Programmation Orientée Objet
IG2I, LE2 – 2023-2024

TP1 – Courses de F1

Diego Cattaruzza, Maxime Ogier, Pablo Torrealba

Contexte et objectif du TP

Dans ce premier TP, nous nous intéressons à la modélisation et la gestion d'une écurie de Formule 1. Plus particulièrement, nous nous intéresserons aux voitures et au personnel de l'écurie. Au cours de ce TP, nous allons donc créer des objets pour gérer une écurie de Formule 1.

Ce TP permet ainsi d'illustrer les notions suivantes :

- les bases du langage Java ;
- les notions de classe, d'attribut, de constructeur et de méthodes ;
- les notions d'encapsulation et de visibilité ;
- les classes composites et composants ;
- les associations 1-1.

On s'attachera en particulier dans ce TP à commenter proprement le code et à garantir au maximum que le code fonctionne et respecte les spécifications données (même dans le cas où l'utilisateur, lui, ne respecte pas ces spécifications).

Bonnes pratiques à adopter

- Testez votre code au fur et à mesure de votre avancement.
- Respectez les conventions de nommage Java.
- Faites des indentations de manière correcte.
- Donnez des noms compréhensibles et pertinents à vos variables, attributs, classes, méthodes.
- Respectez autant que possible le principe d'encapsulation.
- Utilisez des structures de données adaptées.
- Faites des méthodes courtes avec peu de niveaux d'indentation.
- Ne faites pas de duplication de code, mais privilégiez le code réutilisable.
- Utilisez les outils de votre IDE pour faire du *refactoring* (clic-droit puis '*Refactor*' sur IntelliJ IDEA).
- Utilisez la Javadoc pour comprendre le fonctionnement des méthodes que vous appelez.
- Quand c'est nécessaire, commentez votre propre code de façon pertinente au format Javadoc (commentaire commençant par `"/**"` au-dessus des définitions des attributs et méthodes).

1 Création des voitures et personnel d'une écurie de Formule 1

1.1 Rappels sur les objets et les classes

Pour rappel, une *classe* est la modélisation d'un type de données définissant à la fois son état (comment représenter la donnée) et son comportement (les méthodes permettant la manipulation de la donnée). Ceci signifie qu'une classe est un type d'*objet*, c'est un "moule" à objets. On peut ainsi produire ou créer un exemplaire à partir du moule. On appelle cet exemplaire une *instance* de la classe. Un objet est donc une instance d'une classe.

L'état d'un objet est caractérisé par la valeur de ses *attributs*. Son comportement est défini à travers les *méthodes* spécifiées par sa classe.

Un attribut est une variable servant à mémoriser une caractéristique d'un objet. Ils sont précisés dans la classe et chaque instance de la classe aura son propre exemplaire, avec une valeur potentiellement différente d'un objet à l'autre.

1.2 Diagramme de classe

UML (*Unified Modeling Language*) est un langage visuel pour la spécification et la conception d'un système d'informations. C'est un langage couramment utilisé en conception orientée objet. En particulier, un des types de diagramme d'UML est le *diagramme de classe*, qui permet de représenter les classes intervenant dans un système, ainsi que les liens entre ces classes.

1.3 Création de la classe Moteur

Nous nous intéressons dans un premier temps à la représentation d'un moteur. Le responsable technique vous explique qu'un moteur est défini par son carburant (qui peut être essence ou diesel) ainsi que par sa puissance (en *ch*). Par ailleurs, le responsable technique explique que les moteurs avec lesquels il

travaille ont toujours une puissance supérieure à 600 *ch*. Par défaut, on considérera que les moteurs sont à essence avec la puissance minimale. Par ailleurs, le responsable technique ajoute qu'il est possible de modifier la puissance d'un moteur mais absolument impossible de changer son carburant.

Question 1. Représentez sous forme de diagramme de classe un moteur avec ses attributs.

Question 2. Après avoir ouvert InteliJ IDEA, créez un nouveau projet **Formule1**, un paquetage **modele**, et écrivez une classe **Moteur** qui représente le moteur d'une voiture, dont vous définirez les attributs. Commentez chacun des attributs sous format Javadoc.

Question 3. Ajoutez un constructeur par défaut, un constructeur par données et un constructeur par copie. N'oubliez pas que les assistants IntelliJ IDEA sont là pour vous aider (clic-droit puis *Generate...*). Commentez chacun de ces constructeurs au format Javadoc.

