TP 2 : Design Patterns

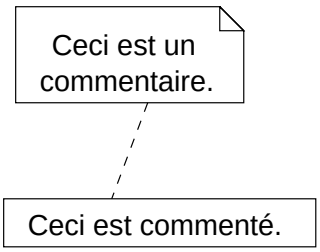
Le but de ce TP est de vous familiariser avec quelques-uns des motifs couramment utilisés en programmation objet. Ces motifs (*design patterns*) servent généralement à faciliter la maintenance du code, en évitant en particulier la multiplication de classes ou des dépendances entre classes.

En préalable à cet aperçu de quelques designs patterns, nous allons introduire une notation visuelle pour les relations entre classes.

1 Diagrammes de classe UML

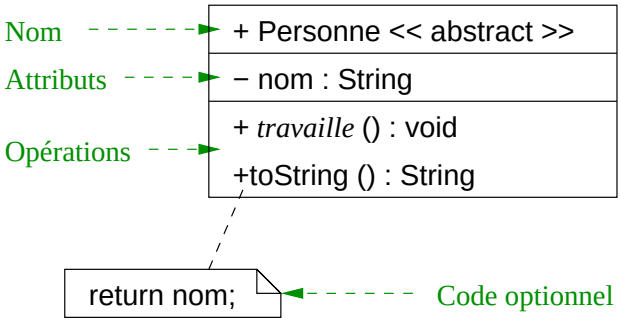
UML est une norme complexe de description de programmes informatiques développée par un consortium d'entreprises et de laboratoires, l'OMG. L'un des intérêts majeurs d'UML est son indépendance vis-à-vis du langage choisi pour l'implémentation, avec pour seule contrainte qu'il soit orienté objet (par exemple Java, mais aussi C++, Eiffel, Objective C, OCaml, SmallTalk, Ruby, C#, ...). Les diagrammes de classes ne constituent qu'une petite partie des possibilités d'UML.

1.1 Commentaire



N'importe quelle information qui ne rentre pas vraiment dans les catégories suivantes, par exemple un morceau de code d'implémentation.

1.2 Classes



La représentation contient trois compartiments :

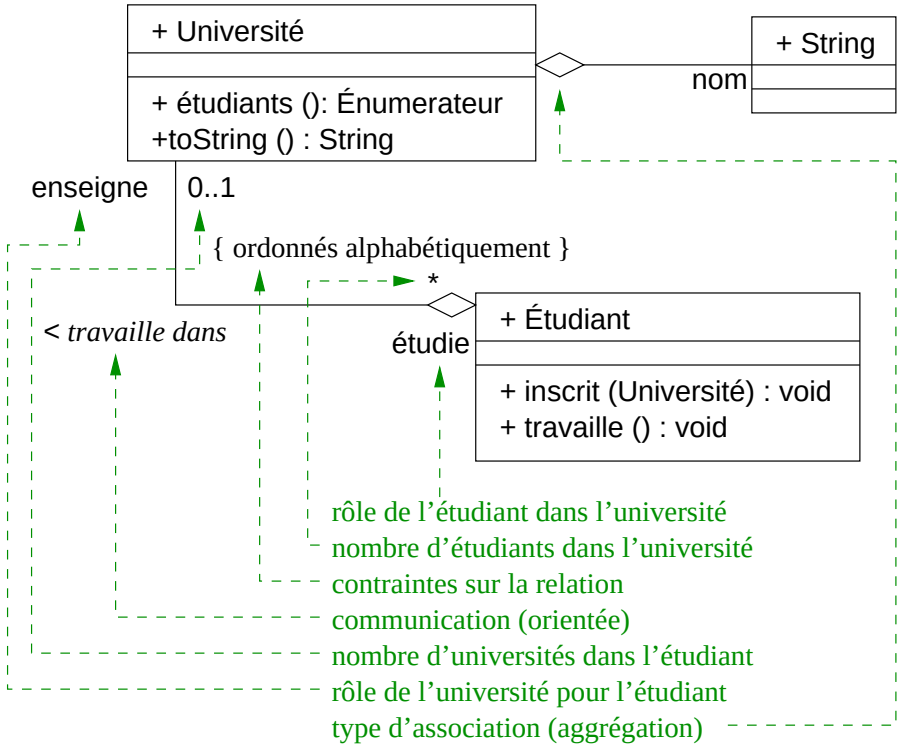
1. le *nom* contient le nom de la classe et d'autres informations de documentation telles que vous les mettriez dans la `javadoc`. Les guillemets identifient des *stéréotypes* comme « `abstract` ». Le `+` est un modificateur d'accès :
   - `+` public
   - `#` protégé
   - `-` privé
2. les *attributs* peuvent aussi être représentés à l'aide d'une relation d'aggrégation
3. les *opérations* sont les définitions des méthodes : `nomDeMéthode (paramètres) : typeDeRetour`. Les opérations abstraites apparaissent en italiques.

