Séquence 4

Assurer la cohérence des objets

1 Notion de cohérence

Si l'on cherche la définition de cohérence, un dictionnaire nous apprendra que c'est la qualité de ce qui est cohérent et cohérent au sens figuré signifie *dont les parties se tiennent et s'enchaînent avec ordre de manière à former un ensemble logique, harmonieux, satisfaisant pour l'esprit* (définition du dictionnaire Trésor de la Langue Française Informatisé).

Ce niveau de cohérence caractérise bien un programme entier : ils faut que tous les objets (qui sont les parties du programme) soient dans un rapport harmonieux. Mais ce dont nous allons traiter ici est un degré plus élémentaire de cohérence : la cohérence interne d'un objet ou d'une classe d'objets.

La cohérence ne peut pas concerner une seule entité, puisqu'il s'agit fondamentalement des propriétés d'une relation entre plusieurs choses. La cohérence qui nous intéresse, c'est la conformité entre un objet d'un programme Java et la réalité qu'il doit modéliser en tenant compte de l'objectif du programme et de ses possibles évolutions.

1.1 Relation entre objet Java et réalité

Un objet doit être présent dans un programme parce qu'il sert à quelque chose. Toutes les données du programme doivent être contenues dans des objets au sens Java. Ces données peuvent représenter des objets du monde physique : une pomme, une voiture, un téléphone ; mais aussi des êtres (animaux, êtres humains) ; mais également des entités purement abstraites (date, contrat de travail, abonnement) ou des abstractions liées à des objets physique (calibres de pommes, modèle de brosse à dent, numéro INSEE ou numéro d'immatriculation de véhicule, etc). Il y a aussi des objets dont le but est de faire fonctionner le programme dans son environnement informatique : son système d'exploitation, son réseau local, internet, etc. Par exemple l'objet System.out que nous utilisons pour afficher des messages à l'écran ne correspond pas à une réalité liée à ce que nous programmons : c'est un objet de service, utilisé dans la plupart de nos programmes.

Donc, il ne faut pas prendre trop littéralement la notion de *réalité* ou de *monde réel* que certain mettent en avant de façon un peu aventureuse.

> Règle de cohérence numéro 1 Lorsqu'un objet représente une réalité, l'objet Java doit être cohérent avec cette réalité.

Le corrolaire de cette règle est qu'il n'est pas possible de coder sans étudier précisément les réalités concernées.

1.2 Relation entre objet Java et finalité du programme

Certaines réalités sont extrêmement complexes. Prenons une voiture. On peut prendre une approche analytique : une voiture est composée d'un ensemble de pièces (en moyenne, 30 000 pièces). Mais ce n'est pas la seule approche possible : si je suis comptable, une voiture est une valeur, un actif soumis aux règles de l'ammortissement. Si je suis un vendeur de voitures, ce qui me concerne n'est pas trop un exemplaire, mais un modèle avec son prix public, la marge de négociation que j'ai, la prime qui me reviendra si je vends ce modèle, les délais de livraisons, les options et couleurs et surtout les service associés (assurance, crédit).

Si je suis l'État, ce qui m'intéresse c'est de pouvoir identifier le propriétaire et lui adresser sans retard les amendes à payer.

Aucun objet Java ne pourra enregistrer la totalité des points de vue sur une voiture. C'est donc la finalité du programme qui imposera un point de vue et la cohérence de l'objet Java devra se faire par rapport à ce point de vue qui devra si possible être parfaitement modélisé.

Une première approximation quant à la finalité d'un programme est obtenue avec l'identité du client : si le client est un marchand de pièces détachées, si c'est le service des amendes, si c'est le service commercial d'un constructeur cela nous dit quelque chose. On ne peut évidemment pas en rester là, il faut une description fine de la finalité du programme, de ses fonctionnalités, pour pouvoir faire un code qui remplit sa mission et donne satisfaction.

> Règle de cohérence numéro 2 Un objet représentant une réalité doit contenir le plus petit ensemble d'informations permettant de calculer les résultats attendus du programme.

