de configuration, par exemple.

**Lorsque le comportement d'un objet dépend de son État, et il doit changer de comportement à l'exécution en fonction de cet État.**

☐ Médiateur

☐ Décorateur

☐ État

☐ Observateur

▼ *Solution*

État

Le patron de conception État doit être utilisé lorsque le comportement d'un objet dépend de son État et qu'il doit modifier son comportement au moment de l'exécution en fonction de cet État. Ce patron permet à un objet de modifier son comportement en changeant son État interne, sans changer de classe. Il encapsule les différents États dans des classes distinctes et permet à l'objet de déléguer le comportement à la classe de l'État actuel.

Cela favorise la flexibilité et la maintenabilité en séparant la logique du comportement de la classe de l'objet et en facilitant l'ajout ou la modification d'États à l'avenir.

**Lorsque de nombreuses classes apparentées ne diffèrent que par leur comportement ou vous avez besoin de différentes variantes d'un algorithme. Par exemple, vous pouvez définir des algorithmes reflétant différents compromis espace/temps.**

☐ Composite

☐ Méthode Fabrique

☐ Proxy

☐ Stratégie

▼ *Solution*

Stratégie

Le patron de conception Stratégie doit être utilisé lorsqu'il existe de nombreuses classes apparentées qui ne diffèrent que par leur comportement ou lorsque différentes variantes d'un algorithme sont nécessaires. Ce patron permet de définir différents algorithmes reflétant différents compromis espace/temps.

En encapsulant chaque algorithme dans une classe distincte et en permettant au client de choisir l'algorithme souhaité de manière dynamique, le patron Stratégie favorise la flexibilité et la maintenabilité de la base de code.

**Lorsque pour contrôler les extensions des sous-classes. Vous pouvez définir une méthode modèle qui appelle des opérations de "crochet" à des points spécifiques, autorisant ainsi les extensions uniquement à ces points.**

☐ État

☐ Stratégie

☐ Template Method

- ☐ Méthode Fabrique

▼ *Solution*

Méthode template

Le patron de conception des méthode template doit être utilisé lorsque vous souhaitez contrôler les extensions des sous-classes. Ce patron vous permet de définir une méthode modèle qui appelle des opérations "crochet" à des points spécifiques, ce qui signifie que les sous-classes ne peuvent étendre la fonctionnalité de la méthode modèle qu'à ces points spécifiques.

Cela permet d'avoir une structure ou un algorithme commun dans la classe de base, tout en permettant aux sous-classes de fournir leurs propres implémentations pour certaines étapes de l'algorithme.

**une structure d'objets contient de nombreuses classes d'objets avec des interfaces différentes, et vous souhaitez effectuer des opérations sur ces objets qui dépendent de leurs classes concrètes.**

- ☐ Façade
- ☐ Adaptateur
- ☐ Visiteur
- ☐ Décorateur

▼ *Solution*

Visiteur

Le patron de conception Visiteur devrait être utilisé dans ce scénario. Le patron Visiteur vous permet de définir de nouvelles opérations sur un groupe de classes sans modifier leurs implémentations individuelles.

Il est utile lorsque vous avez une structure d'objets complexe avec de nombreuses classes qui ont des interfaces différentes, et que vous voulez effectuer des opérations sur ces objets qui dépendent de leurs classes concrètes.

En utilisant le patron Visiteur, vous pouvez séparer l'algorithme des objets sur lesquels il opère, ce qui permet d'obtenir un code plus souple et plus facile à maintenir.

## 4.2. Contrôle continu 2023-24

### 4.2.1. Techniques de programmation

La classe `ArrayList`, présentée ci-dessous, combine deux techniques de programmation différentes~: défensive et assertive. Basez-vous sur cet exemple pour répondre aux questions suivantes.

```
public class ArrayList<E> {
    private E[] data;
    private int size = 0;
    public ArrayList(int initialSize) {
        data = (E[]) (new Object[initialSize]);
    }
    public E get(int index) {
        if (index < 0 || index >= size) throw new IndexOutOfBoundsException();

        return resolve(index);
    }
    private E resolve(int index) {
        assert index < size && index >= 0;
        assert data != null;

        return data[index];
    }
    public void add(@Nonnull E element) {
        ensureCapacity();
        data[size++] = element;
    }
```

```
    private void ensureCapacity() {
        if (size == data.length) {
            E[] oldData = data;
            data = (E[]) (new Object[size * 2]);
            System.arraycopy(oldData, 0, data, 0, size);
        }
    }
}
```

### Programmation défensive

> Expliquez les principes de la **programmation défensive**

▼ *Solution*

La programmation défensive est une technique de développement de code maintenable.