Exemple 1.1.

```
/**
 * Classe abstraite représentant une personne physique.
 */
public abstract class Personne {
    /** Le nom de la personne. */
    private String nom;

    /**
     * Invariant de classe.
     * @returns <code>nom.length() > 0</code>.
     */
    protected boolean fulfillsInvariant() {
        returns nom.length() > 0;
    }

    /**
     * Constructeur d'une Personne.
     * @param nom Le nom de la Personne crée.
     * @throws IllegalArgumentException si <code>nom.length() == 0</code>.
     */
    public Personne(String nom) throws IllegalArgumentException {
        if (nom.length() == 0)
            throw new IllegalArgumentException("Nom vide.");
        else
            this.nom = nom;
        assert fulfillsInvariant(): "Nom vide.";
    }

    /**
     * Fait travailler la Personne.
```

```
     */
    protected abstract void travaille();

    /**
     * Retourne le nom de la personne.
     * @returns {@link #nom}.
     * @overrides java.lang.Object#toString()
     */
    @Override
    public String toString() {
        return nom;
    }
}
```

1.3 Associations



Une association est une relation entre deux classes. On la décrit à l'aide :

- de *rôles* d'un objet d'une classe dans un objet de l'autre (omis si évidents) ;
- de *cardinalités*, c'est-à-dire de nombres d'objets d'une classe dans un objet de l'autre classe ; on utilise pour cela les notations :

  `1` habituellement omis si 1 pour 1

  `n` inconnu au moment de la compilation, mais borné

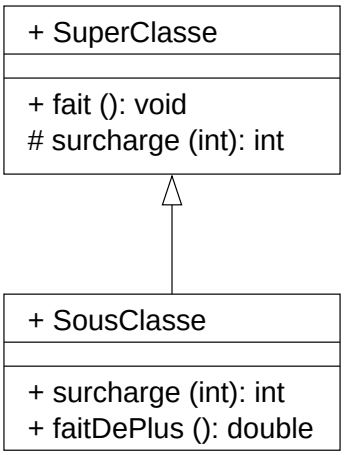
`0..n` entre 0 et n

`1..*` 1 ou plus

`*` 0 ou plus

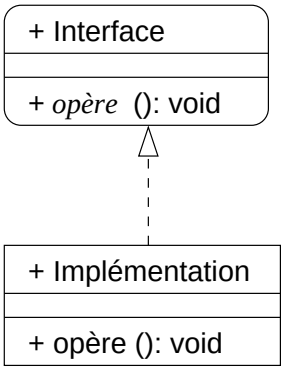
– une *communication* orientée si nécessaire par `<` ou `>`, ou bien avec des flèches sur les traits ;
– un *type de relation* orientée : un des grands types d'association décrits par la suite : aggrégation, héritage, ... si utile.

1.3.1 Héritage (généralisation/spécialisation)



Une flèche triangulaire vide décrit une dérivation. La classe dérivée est la classe de base, mais avec des propriétés additionnelles ou modifiées. Elle spécialise ou étend la superclasse plus générale.

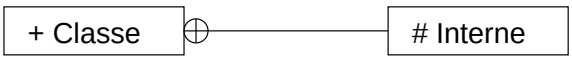
1.3.2 Héritage d'interface (spécifie / raffine)



Une flèche d'héritage en tirets indique qu'une classe raffine ou implémente une interface.

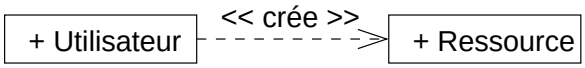
L'interface elle-même est indiquée soit en précisant le stéréotype « `interface` » dans le nom de la classe, soit en utilisant des coins arrondis.

1.3.3 Classe interne



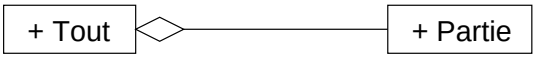
Marque la présence d'une classe interne à une autre.

