Rafraichissement Orienté Objet

Matières abordées

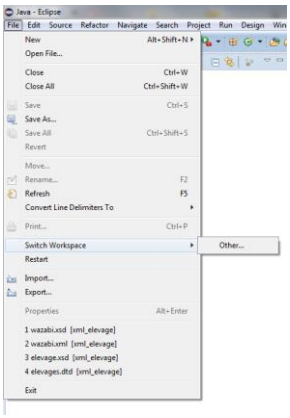
Cette séance a pour but de rafraichir les concepts orientés Objet abordés dans le cadre du cours d'Analyse et Programmation Orientée Objet du bloc1.

Les thèmes abordés sont : attributs de classe et d'instance, `ArrayList`, égalité, constructeurs (chaîne des constructeurs), encapsulation, exceptions (checked et unchecked), héritage, interface, `LocalDate`, `Iterator`, dynamic binding, polymorphisme, overriding, `super`, `this`, javadoc , …

Si ces thèmes restent flous au terme de la séance, il est indispensable d'y travailler davantage avant les séances d'Ateliers Java, de Mobile et de Structure de données.

Travailler Orienté Objet

Le workspace et le projet Eclipse



Pour bien débuter les séances de programmation, il est indispensable de structurer son espace de travail.

1. Dans Z :, créez un répertoire intitulé workspacePJA.
2. Ouvrez Eclipse néon dans ce workspace. Soit Eclipse demande le workspace à l'ouverture, soit il ouvre directement le dernier workspace. Dans ce cas, changez-le.
3. Créez un projet Java intitulé AJ_seance1.

Les consignes de programmation

Votre code doit exploiter au mieux les richesses d'une découpe orientée Objet :

- Les copier-coller sont formellement interdits.
- Chaque attribut est encapsulé.
- Chaque objet est responsable de ses propriétés (état et comportement).
- Le code est réutilisable, lisible et bien structuré (indenté).

Les compléments théoriques

Nous vous demandons de structurer davantage votre code en utilisant des packages.

Vous trouvez les théories nécessaires sur claco :

- Héritage
- Date
- Package
- Exceptions
- Interface
- Equals/hashCode

L'interface `Util`

Nous vous fournissons une interface bien utile pour les différents tests de vos paramètres. Placer la dans un package `util`.

Enoncé

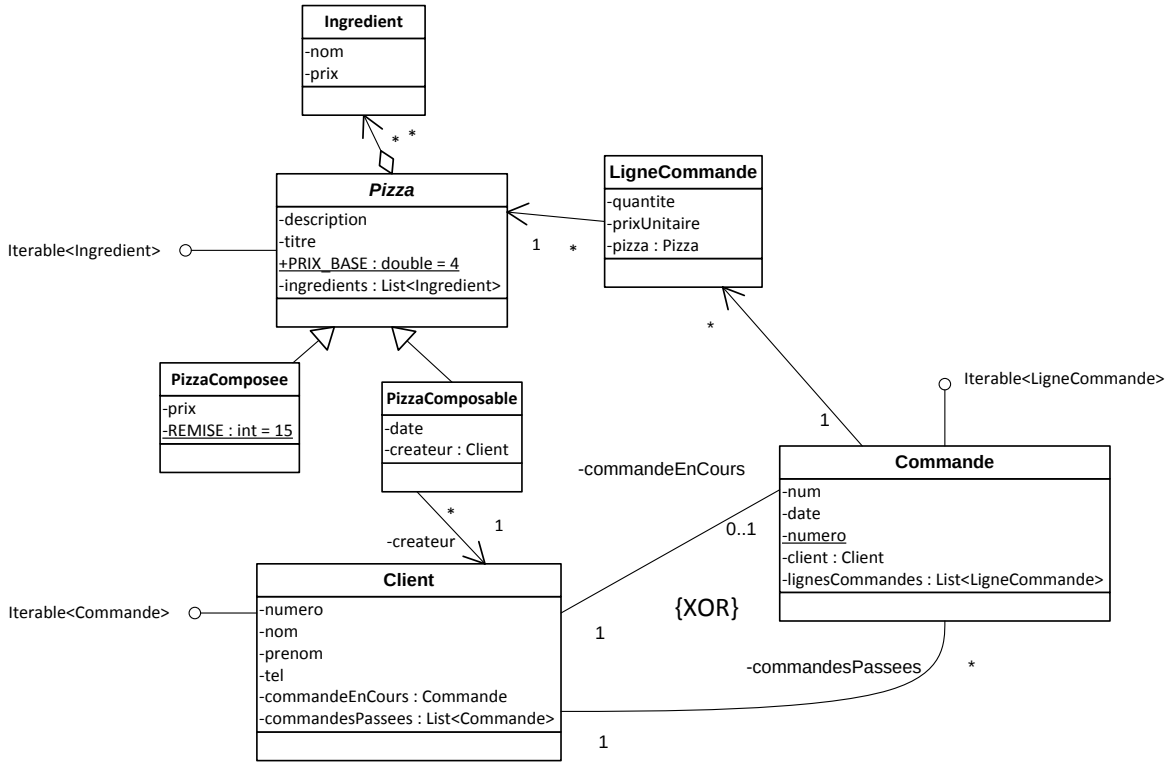
Il s'agit d'implémenter une application déjà partiellement analysée dans le cadre du cours d'UML : la pizzeria en ligne.