- Elle garantit que le code se comporte de manière correcte, en dépit d'une entrée incorrecte.
- Elle garantit qu'une méthode ne peut être exécutée que si certaines conditions sont remplies.

### Gardes

> A votre avis, quel était l'objectif du développeur lorsqu'il a ajouté une **garde** à la méthode `get()` ?

▼ *Solution*

L'objectif du développeur est d'empêcher que le tableau soit accédé avec un index invalide (inférieur à zéro ou égal ou supérieur à la taille du tableau)

### Programmation assertive

> Expliquez les principes de la **programmation assertive**.

▼ *Solution*

- C'est une technique qui suit le principe de l'échec rapide et visible.
- Elle empêche de propagation d'erreurs dans le code : l'exécution s'arrête à l'endroit de l'erreur.
- Elle empêche le système de rentrer dans un état incohérent en raison des éventuels changements dans le code source.

### Assertions

> A votre avis, quel était l'objectif du développeur lorsqu'il a ajouté deux **assertions** à la méthode `resolve()` ?

▼ *Solution*

- Le développeur utilise ces assertions pour deux raisons:
  a. Il veut s'assurer que si cette classe est modifiée, les appelants de la méthode `resolve()` continueront à respecter ses pré-conditions, c'est dire, que l'index est valable et que le tableau a bien été initialisé.
  b. Il veut rendre plus lisible la méthode `resolve()`, les assertions expliquent ce que cette méthode attend avant d'être appelée.
- Notez que le code actuel respecte bien les pré-conditions de la méthode `resolve()`: les assertions empêchent l'impossible.

### Annotations

> Le paramètre `element de la méthode ``add()` utilise l'**annotation** ``@Nonnull`. Quelle est l'utilité de cette annotation ? Quel est son impact sur l'exécution de la méthode ?

▼ *Solution*

- Cette annotation permet aux outils d'analyse statique de code et à certains IDEs de vérifier que la méthode `add()` n'est pas appelée avec des arguments nuls.
- L'annotation n'a aucun impact sur l'exécution.

### 4.2.2. Patrons de conception

#### Poids mouche

> Dans quel contexte il est conseillé d'utiliser le patron "**`Poids plume`**" (Flyweight) ? Donnez des exemples de son utilisation.

▼ *Solution*

- On utilise le patron **`poids mouche`** lors qu'on a besoin d'instancier beaucoup de petits objets et que l'on souhaite réduire l'empreinte mémoire.
- La classe `java.lang.Integer` utilise ce patron pour limiter le nombre d'instances.

#### Singleton

> Quel est l'objectif principal du patron de conception "**`Singleton`**" ? Dans quels cas est-il approprié de l'utiliser ?

▼ *Solution*

- L'objectif du patron **`singleton`** est d'assurer qu'une classe n'a qu'une seule instance et fournir un point d'accès global à cette instance et aussi lorsqu'on souhaite l'instancier de manière paresseuse.
- On l'utilise lorsqu'il ne doit y avoir qu'une seule instance d'une classe, et elle doit être accessible aux clients à partir d'un point d'accès bien connu et lorsque cette unique instance doit être extensible par sous-classe et que les clients doivent être en mesure d'utiliser une instance d'une sous-classe sans modifier leur code

#### Décorateur

> Considérez la classe `Character`, qui représente un personnage de jeu de rôles et dont le code source Java est le suivant :

```
public class Character {
    private String name;
    private int    life;
    private Point  position;
    private int    currentHP;
    private int    intellect;
    private int    strength;

    public int attack() {
        return strength * life/100;
    }

    public int defend() {
        return intellect * life/100;
    }

    public void receiveHit(int intensity){
        life = life - intensity;
    }

    public boolean isAlive() {
        return life > 0;
    }

    public int getLife() {
```

```
        return life;
    }

    public String getName() {
        return name;
    }
}
```

On souhaite faire évoluer cette mise en œuvre pour permettre la gestion d'équipements, comme un **`bouclier`** ou une **`épée`**, qui modifient le comportement d'un personnage. Pour ce faire, nous allons utiliser le patron de conception "**`Décorateur`**".