1.3.4 Dépendance



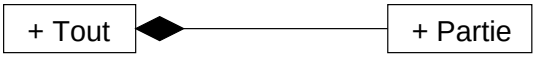
Une classe utilise une autre classe, mais sans que la ressource soit un membre de l'utilisateur. Si la classe de ressource est modifiée, il y a peut-être des méthodes à modifier dans l'utilisateur. La ligne est souvent stéréotypée par « `instancie` » ou « `utilise` ».

1.3.5 Aggrégation (contient)



La destruction du tout ne détruit pas les parties.

1.3.6 Composition (possède)



La destruction du tout détruit les parties. Rare en Java.

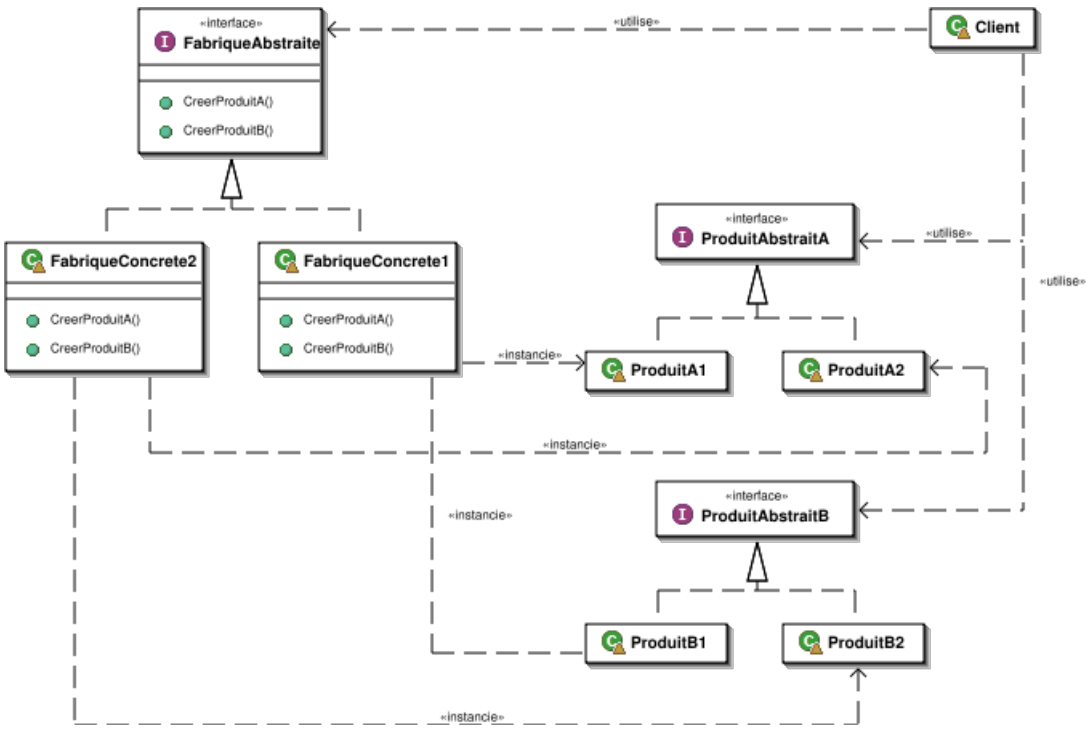
Exercice 1. Dessiner le diagramme de classe de votre implémentation des pièces d'échecs du TP 1.

2 Patrons de conception

Cette section présente quelques-uns des modèles d'organisation de classes les plus courants. On parle d'instancier un tel modèle quand on l'utilise dans le contexte concret d'une application.

2.1 Fabrique abstraite

Le principe d'une fabrique abstraite est de découpler la création de l'utilisation de certains objets. Dans ce patron, les objets clients disposent d'une référence vers une fabrique concrète (par exemple un objet de la classe `FabriqueConcrete1` dans le diagramme suivant), qui aura la charge d'instancier les objets adéquats (des classes `ProduitA1` et `ProduitB1` dans l'exemple). Cependant, les clients utilisent l'interface `FabriqueAbstraite` pour ces instanciations, et n'ont pas besoin de connaître l'existence des différents produits concrets.



Exercice 2 (Instance du patron de fabrique abstraite)**.** On désire implémenter un moteur de jeu générique dans lequel deux éléments peuvent varier : les adversaires (on prévoit des obstacles `Puzzle` et `NastyVillain`) et les joueurs (on prévoit initialement deux types de joueurs `Kitty` et `KungFuGuy`). Cependant les divers jeux que l'on veut créer visent des publics différents : `Kitty` vs. `Puzzle` d'un côté, et `KungFuGuy` vs. `NastyVillain` de l'autre : pas question de faire jouer une instance de `Kitty` contre une de `NastyVillain`.