L'application doit permettre de gérer les commandes de pizzas effectuées par des clients. Les pizzas sont soit composées par la pizzeria soit par le client selon ses goûts (ces dernières sont appelées pizzas composables).

L'application est décomposée dans un premier temps en 2 packages : domaine et casUtilisation.

Le domaine

Le domaine de l'application se résume par le diagramme de classes suivant :



Une **pizza** possède une liste d'**ingrédients**. Un ingrédient renferme simplement un nom et un prix. Ces informations sont fournies en paramètres du constructeur. Elles sont accessibles mais ne possèdent pas de modificateurs. Il ne peut pas y avoir deux ingrédients qui possèdent le même nom.

Un même ingrédient ne peut se trouver deux fois dans la liste des ingrédients d'une pizza. Comment garantir cela par programmation ?

Réponse :

Une **pizza**, qu'elle soit composée ou composable, possède un titre et une description. Le titre d'une pizza permet de l'identifier (il n'existe pas deux pizzas de même titre). Lorsqu'il s'agit d'une pizza composée, le titre et la description sont choisis par la pizzeria. Tandis que lorsqu'elle est composable, le titre est déterminé par l'application (format = « Pizza composable du client » + le numéro du client) et la description mentionne le nom et le prénom du client qui l'a créée (format = « Pizza de »+nom suivi du prénom du client). Lorsqu'on crée une pizza composable, on retient sa date (jour et minutes) de création.

Ceci fait apparaitre une incohérence concernant le titre puisqu'alors les pizzas composables d'un client ont toutes le même titre. Comment remédier à cette incohérence sans modifier la classe `Pizza` et sans modifier le titre ? Quel concept orienté Objet permet de gérer cela ?

Réponse :

Il doit être aisé de parcourir la liste des ingrédients de la pizza. On doit pouvoir également ajouter et supprimer un ingrédient de cette liste. Les accesseurs sont disponibles pour tous les attributs à l'exception de la liste des ingrédients (pas de modificateur).

Lorsqu'on crée une pizza composée, il faut fournir une liste d'ingrédients. À partir du total des prix des ingrédients de la liste, cette classe fournit un calcul du prix de la pizza. Ce prix est majoré du prix de base mais une remise (exprimée en pourcent : 15%) est octroyée. On vous demande d'effectuer un arrondi supérieur de la valeur obtenue (plus d'info dans la javadoc : `java.Math`).