Proposez une solution (une instance du patron Décorateur) permettant la gestion de deux équipements : les boucliers et les épées. Le bouclier doit multiplier par 3 la force de défense, c'est-à-dire, le résultat de la méthode `defend()`. L'épée doit multiplier par 5 la force d'attaque, autrement dit, le résultat de la méthode `attack()`. Le bouclier et l'épée peuvent être utilisés par n'importe quel personnage.

▼ *Solution*

1. Le patron décorateur a besoin d'une classe abstraite ou une interface, les décorateurs ne peuvent pas être des sous-classes de la classe `Character`.
2. Première étape, ajout d'une interface appelée `ICharacter` (le nom n'est pas important), la classe `Character` doit implémenter cette interface.

```
package fr.unantes.sce.patterns.decorator;

public interface ICharacter {
    int attack();
    int defend();
    void receiveHit(int intensity);
    boolean isAlive();
    int getLife();
    String getName();
}
```

3. Deuxième étape, ajout d'un décorateur abstrait. Cette classe n'est pas obligatoire, mais elle permet de réduire le nombre de lignes de code.

```
package fr.unantes.sce.patterns.decorator;

public abstract class AbstractDecorator implements ICharacter {
    protected ICharacter character;

    public AbstractDecorator(ICharacter character) {
        this.character = character;
    }

    @Override
    public int attack() {
        return character.attack();
    }

    @Override
    public int defend() {
        return character.defend();
    }

    @Override
    public void receiveHit(int intensity) {
        character.receiveHit(intensity);
    }

    @Override
    public boolean isAlive() {
```

```
        return character.isAlive();
    }

    @Override
    public int getLife() {
        return character.getLife();
    }

    @Override
    public String getName() {
        return character.getName();
    }
}
```

4. Troisième étape, le décorateur bouclier.

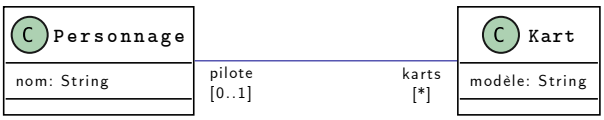
```
package fr.unantes.sce.patterns.decorator;

public class ShieldDecorator extends AbstractDecorator implements ICharacter {

    public ShieldDecorator(ICharacter character) {
        super(character);
    }

    @Override
    public int defend() {
        return character.defend() * 3;
    }
}
```

5. Dernière étape, le décorateur épée.

```
package fr.unantes.sce.patterns.decorator;

public class SwordDecorator extends AbstractDecorator implements ICharacter {

    public SwordDecorator(ICharacter character) {
        super(character);
    }

    @Override
    public int attack() {
        return character.attack() * 5;
    }
}
```

### 4.2.3. Traduction de modèles de conception en code Java

Considérez le diagramme de classes UML présenté ci-dessous.




> Que représente ce diagramme ? Expliquez le rôle de **tous** les éléments qui le composent.

▼ *Solution*

- Le diagramme représente deux **classes**, Personnage et Kart
- La classe Personnage possède un **attribut**, nom, de **type** String

- La classe `Kart` possède, elle-aussi, un seul **`attribut`**, `modèle`, de **`type`** `String`
- Ces deux classes sont reliées par une **`association bidirectionnelle`**, sans nom
- Cette association possède deux **`rôles`**, `pilote` et `karts`
- Le rôle `pilote` la relie à la classe `Personnage` et possède une **`multiplicité`** de valeur `[0..1]`.

### 4.2.4. Diagramme d'objets

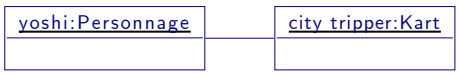
> Dessinez un diagramme d'objets UML représentant l'**`état final`** des instances du diagramme de classes précédent, après l'exécution des opérations énumérées ci-dessous :

1. Création de l'instance `Yoshi` de la classe `Personnage`
2. Création de l'instance `Mario` de la classe `Personnage`
3. Création de l'instance `Carrera Quad` de la classe `Kart`
4. Création de l'instance `City Tripper` de la classe `Kart`
5. Ajout de `City Tripper` aux `karts` de `Yoshi`
6. Affectation de `Mario` en tant que `pilote` de `Carrera Quad`

