Chapitre 3

Diagrammes de classes complexes

Ce chapitre complète quelques points de théorie sur le langage Java
- classes et méthodes abstraites
- interfaces
- attributs et méthodes statiques

On envisage ensuite quelques exemples concrets.

3.1 Description d'un programme plus complet

Dans ce chapitre, nous introduirons quelques nouvelles notions, mais dans le cadre d'un projet plus complet. Le contexte sera celui d'une écurie de rallye qui possède plusieurs voitures. Le but du programme est de parvenir à estimer le budget à prévoir pour une course dont on connaît la durée et la longueur. On pourra ainsi comparer les coûts de différentes épreuves afin d'opérer des choix.

1. l'écurie se compose de quatre voitures
2. chaque voiture a un mécanicien et deux pilotes.
3. l'écurie dispose également de deux camions remplis d'outils et de pièces de rechange. Ces camions ont été achetés et remplis en début de saison, grâce à l'aide d'un sponsor
4. un mécanicien supplémentaire est attribué à chaque camion.
5. deux voitures sont attachées à chaque camion.
6. les pilotes sont payés à la journée (2000 euros) avec une prime de 5 euros au kilomètre. Ces chiffres peuvent varier pour un pilote vedette.
7. les coûts de remplacement des pièces utilisées pour les épreuves dépendront des incidents et accidents, ils ne peuvent donc pas être pris en compte.
8. un mécanicien gagne 250 euros par jour s'il est attaché à un camion et le double s'il est attaché à une voiture.
9. on compte 2 euros le litre pour les voitures, 30 litres au 100 km.
10. il y a une prime d'assurance de 1000 euros par jour et par véhicule (2000 pour les voitures).

J'ai mis de côté le calcul de l'usure des pneus et d'autres facteurs qu'on pourrait facilement ajouter, mais qui compliqueraient inutilement le programme final, qui reste un exercice d'école.

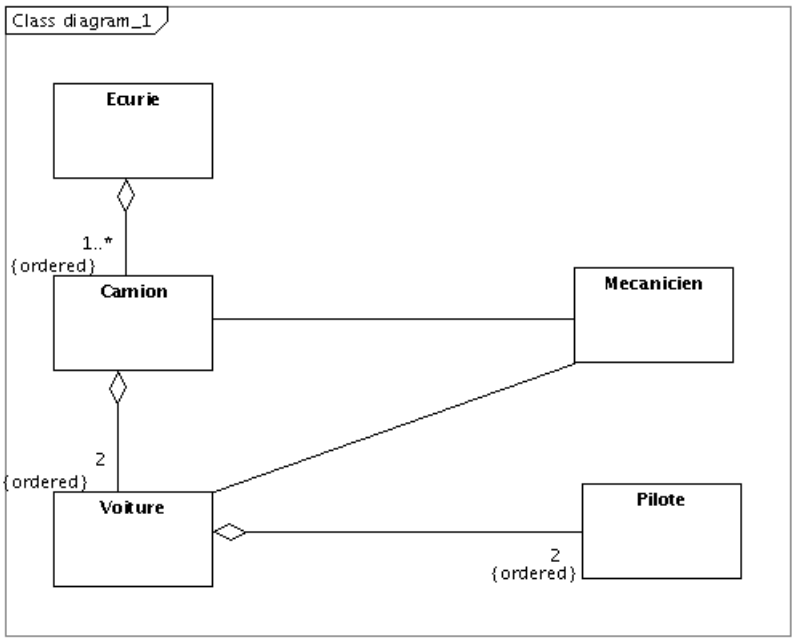

FIG. 3.1 – Associations entre les classes principales

3.2 Classes abstraites

Le problème présenté met en évidence quatre classes principales : les pilotes, les mécaniciens, les voitures et les camions. Une première approche permet d'envisager les associations entre les classes. J'y fais figurer des associations un peu particulière, dites *agrégations*, qui manifestent le fait que l'écurie *comporte* deux camions, le camion deux voitures et chaque voiture deux pilotes .

On voit néanmoins qu'une telle modélisation ne rend pas clairement l'idée qu'un mécanicien est attaché soit à une voiture, soit à un camion. On pourrait ajouter une contrainte. Mais nous verrons dans un moment une autre solution.

Lorsqu'on examine les quatre classes principales, on remarque qu'elles se groupent par deux de manière naturelles : camions et voitures sont des véhicules, alors que les mécaniciens et les pilotes sont des personnes .