Il ne faut rien mettre d'inutile, mais surtout, plus important, il ne faut rien oublier de nécessaire. Le corrolaire de cette règle c'est qu'il n'est pas possible de coder correctement sans savoir très précisément ce que le programme doit faire. Cela concerne le client qui a son mot à dire sur le sujet. Cela peut nécessiter l'aide d'un expert d'un autre champs disciplinaire. Par exemple, on ne peut pas faire un logiciel de comptabilité sans le concours d'un comptable et sans se poser la question de la conformité par rapport à la loi Française.

Un bémol à ajouter concernant la règlede cohérence numéro 2 : il faut anticiper les évolutions probables du logiciel et ne pas faire des choix qui rendraient ces évolutions difficiles. On peut mettre un peu plus que le strict minimum dans les objets pour rendre le logiciel plus évolutif.

2 Un exemple

Voyons un cas concret qui permettra d'étudier les problèmes de la cohérence. Il s'agit des scores de matchs de rugby. Au début d'un match, le score est de 0 à 0 (noté 0-0). Des actions de jeu permettent de marquer 3 points (drop ou pénalité), 5 points (essai non transformé) ou 7 points (essai transformé). Dans un score comme 12-7, on mentionne en premier le nombre de points de l'équipe qui joue à domicile et en second le nombre de points de celle qui joue à l'extérieur.

2.1 Discussion sur la finalité du programme

Même si un score de rugby est une réalité moins complexe qu'une voiture, il y a différents usages possible pour un objet le représentant. Par exemple, l'objet peut être mutable ou immutable. L'objet est créé une fois le match terminé et le score final ne changera plus jamais : les données contenues dans l'objet sont enregistrées par le constructeur et il n'y a aucun modificateur à prévoir dans la classe. C'est un objet immutable, comme une chaîne de caractère Java.

Si au contraire, l'objet est créé avant le début du match, le constructeur mettra le score à 0-0 et il y aura des modificateurs pour enregistrer les variations de score. Il s'agit cette fois d'un objet mutable, avec des modificateurs qui permettent de changer le score.

Et dans ce deuxième cas, on n'a pas encore épuisé la question : il faut savoir comment vient l'information. Est-ce qu'il y a un informateur qui va enregistrer tous les évènements du match (par exemple, drop pour l'équipe qui reçoit, puis essai non transformé pour l'équipe qui se déplace) ? Ou alors, y a-t-il une mise à jour périodique du score courant (à la 47ème minute, il y a 15-9) ? Ces deux possibilités sont réalistes et correspondent à des méthodes de catégorie *modificateur* différentes.

Pour la suite de l'exemple, nous allons supposer que chaque évènement de match changeant le score est enregistré.

2.2 Première ébauche

Dans une première ébauche, nous allons déclarer deux attributs de type int pour contenir respectivement le nombre de points de l'équipe qui reçoit et de l'équipe qui se déplace.

A partir de ce seul postulat, je fais générer automatiquement à éclipse le code suivant (respectivement avec clic-droit->source->generate constructor from fields et clic-droit->source->generate getters and setters).

```
package rugby;

public class RugbyScore {

    private int pointsEquipeDomicile,pointsEquipeExterieur;

    public RugbyScore(int pointsEquipeDomicile, int pointsEquipeExterieur) {
        super();
        this.pointsEquipeDomicile = pointsEquipeDomicile;
        this.pointsEquipeExterieur = pointsEquipeExterieur;
    }

    public int getPointsEquipeExterieur() {
        return pointsEquipeExterieur;
    }

    public void setPointsEquipeExterieur(int pointsEquipeExterieur) {
        this.pointsEquipeExterieur = pointsEquipeExterieur;
    }

    public int getPointsEquipeDomicile() {
        return pointsEquipeDomicile;
    }

    public void setPointsEquipeDomicile(int pointsEquipeDomicile) {
        this.pointsEquipeDomicile = pointsEquipeDomicile;
    }
}
```