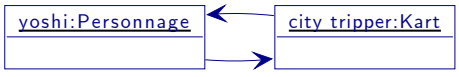
▼ *Solution*

1. La notation des diagrammes objet est disponible ici : https://www.uml-diagrams.org/class-diagrams-overview.html#object-diagram
2. Création des instances

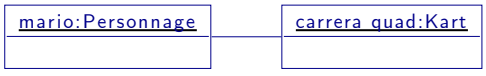

3. Ajout de `City Tripper` aux `karts` de `Yoshi`



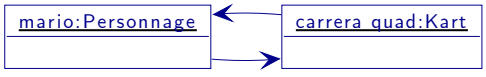

4. Notation alternative (liens unidirectionnels)




5. Affectation de `Mario` en tant que `pilote` de `Carrera Quad`




6. Notation alternative (liens unidirectionnels)



# Chapitre 5. Contrôle continu pratique

## 5.1. Projet Trivial Pursuit

Lisez attentivement les consignes du projet. Le non respect de ces consignes peut résulter en une **très mauvaise évaluation** pour un ou plusieurs participants.

La méconnaissance des consignes ne sera pas un argument valide pour modifier votre note.

### 5.1.1. Introduction

L'objectif du projet est d'améliorer la quallité du code d'un projet existant, en applicant les différents principes vus en cours : mauvauses odeurs de code, refactorings, maintenance, etc.

Lisez attentivement les sections suivantes:

- Section 5.2
- Section 5.3
- Section 5.4

Pour poser des questions sur le projet, utilisez le Canal Mattermost [https://mattermost.univ-nantes.fr/construction/] appropriée.

## 5.2. Préparation

Pour simplifier l'identification des participants, utilisez **toujours** le compte GitLab lié à votre numéro d'étudiant.

### 5.2.1. Avant de commencer

1. Les seuls identifiants valides sont votre **adresse e-mail** et votre **compte GitLab** de l'Université de Nantes.
2. Les validations (commits) du gestionnaire de versions témoignent de l'investissement de chaque participant.
3. Les participants qui n'auront effectué aucune validation seront notés "Absent".
4. Les validations arrivées **après** le 29 mars 2024 à minuit ne seront pas prises en compte.

Ne validez **jamais** un code dont la production automatique (build) ne marche pas.

### 5.2.2. Configuration de vos identifiants Git

1. Pour être sûr que les validations réalisées sur votre machine vous seront correctement attribuées, configurez Git avec votre nom et votre e-mail[1]:

```
git config --global user.name "Jean Pierre"
git config --global user.email "jean.pierre@etu.univ-nantes.fr"
```

### 5.2.3. Configuration du JDK

Si vous utilisez les machines des salles de TPs dans le cadre de ce projet, vous devez utiliser le Java JDK 17. Vous pouvez changer le JDK utilisé dans un projet dans **File > Project structure > Project > Project SDK**.

### 5.2.4. Organisation

Ce projet sera réalisé par groupe d'au plus 4 étudiants. La composition des groupes sera choisie aléatoirement.

Vous allez suivre le processus de maintenance vu en cours. Pour rappel :

Commencez par préparer l'environnement de votre projet :

1. Avant toute chose, un membre du groupe doit créer une **`"Bifurcation"`** du projet sur le serveur [https://gitlab.univ-nantes.fr/gl/construction/trivial-pursuit]. Pour ce faire, cliquez sur l'icône "Fork" (ou "Créer une bifurcation" en français) de la page du projet pour accéder au lien suivant.
2. Ajoutez tous les autres membres du groupe à votre fork.
3. Ajoutez également comme nouveau membre l'utilisateur virtuel appelé **`Naobot`**, avec le statut "Reporter". Cet utilisateur virtuel que nous contrôlons nous donne le droit d'accéder à votre travail et nous permettra de récupérer vos projets.
4. Créez un fichier nommé `CONTRIBUTORS.adoc` et ajoutez-y le prénom, le nom, le numéro d'étudiant et l'adresse email de tous les participants.
5. Créez des étiquettes pour organiser les tickets du projet : *bug*, *improvement*, *smell*, *performance*, etc.
6. Chaque membre du groupe doit cloner **`votre bifurcation du projet`** (et non pas celui d'origine). Toutes vos modifications devront être poussées sur votre *fork* et toutes les issues (ou "tickets" en français) ouvertes le seront sur votre version du projet.
7. **`Il ne doit y avoir qu'une seule "bifurcation" par groupe d'étudiants.`** Elle sera utilisée comme espace de rendu des fichiers liés au projet.