Comme on ne trouve aucune personne, ni aucun véhicule dans la description des spécifications, nous allons déclarer les classes *Personne* et *Véhicule* abstraites. Cela nous empêchera d'instancier des objets pour ces deux classes. En outre, le fait que la méthode `ajouterMecano()` est abstraite nous impose de rendre *Véhicule* abstraite. La généralisation obtenue permettra de centraliser une partie de la gestion des personnes et des véhicules dans les classes abstraites ainsi définies. On pourra doter les personnes d'adresses, de numéros de téléphones, de caractéristiques sociales et juridiques. Les véhicules pourront recevoir des responsabilités particulières en rapport notamment avec les obligations d'immatriculation, de contrôle technique etc... On remarquera que le modèle est ici simplifié au maximum. Je n'ai pas prévu de primes d'assurances pour les personnes.

La présence d'une classe *Véhicule* nous permet de revoir notre modélisation des associations où interviennent les mécaniciens. En reliant le mécanicien au véhicule, on impose que chaque véhicule ait son mécanicien et que chaque mécanicien soit relié exactement soit à une voiture, soit à un camion .

Nous allons maintenant examiner chacune des classes (j'expliquerai à quoi correspond la

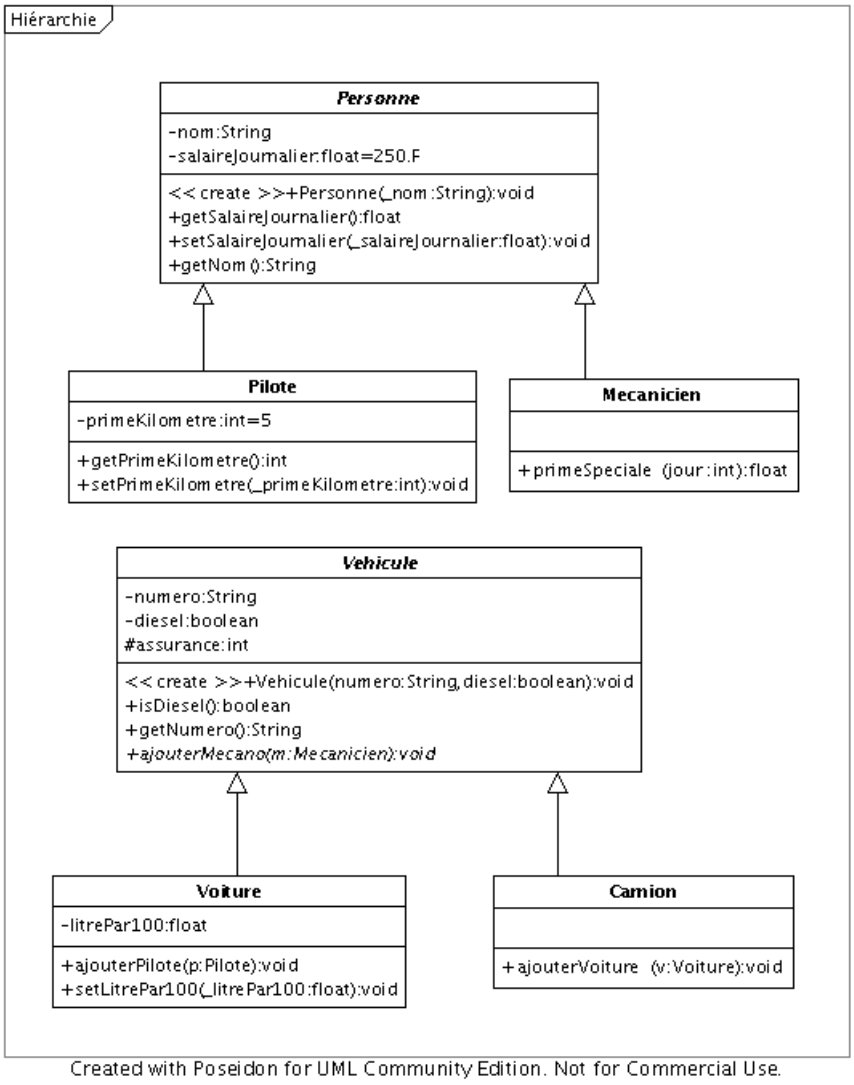

FIG. 3.2 – Hiérarchie de classes

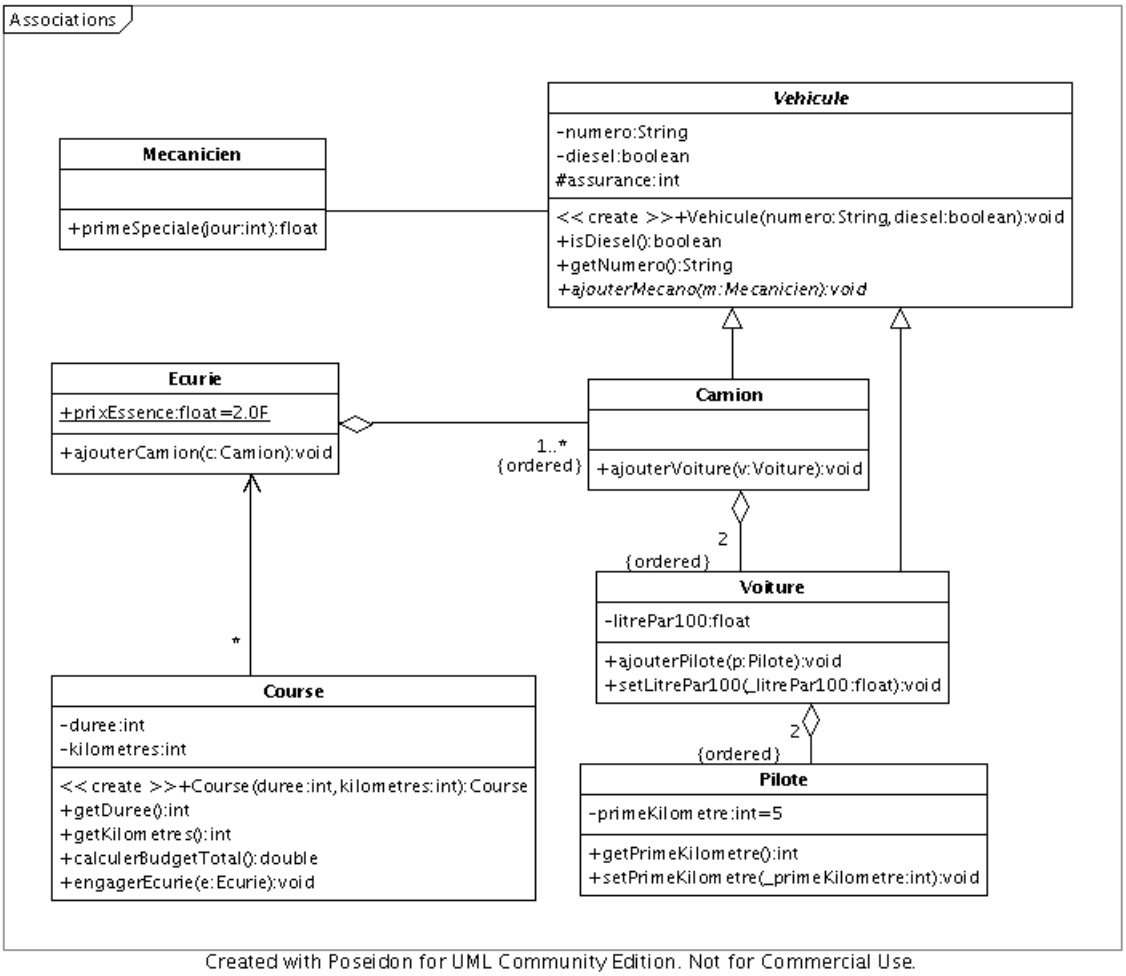

FIG. 3.3 – Associations, version finale

clause `implements` dans la section suvante).