Question 4. Complétez la classe **Moteur** avec les accesseurs et mutateurs nécessaires et une méthode **toString()** qui renvoie une chaîne de caractères avec les informations sur le moteur (en fait il s'agit d'une redéfinition de la méthode **toString()** de la classe **Object**). N'oubliez pas que les assistants IntelliJ IDEA sont là pour vous aider (clic-droit puis *Generate...*). Commentez au format Javadoc les méthodes que vous venez d'écrire.

Question 5. Écrivez une méthode principale (**public static void main(String[] args)**) afin de tester l'utilisation de la classe **Moteur**. Assurez-vous de tester tous les cas possibles et de veiller à ce que les préconisations sur la puissance et le carburant soient bien toujours respectées. N'oubliez pas que les assistants IntelliJ IDEA sont là pour vous aider (tapez "*psvm*" puis *Entrée*, ou bien "*sout*" puis *Entrée*) !

1.4 Création de la classe Voiture

Maintenant que nous sommes capables de créer des moteurs, nous allons nous attaquer aux voitures. Le responsable du parc de véhicules de l'écurie de Formule 1 vous explique que chaque voiture est définie par un identifiant unique (à savoir le numéro d'immatriculation), ainsi qu'une marque. Et bien entendu, une voiture est forcément composée d'un unique moteur. Par ailleurs, le responsable stipule qu'il n'est pas possible de modifier l'immatriculation ou la marque d'une voiture. Par contre, il est possible de remplacer son moteur.

Question 6. Modifiez le diagramme de classe afin d'y rajouter la classe **Voiture** et le lien avec la classe **Moteur**.

Question 7. Dans le paquetage **modele**, ajoutez une classe **Voiture** avec ses attributs. Comment gérer l'identification unique de chaque voiture ? Comment gérer la relation entre une voiture et un moteur ?

Question 8. Ajoutez les constructeurs suivants :

- par défaut ;
- par donnée de la marque ;
- par données de la marque, de la puissance et du carburant du moteur ;
- par données de la marque et d'un moteur.

Question 9. Ajoutez les accesseurs et mutateurs nécessaires et une méthode **toString()** qui renvoie une chaîne de caractères avec les informations sur la voiture. Commentez votre code si n'est pas déjà fait... Écrivez une méthode principale afin de tester l'utilisation de la classe **Voiture**. Testez entre autre le code suivant :

```
1  Moteur m1 = new Moteur(null);
2  Moteur m2 = new Moteur(1000, 'E');
3  Voiture v1 = new Voiture(null, m1);
4  Voiture v2 = new Voiture("Ferrari", m2);
5  Voiture v3 = new Voiture("Ferrari", m2);
6  System.out.println("v1 : " + v1.toString());
7  System.out.println("v2 : " + v2.toString());
8  System.out.println("v3 : " + v3.toString());
9  m2.setPuissance(1200);
10 System.out.println("v2 : " + v2.toString());
11 System.out.println("v3 : " + v3.toString());
```

1.5 Création de la classe Personne

Pour constituer une bonne écurie de Formule 1, il faut de bons véhicules, mais aussi une bonne équipe de personnes pour conduire et réparer les voitures. Nous nous intéressons donc à présent à la création des objets de type **Personne**. Pour le responsable de l'équipe, il est important que chaque personne puisse être associée à un numéro d'identification unique, et que l'on connaisse son nom, son prénom et son adresse.

Question 10. Modifiez le diagramme de classe afin d'y rajouter la classe **Personne**.

Question 11. Ajoutez une classe **Personne** avec ses attributs. Écrivez un constructeur par défaut et un constructeur par données. Ajoutez les accesseurs et mutateurs nécessaires, ainsi qu'une méthode **toString()** qui renvoie une chaîne de caractères avec les informations sur la personne. Commentez votre code et écrivez une méthode principale afin de tester l'utilisation de la classe **Personne**.

2 Association 1-1 unidirectionnelle : Personne - Voiture

Dans les équipes de Formule 1, il existe quelques règles pour associer les voitures et les personnes :

- une personne peut éventuellement conduire une voiture (dans la vie réelle c'est difficile de conduire deux voitures à la fois...) ;
- et évidemment une voiture possède éventuellement un conducteur (là aussi, avez-vous jamais vu une voiture conduite par deux personnes en même temps ?).

Nous supposerons ici que chaque personne peut accéder à la voiture qu'elle conduit, mais que les voitures ne peuvent pas accéder à leur conducteur. Ainsi, cela se traduira par une relation unidirectionnelle, navigable dans un seul sens.