Attention, c'est bien la classe `Pizza` qui a la responsabilité de la gestion des ingrédients. C'est donc cette classe qui les parcourt et fournit un montant total. C'est la responsabilité de la pizza composée de calculer son prix sur base du montant calculé par la classe `Pizza`.

Lorsqu'on crée une pizza composable, on fournit uniquement le créateur de la pizza. Il faudra alors ajouter ou retirer les ingrédients unitairement.

Il ne doit pas être possible de modifier (ajouter ou retirer) des ingrédients lorsqu'il s'agit d'une pizza composée. Une exception est fournie par java pour ce genre de comportement.

Laquelle ?

Réponse :

Est-elle checked ou unchecked ?

Réponse :

Un **client** possède un numéro, un nom, un prénom et un numéro de téléphone. Toutes ces informations sont fournies lors de la création de l'objet. Il existe des accesseurs pour tous les attributs sauf pour la liste des commandes passées. Pour pouvoir consulter ces commandes, on implémente l'interface `Iterable`. Le numéro permet d'identifier clairement le client.

La classe `Client` doit permettre d'ajouter une commande en cours au client. Ceci est possible seulement s'il n'en n'a pas déjà une. Il faut signaler si la commande a pu être ajoutée ou pas. Vérifiez bien la cohérence de votre code.

La classe `Client` doit aussi permettre de clôturer la commande en cours. Inutile dans ce cas de la passer en paramètre, pourquoi ?
Si jamais il n'y a pas de commande en cours, signalez que l'opération n'est pas possible en lançant une NoCommandeEnCoursException.

Réponse :

La clôture de la commande permet donc de placer la commande en cours dans les commandes passées. La commande est donc soit en cours soit dans les commandes passées (c'est le XOR).

Une commande contient un num. Il ne peut exister deux commandes avec le même `num` ; il s'agit de la définition de l'**égalité structurelle** de la commande. Comment implémenter cela en Java ? Quelles conséquences pour les collections de commandes ?

Réponse :

Le `num` doit être auto incrémenté par la classe elle-même grâce à la variable de classe `numero`.

Qu'est-ce qu'une variable de classe ? Comment préciser cela en java ?

Réponse :

Lorsqu'on crée une commande, on fournit uniquement le client qui passe la commande. La date est celle de l'instant de construction. Il faut également ne pas oublier d'ajouter la commande au client (l'association est bidirectionnelle !). Si cet ajout ne fonctionne pas, il faut empêcher la création de l'objet avec une exception unchecked que vous connaissez bien …

En Java, que signifie une association bidirectionnelle ?

Réponse :

La commande doit permettre d'ajouter des pizzas en précisant la quantité souhaitée. Ces informations sont maintenues dans une liste de lignes de commande. Chaque ligne de commande possède une quantité, une pizza ainsi que le prix unitaire de celle-ci.

Pourquoi est-il nécessaire d'indiquer le prix unitaire dans la ligne de commande alors qu'il se trouve déjà dans la pizza ?

Réponse :

Il faut veiller à ce qu'il n'existe pas deux lignes de commande pour la même pizza dans une commande.

Attention aux éventuelles `ConcurrentModificationException` … Quoi donc ?

Réponse :

`Commande` implémente `Iterable<LigneCommande>` et fournit les accesseurs pour les attributs `date`, `num` et `client`. Il faut ajouter une méthode qui calcule le montant total de la commande et une autre méthode qui renvoie la chaîne de caractères qui détaille toutes les lignes de commande.

La classe `LigneCommande` construit ses instances avec les informations suivantes : une pizza et une quantité. Elle contient une méthode qui calcule le montant total de la ligne de commande. Elle fournit les accesseurs de ses attributs.

Dans le constructeur de `LigneCommande`, le code permettant d'initialiser le prix unitaire ne dépend pas du type de pizzas bien que le calcul du prix soit dépendant du type.

Comment appelle-t-on ce mécanisme en orienté objet?