3.2.1 Superclasses abstraites

```
package rallye;

public abstract class Vehicule implements Participant {
    // le numéro du véhicule
    private String numero;
    private boolean diesel;
    private float cylindree;
    protected double assurance=1000;
    public Mecanicien mecanicien;

   // le créateur initialise le numéro et le type de carburant
   public  Vehicule(String numero, boolean diesel) {
        this.numero=numero;
        this.diesel=diesel;
    }
    // Différents accesseurs
    public float getCylindree() {
        return cylindree;
    }
    public boolean isDiesel() {
        return diesel;
    }
    public String getNumero() {
        return numero;
    }
    // méthode d'ajout du mécanicien à spécifier
    public abstract void ajouterMecano(Mecanicien m);

    // surcharge d'une méthode de Object pour affichage élégant de l'objet
    public String toString() {
        return this.getClass().getName()+"␣"+getNumero();
    }

 }

package rallye;

public abstract class Personne implements Participant {
    private String nom;
    private float salaireJournalier=250.F;

    // le créateur initialise le nom
    public  Personne(String _nom) {
        nom = _nom;
    }

    // accesseurs
    public float getSalaireJournalier() {
```

```
        return salaireJournalier;
    }
    public void setSalaireJournalier(float _salaireJournalier) {
        salaireJournalier = _salaireJournalier;
    }
    public String getNom() {
        return nom;
    }
    // la méthode toString sera surdéfinie dans les sous-classes
}
```

3.2.2 Sous-classes[1]

```
package rallye;
import java.util.ArrayList;
public class Camion extends Vehicule {

    // permet d'atteindre les voitures du camion (voir association)
    public ArrayList<Voiture> voiture = new ArrayList<Voiture>();
    // permet d'atteindre l'écurie à laquelle le camion appartient
    public Ecurie ecurie;

    // le créateur crée un véhicule au diesel
    public Camion(String numero){
        super(numero, true);
    }

    public void ajouterVoiture(Voiture v) {
        voiture.add(v);
        // ne pas oublier le double lien (navigabilité double)
        v.camion=this;
        ecurie.participants.add(v);// dans la version finale
    }

    public void ajouterMecano(Mecanicien m) {
        mecanicien=m;
        // ne pas oublier le double lien (navigabilité double)
        m.vehicule=this;
        ecurie.participants.add(m);// dans la version finale
    }
 }

package rallye;
import java.util.ArrayList;

public class Voiture extends Vehicule {

    private float litrePar100=30;
    // Liste des pilotes
```

[1] Les instructions suivies du commentaire `//dans la version finale` peuvent être ignorées lors d'une première lecture.

```
        public ArrayList<Pilote> pilote = new ArrayList<Pilote>();
        // camion dont la voiture dépend
        public Camion camion;

        // le créateur crée un véhicule à essence
        // et augmente la prime d'assurance
        public Voiture(String numero) {
            super(numero, false);
            assurance=2000;
        }

        public void ajouterPilote(Pilote p) {
            pilote.add(p);
            // ne pas oublier le double lien (navigabilité double)
            p.voiture=this;
            camion.ecurie.participants.add(p);// dans la version finale
        }

        public void ajouterMecano(Mecanicien m) {
            mecanicien=m;
            // ne pas oublier le double lien (navigabilité double)
            m.vehicule=this;
            camion.ecurie.participants.add(m); // dans la version finale
        }

        public void setLitrePar100(float _litrePar100) {
            litrePar100 = _litrePar100;
        }

    }
    package rallye;

    public class Pilote extends Personne {

        // montant de la prime au kilomètre
        private int primeKilometre=5;
        // voiture pilotée
        public Voiture voiture;

        // le constructeur crée une personne et
        // augmente son salaire
        public Pilote(String _nom){
            super(_nom);
            setSalaireJournalier(2000);
        }

        // accesseurs**
        public int getPrimeKilometre() {
            return primeKilometre;
        }
        public void setPrimeKilometre(int _primeKilometre) {
            primeKilometre = _primeKilometre;
        }
```

```
    public String toString(){
        return "Pilote␣"+getNom()+"␣sur␣"+voiture;
    }

}

package rallye;

public class Mecanicien extends Personne {

    // le véhicule sera soit un camion soit une voiture
    public Vehicule vehicule;

    public Mecanicien(String _nom){
        super(_nom);
    }

    // prime spéciale du mécanicien s'il est attaché à une voiture
    // on examine la classe réelle du véhicule
    public float primeSpeciale(int jour) {
        if(vehicule.getClass().getName()=="Voiture")
            return jour*500;
        else
            return 0.0F;
    }

    public String toString(){
        return "Mécanicien␣"+getNom()+"␣sur␣"+vehicule;
    }
}

package rallye;
import java.util.ArrayList;

public class Ecurie {

    // prix de la l'essence accesible à tous
    public static float prixEssence=2.0F;
    public ArrayList<Camion> camion = new ArrayList<Camion>();
    // ajouté pour la version finale
    public ArrayList<Participant> participants = new ArrayList<Participant>();



    public void ajouterCamion(Camion c) {
        camion.add(c);
        // ne pas oublier le double lien (navigabilité double)
        c.ecurie=this;
        participants.add(c); // dans la version finale
    }    }

package rallye;
```

```
public class Course {

    private int duree;
    private int kilometres;
    public Ecurie ecurie;

    public  Course(int jours, int kilometres) {
        this.duree=jours;
        this.kilometres=kilometres;
    }

    // Cette méthode utilise déjà l'interface participant
    public double calculerBudgetTotal(){
        double total=0.;
        for(Participant p:ecurie.participants){
            total+=p.calculerCout(duree, kilometres);
        }
        return total;
    }

    public void engagerEcurie(Ecurie ecurie){
        this.ecurie=ecurie;
    }
}
```