On pourrait souhaiter réaliser une association bidirectionnelle, navigable dans les deux sens. Cependant, il faut bien noter que les associations bidirectionnelles sont plus complexes à coder car pour assurer la cohérence de la relation il faut mettre à jour les attributs dans les deux classes correspondantes.

Question 12. Modifiez le diagramme de classe afin d'y rajouter la relation unidirectionnelle entre les voitures et les personnes.

Question 13. Modifiez les classes **Voiture** et **Personne** en y ajoutant les attributs nécessaires afin de modéliser la relation entre ces deux classes. Modifiez si besoin les constructeurs de ces classes (on supposera qu'à l'initialisation d'une personne elle ne conduit pas de voiture). Modifiez si besoin les méthodes **toString** afin de prendre en compte les attributs que vous avez ajoutés. Testez vos modifications.

Question 14. Ajoutez dans la classe **Personne** une méthode **public boolean estPieton()** qui indique si la personne est à pied ou non.

Pour le moment, on n'a pas écrit de mutateur pour la voiture conduite par une personne car il faut gérer de manière plus fine la relation entre les deux classes. Nous définissons donc des règles métiers *affectation* et *restitution* d'une voiture pour bien gérer l'association Personne-Voiture. Pour pouvoir affecter une voiture à une personne, nous appliquerons les règles suivantes :

- la voiture doit exister ;
- la personne doit être un piéton (pas encore de voiture affectée).

Si ces deux conditions sont vérifiées, alors on affecte la voiture à la personne.

Question 15. Dans la classe **Personne** définissez une méthode **public boolean affecter(Voiture v)** qui gère l'affectation de la voiture **v** à la personne. Cette méthode renvoie un booléen qui indique si l'affectation a pu être effectuée correctement ou non. Faites des tests pour vérifier que tout se déroule correctement. En particulier, vérifiez ce qui se passe dans les cas suivants :

- sur une personne, on affecte une voiture qui vaut **null** ;
- sur une personne qui conduit déjà une voiture, on affecte une voiture ;
- on affecte une même voiture à deux personnes différentes.

Maintenant que nous savons affecter des voitures à des personnes, il faut qu'une personne puisse restituer la voiture qu'elle conduit.

Ainsi, pour pouvoir restituer la voiture d'une personne, on s'assure juste que la personne n'est pas piéton. Si tel est le cas, on restitue la voiture, et la personne devient donc piéton.

Question 16. Dans la classe **Personne** définissez une méthode **public boolean restituer()** qui restitue la voiture conduite par la personne. Cette méthode renvoie un booléen qui indique si la restitution a pu être effectuée correctement ou non. Faites des tests pour vérifier que tout se déroule correctement. Notamment, vérifiez qu'après avoir restitué une voiture, il est possible d'affecter une nouvelle voiture à la personne.

3 Association bidirectionnelle

Vous avez dû remarquer que l'association unidirectionnelle entre une personne et une voiture pouvait poser quelques problèmes :

- il est possible qu'une même voiture soit affectée à plusieurs personnes ;
- une voiture ne peut pas avoir accès à son conducteur.

Pour remédier à cela, il faudrait mettre en place une association bidirectionnelle.

Question 17. Modifiez la classe **Voiture** en y ajoutant les attributs nécessaires afin de modéliser la relation bidirectionnelle entre les deux classes. Modifiez en conséquence les constructeurs de la classe **Voiture**. Modifiez également la méthode **toString** de cette classe afin de prendre en compte les nouveaux attributs. Testez vos modifications.

Question 18. Ajoutez dans **Voiture** une méthode **public boolean estDisponible()** qui indique si la voiture est disponible (elle n'a pas de conducteur) ou non.

Pour rappel, nous avons choisi de ne pas utiliser de mutateurs pour la voiture et son conducteur car il faut gérer de manière plus fine la relation entre les deux classes. En effet, il ne faudrait pas se retrouver avec, par exemple, une personne p qui conduit une voiture v, mais cette voiture v n'aurait aucun conducteur affecté ou bien un autre conducteur p' ! Pour gérer la relation bidirectionnelle, on définit une classe "maître" qui va gérer en priorité cette relation. Ici nous choisissons par exemple la classe **Personne**. Dans cette classe, nous définissons des règles métiers *affectation* et *restitution* d'une voiture pour bien gérer l'association bidirectionnelle Personne-Voiture. Ainsi, dans le cas de l'association bidirectionnelle, pour pouvoir affecter une voiture à une personne :

- la voiture doit exister ;
- la voiture doit être disponible (pas encore de conducteur affecté à la voiture) ;
- la personne doit être un piéton (pas encore de voiture affectée).