Réponse :

Les cas d'utilisation

Le singleton

Dans le package `casUtilisation`, on implémente les différents cas d'utilisation de l'application.

Ce package contient une seule classe `GestionPizzeria`. Cette classe ne peut être instanciée qu'une seule fois. On appelle cela un singleton. Il s'agit d'un design pattern dont l'objet est de restreindre l'instanciation d'une classe à un seul objet. Il est utilisé lorsque l'on a besoin d'exactement un objet pour coordonner des opérations dans un système (parfait pour gérer nos cas d'utilisation !).

Pour implémenter un singleton, il existe plusieurs méthodes. Voici celle que nous vous recommandons dans un premier temps :

```
public final class Foo {

    private static class FooLoader {
        private static final Foo INSTANCE = new Foo();
    }

    private Foo() {
        if (FooLoader.INSTANCE != null) {
            throw new IllegalStateException("Already instantiated");
        }
    }

    public static Foo getInstance() {
```

```
        return FooLoader.INSTANCE;
    }
}
```

Extrait du site http://stackoverflow.com/questions/70689/what-is-an-efficient-way-to-implement-a-singleton-pattern-in-java

Nous vous recommandons fortement ce site qui regorge d'informations constructives, de solutions, de trucs, … dans beaucoup de langages de programmation. http://stackoverflow.com

Lors de la création de l'instance, on crée aussi les différentes pizzas composées de la pizzeria. Ces objets sont des attributs `public` de la classe et ne doivent plus être modifiables après la création de l'instance; ils représentent la carte de la pizzeria.

Voici le code :

```
        ArrayList<Ingredient> ingredients = new ArrayList<>();
        ingredients.add(tomate);
        ingredients.add(artichaut);
        ingredients.add(jambon);
        ingredients.add(olives);
        ingredients.add(parmesan);
        ingredients.add(mozza);
        p_4saisons = new PizzaComposee("4 saisons", "4 goûts qui défilent selon les saisons",
ingredients);
        ingredients = new ArrayList<>();
        ingredients.add(tomate);
        ingredients.add(parmesan);
        ingredients.add(gogonzola);
        ingredients.add(pecorino);
        ingredients.add(mozza);
        p_4fromages = new PizzaComposee("4 fromages", "le mélange généreux des fromages italiens",
ingredients);
        ingredients = new ArrayList<>();
        ingredients.add(tomate);
        ingredients.add(mozza);
        p_margarita = new PizzaComposee("margarita", "la simplissité culinaire", ingredients);
        ingredients = new ArrayList<>();
        ingredients.add(tomate);
        ingredients.add(aubergines);
        ingredients.add(jambon);
        ingredients.add(epinards);
        ingredients.add(mozza);
        ingredients.add(gogonzola);
        p_duchef = new PizzaComposee("du chef", "l'équilibre des saveurs du chef", ingredients);
        ingredients = new ArrayList<>();
        ingredients.add(tomate);
        ingredients.add(scampis);
        ingredients.add(mozza);
        p_mariniere = new PizzaComposee("marinière", "les saveurs de l'océan", ingredients);
```

L'interface `Ingredients`

L'interface `Ingredients` fournie contient la liste des constantes de type `Ingredient` de l'application.

Les différents cas d'utilisation

L'analyse a relevé dans un premier temps les cas d'utilisation suivants :

- *Enregistrer un client* qui permet d'enregistrer un nouveau client dans le système.
- *Passer commande* qui se décompose en plusieurs sous-fonctions :

- *Créer commande* qui instancie une nouvelle commande pour le client (= commande en cours du client) ;
- *Ajouter des pizzas* à une commande qui permet l'ajout de pizzas à la commande en cours du client ;
- *Créer des pizzas composables* qui permet au client de créer des pizzas composables ;
- *Valider commande* qui permet de clôturer la commande en cours qui devient donc une commande passée.