3.3 Interfaces

L'un des problèmes pour réaliser le calcul du budget d'une course consiste à devoir prévoir une méthode de calcul qui tienne compte de la hiérarchie réalisée. On peut facilement calculer le coût d'un pilote, mais ce pilote est accessible à travers sa voiture, elle-même connue par le camion. On pourrait totaliser les coûts des pilotes, du mécanicien et de la voiture proprement dite au niveau de celle-ci, puis regrouper les coûts des voitures dépendant d'un camion, en y ajoutant les coûts propres du camion et de son mécanicien, pour finalement totaliser les deux camions et tout ce qui en dépend. Cette technique devrait fonctionner, mais avec quelle lourdeur.

Une solution plus élégante serait de regrouper tous les objets appartenant à l'écurie dans une structure plate, une liste, qu'on pourrait ensuite parcourir. Ce n'est malheureusement pas possible puisque nous avons quatre classes instanciées et deux superclasses abstraites. On pourrait évidemment regrouper tous nos participants comme instances d'`Object`, mais c'est peu efficace, parce que les objets ne savent rien faire d'intéressant du point de vue de l'écurie. Créer une classe abstraite regroupant les véhicules et les personnes reste une possibilité, mais cette superclasse ne correspond à rien d'intuitif. Il arrive d'ailleurs fréquemment que cette dernière possibilité n'existe même pas dans la mesure où les classes qu'on désire généraliser dérivent déjà d'une autre classe.

Ce que nous voulons faire en fait, c'est regrouper des objets hétérogènes afin de leur donner un comportement commun : la capacité de calculer leur coût. Cela correspond en Java à la notion d'*interface*. Une interface est une sorte de classe abstraite qui ne définit que des méthodes abstraites, sans spécifier ni variables d'instance, ni code. C'est en fait une sorte de contrat que toutes les classes qui prétendent la *réaliser* s'engagent à honorer.