1. Instancier le patron de la fabrique abstraite pour permettre la création et le lancement de différents jeux à partir du moteur générique.
2. Comment ajouter un jeu `Fairies` vs. `Gnomes` dans votre moteur de jeu ?

Exercice 3 (Injection de dépendance)**.** Une fabrique abstraite nécessite de la part du client de connaître l'existence d'une fabrique. Comment écrire le client, c'est-à-dire le code qui va interagir avec la fabrique et les objets qu'elle crée ?

2.2 Observateur

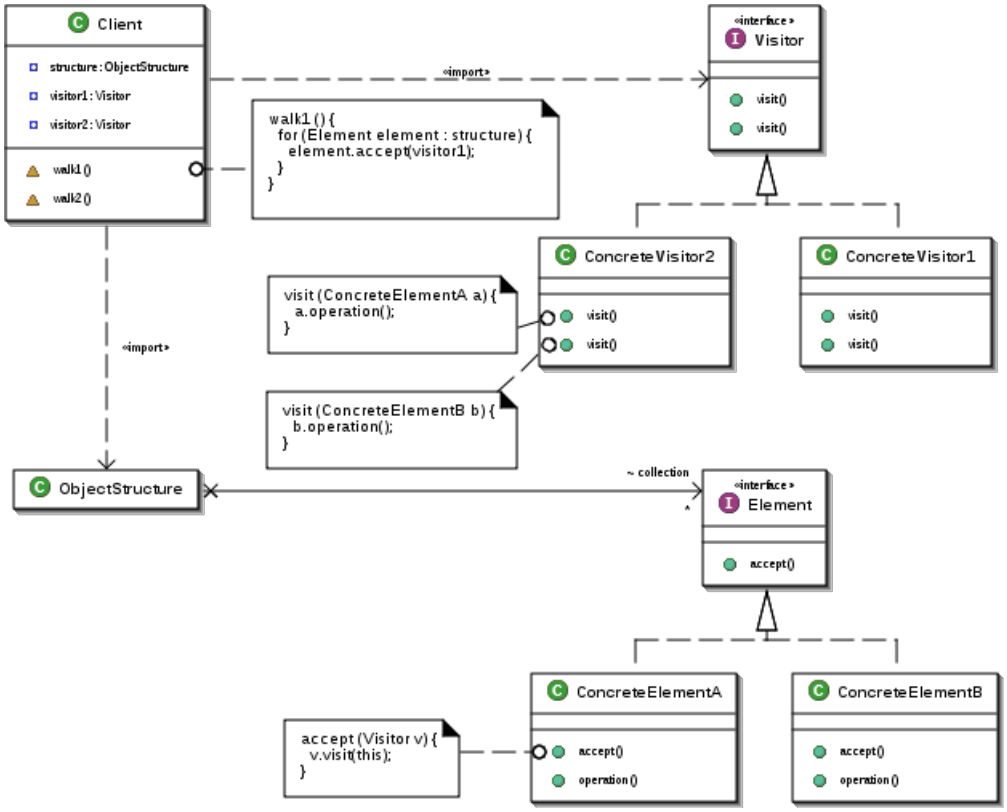
Voici un patron aisément réutilisable, et déjà implémenté en Java dans `java.util.Observer` et `java.util.Observable`.

Exercice 4 (Diagramme de l'implémentation Java)**.** Lire la documentation de `java.util.Observer` et `java.util.Observable` et dessiner le diagramme de classes correspondant.

Exercice 5 (Instance du patron d'observateur)**.** Implémenter en Java une classe `Flower` qui a deux états ouverts et fermés, et des classes `Bee` et `HummingBird` qui observent des fleurs.

2.3 Visiteur

Le patron du visiteur permet d'implémenter différentes opérations sur une hiérarchie de classe entière sans avoir à ajouter une méthode pour chaque opération dans chaque classe.



Exercice 6 (Instanciation du patron de visiteur)**.** Créer une hiérarchie de classes avec des classes `Chrysanthemum`, `Gladiolus`, etc. sous `Flower`. Nos abeilles et pinsons vont maintenant visiter les fleurs avec des comportements différents : les abeilles butinent les `Chrysanthemum` mais pas les `Gladiolus` et les pinsons ont le comportement inverse.