L'instance de `GestionPizzeria` retient la liste des clients enregistrés. Elle implémente une méthode qui permet d'enregistrer un client (1er cas d'utilisation relevé) dont voici la signature:

```
public Client enregistrerClient(String nom, String prenom, String telephone)
```

Ce constructeur crée une nouvelle instance du client ; l'attribution du numéro est séquentielle (dépend du nombre de clients).

Elle implémente ensuite les méthodes suivantes qui répondent aux sous-fonctions détaillées ci-dessus :

```
        /**
         * Crée une commande
         *
         * @param client
         *            le client pour qui la commande est créée; ce client appartient
         *            à la liste des clients
         * @return la commande créée pour le client sinon null (pas d'exception)
         */
        public Commande creerCommande(Client client) {    }

        /**
         * Ajoute à la commande en cours du client la pizza en quantité indiquée
         *
         * @param client
         *            le client qui a une commande en cours à qui on ajoute la pizza;
         *            ce client appartient à la liste des clients
         * @param pizza
         *            la pizza à rajouter à la commande en cours du client ; la pizza
         *            appartient à la carte des pizzas
         * @param quantite
         *            le nombre de fois qu'on ajouter la pizza à la commande en
         *            cours du client
         * @throws NoCommandeEnCoursException
         *             si le client n'a pas de commande en cours
         * @throws IllegalArgumentException si la quantité est négative ou nulle
         */    public void ajouterALaCommande(Client client, Pizza pizza, int quantite)
throws NoCommandeEnCoursException { }

        /**
         * Crée une pizza composable pour un client à partir des ingrédients. Ajoute
         * ensuite cette pizza à la commande en cours du client en quantité passée
         * en paramètre.
         *
         * @param client
         *            le client qui a une commande en cours qui crée un pizza
         *            composable ; ce client appartient à la liste des clients
         * @param quantite
         *            le nombre de fois qu'on ajoute la pizza composabe à la
         *            commande en cours du client
         * @param ingredients
         *            les ingrédients qui composent la pizza composable du client ;
```

```
         *            la liste contient au moins un ingrédient et tous les ingrédients
         *            de la liste sont dans Ingrédients
         * @throws NoCommandeEnCoursException
         *             si le client n'a pas de commande en cours
         * @throws IllegalArgumentException si la quantité est négative ou nulle
         */
        public void creerPizzaComposable(Client client, int quantite, Ingredient...
ingredients) throws NoCommandeEnCoursException {  }

        /**
         * Crée une pizza composable pour un client à partir des ingrédients. Ajoute
         * ensuite cette pizza à la commande en cours du client.
         *
         * @param client
         *            le client qui a une commande en cours qui crée un pizza
         *            composable ;  ce client appartient à la liste des clients
         * @param ingredients
         *            les ingrédients qui composent la pizza composable du client ;
         *            la liste contient au moins un ingrédient et tous les ingrédients
         *            de la liste sont dans Ingrédients
         * @throws NoCommandeEnCoursException
         *             si le client n'a pas de commande en cours
         */
        public void creerPizzaComposable(Client client, Ingredient... ingredients) throws
NoCommandeEnCoursException { }

        /**
         * Valide la commande en cours du client; la commande devient passée et
         * n'est donc plus en cours.
         *
         * @param client
         *            le client qui valide sa commande ; ce client appartient à
         *            la liste des clients
         * @throws NoCommandeEnCoursException
         *             si le client n'a pas de commande en cours
         */
        public void validerCommande(Client client) throws NoCommandeEnCoursException { }
```

Constatez qu'il y a deux méthodes `creerPizzaComposable` avec des types de paramètre différents. Comment appelle-t-on ce principe en java ?

Réponse :

Les tests

Une classe de tests `Main` est fournie. Examinez-la et placez-là au bon endroit.

Il faudra certainement adapter votre code pour observer un résultat en console. Ne changez pas le code de tests mais vos classes à vous !