Si vous avez deux bifurcations du projet sur votre compte GitLab, faites en sorte qu'une seule soit visible par l'utilisateur virtuel.

### 5.2.5. Dépendances Maven

Le projet de démarrage est configuré comme un projet Maven standard. Vous êtes libres d'ajouter de nouvelles extensions lors du développement du projet. Par défaut, les dépendances suivantes sont configurées :

- JUnit Jupiter [https://junit.org] pour exécuter les tests unitaires.
- Mockito [https://site.mockito.org] pour créer des bouchons de test.
- Atlanmod Commons [https://github.com/atlanmod/Commons] qui fournit une extension de la librairie Java standard.
- AssertJ [https://joel-costigliola.github.io/assertj/], qui permet l'écriture d'assertions "fluides" en Java.

## 5.3. Travail à réaliser

### 5.3.1. Contexte

Le projet Trivial Pursuit est composé de 3 modules, développés par des équipes différentes.

**`Module A`**
: Ce module est le moins abouti. Il est composé de quelques classes et possède une IHM permettant l'ajout de cartes, développée en Java Swing.

**`Module B`**
: Ce module contient une application presque complète, permettant de jouer à Trivial Pursuit. Son IHM a été développée en Java Swing.

**`Module C`**
: Ce module est le plus abouti. Il possède lui-aussi une IHM, développée en JavaFX. Attention, l'exécution de ce module demande un JDK contenant le module JavaFX.

### 5.3.2. La chasse aux mauvaises odeurs

La première étape consiste à énumérer, dans le fichier `SMELLS.adoc`, les mauvaises odeurs que vous trouverez dans le code source. Vous pouvez effectuer cette tâche manuellement, en utilisant des outils d'analyse statique, ou les deux.

Quelques exemples d'outils de détection de mauvaises odeurs pour java, disposant d'un plugin Maven :

1. PMD [https://maven.apache.org/plugins/maven-pmd-plugin/]
2. Checkstyle [https://maven.apache.org/plugins/maven-checkstyle-plugin/]
3. FindBugs [https://gleclaire.github.io/findbugs-maven-plugin/]

L'éditeur IntelliJ propose un plugin appelé Sonarlint [https://www.sonarlint.org/], capable de détecter les code smells dans vos projets. Nous vous recommandons de l'installer et de l'utiliser dans le cadre de ce projet.

Pour l'installer, vous avez deux options:

1. Aller dans **`Préférences > Plugins > Sonarlint`** puis cliquer sur **`[Installer]`**.
2. L'installer manuellement : https://plugins.jetbrains.com/plugin/7973-sonarlint

### 5.3.3. Évaluation de la suite de tests

La deuxième étape consiste à déterminer les parties du code qui ne sont pas bien testées. Comme vous allez restructurer le code, vous aurez besoin d'un filet de sécurité capable de détecter l'introduction d'erreurs. Les tests unitaires joueront ce rôle.

Vous pouvez utiliser le *plug-in* de couverture de code de IntelliJ, ou alors le plugin Maven JaCoCo [https://www.eclemma.org/jacoco/trunk/doc/maven.html].

L'analyse de mutation est une alternative à la couverture de code, qui permet aussi de déterminer les parties du code qui sont mal testées. Vous pouvez utiliser le plug-in Maven Pitest [https://pitest.org/quickstart/maven/] pour le faire.

### 5.3.4. Création de tickets

Une fois que vous avez énuméré les mauvaises odeurs et que vous savez quelles parties du code doivent être mieux testées, c'est le moment d'organiser votre travail.

Pour chaque mauvaise odeur à éliminer et test à réaliser :

1. Ouvrez un ticket dans votre projet Gitlab (sur l'interface en ligne de Gitlab, section *Tickets*). Vous y détaillerez les points suivants :
   - Un bref résumé du problème lié au ticket.
   - Comment la solution au ticket doit être mise en œuvre ?
2. **`Associez un membre du groupe`** à la résolution du ticket, via l'interface de GitLab. Cette personne sera chargée de résoudre le ticket.

Certains tickets sont plus longs et peuvent être réalisés par différents membres, tout au long du projet. Par exemple, les tickets qui concernent la traduction ou le renommage d'identifiants.

1. Écrivez le code qui résout le ticket.