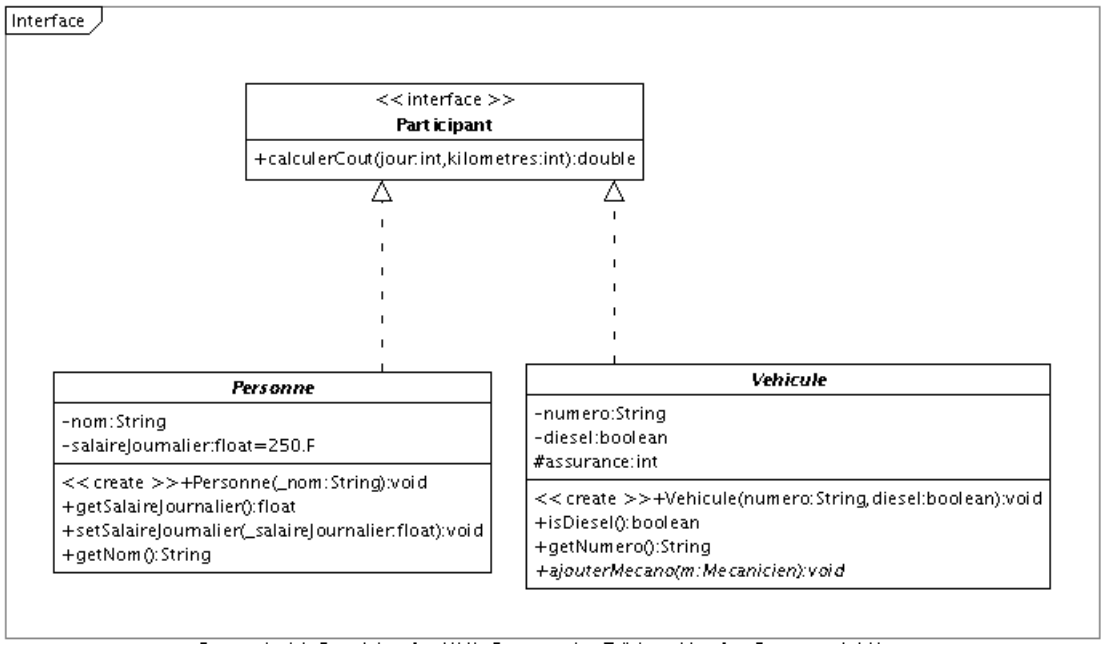

FIG. 3.4 – Interface Participant

Nous allons donc définir une interface `Participant` qui aura une seule méthode, `calculerCout()`. Les classe *Véhicule* et *Personne* s'engageront à la réaliser .

Comme les deux classes qui réalisent l'interface *Participant* sont des classes abstraites, elles peuvent choisir de surdéfinir les métodes de l'interface ou de charger leur sous-classe de la faire. Nous pourrons donc demander à la classe véhicule de calculer son budget (lié à la prime d'assurance). Ce calcul conviendra à la classe *Camion*, mais pas à la classe *Voiture* qui doit en plus prendre en compte sa consommation de carburant. Celle-ci va surdéfinir la méthode. Commme nous n'avons aucun moyen de calculer le budget d'une personne, nous passons le relais aux sous-classes *Pilote* et *Mécanicien*. Pour ce qui concerne la classe *Personne*, nous devons donc nous limiter à déclarer qu'elle réalisent l'interface `Participant`, au moyen de la clause `implements`. Pour *Véhicule*, nous ajouterons une surdéfinition de `caluler-Cout()`.

```
public abstract class Personne implements rallye.Participant {
...}
public abstract class Vehicule implements rallye.Participant {
...
   public double calculerCout(int jour, int kilometres) {
        return  jour*assurance;
    }

...}
```

Nous allons donc modifier trois des sous-classes en les dotant d'une méthode `calculer-Cout()` spécifique :

```
   // méthode surdéfinie du PILOTE
   public double calculerCout(int jour, int kilometres) {
        double montant=0.;
        montant+=jour*getSalaireJournalier();
        montant+=kilometres*getPrimeKilometre();
```

```
        return montant;
    }
    // méthode surdéfinie du MECANICIEN
    public double calculerCout(int jour, int kilometres) {
        double montant=jour*getSalaireJournalier();
        montant+=primeSpeciale(jour);
        return montant;
    }
    // méthode surdéfinie de la VOITURE
     public double calculerCout(int jour, int kilometres) {
        double montant=super.calculerCout(jour, kilometres);
        montant+=kilometres/100*Ecurie.prixEssence*litrePar100;
        return montant;
    }
```

On note que la méthode propre à la voiture utilise la méthode de sa classe mère pour calculer le montant de l'assurance (appel de `super.calculerCout()`).

Pour réaliser un parcours séquentiel des participants, il faut disposer d'un tableau ou d'une collection qui contienne tous les participants à l'écurie. Ce tableau a été conçu comme une variable d'instance publique de la classe *Ecurie*. On aurait pu aussi créer une méthode pour y accéder. L'instruction mystérieuse qui apparaissait dans toutes les méthodes dont le nom commence par `ajouter` ajoute un participant à la liste.

```
camion.ecurie.participants.add(x); // dans la version finale
```

On peut facilement parcourir cette liste par une instruction de type « for-each » :

```
for(Participant p :participants)
 {p.calculerCout(....)
```

Lorsqu'on utilise une variable dérivée d'une interface, on ne peut pas utiliser que les méthodes défines dans l'interface. Comme je désirais pouvoir utiliser facilement la méthode `toString()` lors d'un parcours de vérification, j'ai inclu cette opération dans l'interface.

3.4 Attributs et méthodes statiques

Nous pouvons revenir un moment sur le programme principal. Il se compose, pour les programmes Java autonomes, d'une méthode statique de type `void`, nommée `main()`. On peut considérer une telle méthode comme une procédure. Les méthodes statiques dépendent de la classe et non d'un objet. De la même manière, des attributs statitiques peuvent être définis : ils fonctionnent comme des variables globales à l'intérieur du programme. On pourrait aisément écrire un programme Java qui ne comporterait que des méthodes et des variables statiques (des procédures/fonctions et des variables globales). C'est évidemment une entorse à l'esprit de la programmation objet. L'API Java contient un certain nombre de méthodes statiques prédéfinies, notamment dans la classe `Math`. On peut les considérer comme des sortes de fonctions.