Si ces trois conditions sont vérifiées alors :

- on affecte la voiture à la personne ;
- et on définit la personne comme étant conducteur de la voiture.

Ainsi, du côté de la voiture, lorsqu'on lui affecte un conducteur, ce conducteur doit déjà être déclaré comme conducteur de la voiture. Et donc pour affecter un conducteur à une voiture :

- la personne doit exister ;
- la voiture doit être disponible (par encore de conducteur affecté) ;
- le conducteur doit conduire la voiture à laquelle on l'affecte (on peut tester l'égalité des immatriculations).

Question 19. Proposez un diagramme de séquence pour l'affectation d'une voiture à une personne. Vérifiez que la relation entre les classes **Personne** et **Voiture** est toujours préservée.

Question 20. Dans la classe **Personne** modifiez la méthode **public boolean affecter(Voiture v)** afin qu'elle gère à présent l'association bidirectionnelle. N'hésitez pas à faire des affichages dans vos méthodes pour voir ce qui se passe. Faites des tests pour vérifier que tout se déroule correctement. En particulier, vérifiez ce qui se passe dans les cas suivants :

- sur une personne, on affecte une voiture qui vaut **null** ;
- sur une personne, on affecte une voiture qui a déjà un conducteur ;
- sur une personne qui conduit déjà une voiture, on affecte une voiture ;
- sur une voiture, on met un conducteur qui vaut **null** ;
- sur une voiture, on met un conducteur qui n'a pas de voiture affectée ;
- sur une voiture qui a déjà un conducteur affecté, on met un conducteur ;
- sur une voiture, on met un conducteur qui conduit déjà une autre voiture.

Maintenant que nous savons affecter des voitures à des personnes avec l'association bidirectionnelle, il nous faut pouvoir restituer une voiture, tout en conservant bien la relation entre les deux classes, i.e. si la personne n'a plus de voiture, alors la voiture n'a plus de conducteur. De même que pour l'affectation, nous allons supposer ici que la classe **Personne** est la classe "maître" qui va gérer en priorité la restitution.

Ainsi, pour pouvoir restituer la voiture d'une personne, on s'assure d'abord que la personne n'est pas piéton. Puis on restitue la voiture, et la personne devient donc piéton, et on demande à la voiture de libérer son conducteur. Du côté de la voiture, pour libérer le conducteur, on s'assure que la voiture n'est pas disponible (elle a bien un conducteur), et que le conducteur de la voiture est piéton. Alors dans ce cas on rend la voiture disponible.

Question 21. Proposez un diagramme de séquence pour la restitution d'une voiture par une personne. Vérifiez que la relation entre les classes **Personne** et **Voiture** est toujours préservée.

Question 22. Dans la classe **Personne** modifiez la méthode **public boolean restituer()** afin qu'elle prenne en compte l'association bidirectionnelle. Faites des tests pour vérifier que tout se déroule correctement.

4 Quelques notions utiles

4.1 Classes composites et composantes

Le composant est un objet inclus dans un autre objet, sauvegardé en tant que type valeur, et non en tant que référence entité. Dans le cas de notre TP, le **Moteur** est donc un composant de la classe **Voiture**. La classe **Voiture** est dit composite parce qu'elle contient une composante. Quand on construit une classe composite, il faut bien veiller à construire un nouvel objet composant. Par exemple :

```java
public class Composant {
   // Attributs
   private int compAttr1;
   private char compAttr2;

   // Constructeur par défaut
   public Composant () {
      compAttr1 = 0;
      compAttr2 = ' ';
   }

   // Constructeur par données
   public Composant (int compAttr1, char compAttr2) {
      this.compAttr1 = compAttr1;
      this.compAttr2 = compAttr2;
   }
}
```

```java
public class Composite {
   // Attributs
   private int attr1;

   // On s'assure de toujours bien construire l'objet
   private Composant comp = new Composant();

   // Constructeur par défaut
   private Composite () {
      attr1 = 0;
      comp = new Composant();
   }

   // Constructeur par données
   private Composite (int attr1, int compAttr1, char compAttr2) {
      this.attr1 = attr1;
      comp = new Composant(compAttr1, compAttr2);
   }
}
```