Faites attention à la régression ! Toute modification ne doit pas "casser" du code qui marchait auparavant (les autres tests unitaires doivent passer).

1. Si jamais vous devez changer d'approche au niveau des tests, de l'implémentation, etc, **`ajoutez un commentaire sur le ticket Gitlab`** pour documenter tout changement. **`N'éditez pas le texte du ticket original`**, afin de garder un historique de votre travail.

2. Effectuez un (ou plusieurs) commit(s) pour pousser vos modifications sur le dépôt, en référençant le numéro du ticket et en indiquant votre progression dans sa résolution. Nous vous invitons à suivre la norme Conventional Commits [https://www.conventionalcommits.org/].
3. Enfin, quand le ticket est résolu, marquez-le comme "résolu" dans l'interface de Gitlab. Vous pouvez aussi fermer les tickets automatiquement à l'intérieur d'un message de commit : Automatic issue closing [https://docs.gitlab.com/ee/user/project/issues/managing_issues.html#closing-issues-automatically]

Le code du projet est là pour vous fournir une base de code. Vous êtes libre de *modifier l'implémentation comme vous l'entendez*. **`Mais attention, vous devrez motiver tous vos changements dans vos différents tickets/commits !!!`**

### 5.3.5. Écriture de tests unitaires

Écrivez les test unitaires en JUnit 5 (Jupiter). Si vous souhaitez, vous pouvez utiliser des outils de génération de tests unitaires, comme :

1. Symflower [https://symflower.com]
2. Evosuite [https://www.evosuite.org] (Attention, il demande un JDK $\leq 9$)
3. Randoop [https://randoop.github.io/randoop/]
4. Diffblue [https://www.diffblue.com] (Limité à 100 tests par semaine)
5. Squaretest [https://squaretest.com/] (Limité à 30 jours)
6. Codiumate [https://www.codium.ai/]

Vous pouvez aussi faire appel à un grand modèle de langage, comme Llama, Gemini ou ChatGPT.

### 5.3.6. Refactorings

Pendant cette étape vous allez utiliser les opérations de refactoring pour supprimer les mauvaises odeurs. Référez-vous au cours sur les refactorings pour trouver des solutions correspondantes aux mauvaises odeurs.

Bien que IntelliJ propose un ensemble important d'opérations de refactoring, cet ensemble n'est pas complet. Vous pouvez réaliser des refactorings manuellement, en faisant attentions aux préconditions de chaque opération.

### 5.3.7. Consolidation

Dans cette dernière étape, vous allez consolider le code.

1. Créez un module Maven appelé `trivial-core` et ajoutez-le comme dépendance des modules A, B et C.
2. Ce module contiendra les classes communes aux trois autres modules.
3. Par exemple, les trois modules possèdent une classe appelée `Player` : trouvez ce que ces classes ont en commun et déplacez ces propriétés à une super-classe commune, que vous placerez dans le nouveau module.
4. Faites la même chose avec les autres classes communes aux trois modules.

## 5.4. Evaluation

Le projet s'effectue en groupe, mais les évaluations sont **`individuelles`**. Si vous souhaitez faire de la programmation en binôme, pensez à alterner les comptes.

- Le travail à rendre se composera de votre **`bifurcation en ligne Gitlab`**, sur lequel vous aurez poussé toutes vos modifications. Cela inclut également tous les messages de commits et tickets ouverts.
- Ajoutez un fichier "RENDU.adoc" à la racine du projet, afin de décrire les spécificités de votre projet (choix techniques, parties non traitées, extensions non demandées, etc.).
- Pour être évalué, **`tout étudiant doit participer activement du projet`**, en réalisant des "commits", en ajoutant des lignes de code, en ouvrant des tickets sur le serveur GitLab, etc.
- L'évaluation portera sur les critères suivants :

- Qualité des tickets ouverts sur votre projet Gitlab : description du problème, des tests requis et de la solution mise en œuvre.
- Qualité du code produit (amélioration du code source).
- Qualité et pertinence des tests unitaires mis en place.
- Approche choisie pour résoudre chaque ticket.
- Qualité des messages de commits, respect de la spécification Conventional Commits [https://www.conventionalcommits.org/].

**`Ne sacrifiez pas la qualité à la quantité !`** Il vaut mieux rendre un projet bien réalisé avec des tickets non résolus qu'un projet avec tous les tickets mal résolus.

[1] Seulement si vous ne vous appelez pas Jean Pierre