Une méthode statique ne peut évidemment pas accéder aux variables d'instances, dont il existe autant de copies qu'il y a d'objets instanciés, ni aux méthodes d'instances qui manipulent ces variables. Par contre, une méthode d'instance peut accéder à des variables statiques et à des méthodes statiques. Toute modification apportée à une variable statique est immédiatement partagée par toutes les instances.

Dans le programme de gestion des courses, j'ai, pour simplifier, défini deux méthodes statiques qui créent respectivement une écurie et une course. C'est une pratique courante, lorsque la création d'un objet est complexe, de déléguer cette création à une méthode statique. Quand la méthode statique permet de créer des objets différents selon les besoins, on les nomme des *factories*. Certaines classes n'ont d'autre but que de fournir des méthodes statiques permettant la création d'objet complexes.

```
package rallye;

public class Main {

    // cette variable et cette méthode statique permet l'affichage
    // de messages pendant la mise au point (initialiser DEBUG à true)
    public static final boolean DEBUG = false;
    public static void controle(String message){
        if(DEBUG) System.out.println(message); }

    /* Création d'une écurie */
    public static Ecurie creerEcurie(){
        // tableaux contenant les noms des personnes et
        // les numéros de différents véhicules
        String[] nomsMec={"Jean","Jacques","Jim","Jack","Jules","Jérome"};
        String[] nomPilotes = {"Pierre","Paul","Pascal","Pablo",
                "Serge","Sylvain","Sandro","Simon"};
        String[] numeroCamions = {"BFV674","LUJ800"};
        String[] numeroVoitures= {"BUI456","TTR587","SPA513","DNU917"};

        Ecurie ecurie = new Ecurie();
        // trois variables pour accéder au mécanicien/voiture/pilote suivant
        int m=0,v=0,p=0;
        // Création de deux camions
        for(int c=0;c<numeroCamions.length;c++){
            Camion camion= new Camion(numeroCamions[c]);
            controle("=======================================");
            controle("Camion : "+camion.getNumero());
            ecurie.ajouterCamion(camion);
            camion.ajouterMecano(new Mecanicien(nomsMec[m++]));
            controle(camion.mecanicien.toString());
            // Création de deux voitures
            for(int i=0;i<2;i++){
                Voiture v1=new Voiture(numeroVoitures[v++]);
                camion.ajouterVoiture(v1);
                controle("Voiture :"+v1);
                v1.ajouterMecano(new Mecanicien(nomsMec[m++]));
                controle("       "+v1.mecanicien);
                // création de deux pilotes
                for(int j=0;j<2;j++){
                    Pilote pilote=new Pilote(nomPilotes[p++]);
                    v1.ajouterPilote(pilote);
                    controle("       "+pilote);
                    // Jim gagne plus
                    if(pilote.getNom()=="Pablo"){
                        pilote.setSalaireJournalier(4000);
```

```
                        pilote.setPrimeKilometre(20);
                    }
                }
            }
        }
        controle("=========================================");
        // Demo de recherche de coÃ»t pour 10 jours 5000 km
        for(Participant p1:ecurie.participants){
            controle(p1.toString()+" coÃ»te "+p1.calculerCout(10,5000));
        }
        return ecurie;
    }

    public static Course creerCourse(){
        // lecture des paramètres de la course
        String sNbreJours=javax.swing.JOptionPane.
             showInputDialog("Nombre de jours");
        int nbreJours = Integer.parseInt(sNbreJours);
        String sNbreKilometres=javax.swing.JOptionPane.
             showInputDialog("Nombre de kilomètres");
        int nbreKilometres = Integer.parseInt(sNbreKilometres);
        Course course = new Course(nbreJours, nbreKilometres);
        return course;
    }

    public static void main(String[] args) {
        Ecurie ecurie = creerEcurie();
        Course course = creerCourse();

        course.engagerEcurie(ecurie);
        // affichage du coût de la course
        javax.swing.JOptionPane.showMessageDialog(
            null,"Cout total de la course:"+course.calculerBudgetTotal(),
            "Calcul",javax.swing.JOptionPane.INFORMATION_MESSAGE);

    }

}
```