Il ne faut pas longtemps pour se rendre compte que cette première ébauche permet de créer des objets Java ne correspondant pas à des scores de rugby.

```
package rugby;
```

```
public class Main {
    public static void main(String[] args) {
        RugbyScore rs;
        rs = new RugbyScore(0,0);
        System.out.println(rs);
        rs.setPointsEquipeDomicile(-58);
        System.out.println(rs);
        rs = new RugbyScore(12895,-859);
        System.out.println(rs);
    }
}
```

Exécution :

```
0-0
-58-0
12895--859
```

On voit d'abord le cas d'un objet représentant un score correct (0-0), qui à la suite d'une invocation de modificateur (setPointsEquipeDomicile) ne correspond plus à rien de réel et ensuite le cas d'un objet incorrect dès sa création.

En écrivant la classe autrement, on peut empêcher la création d'objets incorrects ainsi que les évolutions incorrectes d'objets existants.

2.3 Version assurant la cohérence des scores

Pour tous les événements susceptible de faire évoluer le score, nous avons choisi de donner en paramètre l'équipe qui marque des points sous la forme d'un booléen qui dit si oui ou non il s'agit de l'équipe qui joue à domicile. Cela évite de faire deux méthodes pour chacun de ces événements.

```
package rugby;

public class RugbyScore2 {
    private int pointsEquipeDomicile, pointsEquipeExterieur;

    public RugbyScore2() {
        this.pointsEquipeDomicile = 0;
        this.pointsEquipeExterieur = 0;
    }

    public int getPointsEquipe(boolean aDomicile) {
        if (aDomicile)
            return pointsEquipeDomicile;
        else
            return pointsEquipeExterieur;
    }

    public void enregistrePenaliteOuDrop(boolean aDomicile){
        if (aDomicile)
            pointsEquipeDomicile = pointsEquipeDomicile+3;
        else
```

```
            pointsEquipeExterieur = pointsEquipeExterieur+3;
    }

    public void enregistreEssai(boolean aDomicile){
        if (aDomicile)
            pointsEquipeDomicile = pointsEquipeDomicile+5;
        else
            pointsEquipeExterieur = pointsEquipeExterieur+5;
    }

    public void enregistreEssaiTransforme(boolean aDomicile){
        if (aDomicile)
            pointsEquipeDomicile = pointsEquipeDomicile+7;
        else
            pointsEquipeExterieur = pointsEquipeExterieur+7;
    }


    public String toString(){
        return pointsEquipeDomicile + "-" + pointsEquipeExterieur;
    }
}
```

Avec cette classe, lorsqu'on crée un objet, le score est nul et vierge. Ensuite, aucune méthode ne permettra de mettre l'objet dans un état tel qu'il ne corresponde pas à un score possible au rugby.

Parmi les propriétés des scores de rugby respectées par la classe, il y a le fait que les nombres de points de chaque équipe ne peuvent qu'augmenter, jamais diminuer, et le fait que les augmentations se font par palliers de 3, 5 ou 7 points.

2.4 Amélioration de la classe

Il existe une propriété importante des scores de rugby qui n'est pas reflétée par la classe que nous avons écrite, c'est le fait qu'un score peut être définitif et ne plus jamais évoluer. Une fois la fin du match sifflée, plus aucun point ne peut être marqué.

Nous allons améliorer notre classe avec l'ajout d'un attribut pour enregistrer le passage du score à l'état définitif et une méthode pour basculer cet attribut d'un état à l'autre.

```
public class RugbyScore3 {
    private int pointsEquipeDomicile, pointsEquipeExterieur;
    boolean termine;

    public RugbyScore3() {
        this.pointsEquipeDomicile = 0;
        this.pointsEquipeExterieur = 0;
        termine = false;
    }
    public void terminer(){
        if (!termine)
            termine = true;
    }
    ...
```

Ensuite, il faut adapter les méthodes modificatrices pour que le score n'évolue plus après avoir atteint son état définitif. Quand une méthode qui change le score est appelée alors que le score est définitif, cette méthode va lever une exception. Nous allons utiliser une exception prédéfinie de Java qui est prévue pour des cas de ce genre : l'exception `IllegalStateException`.

```
public void enregistrePenaliteOuDrop(boolean aDomicile){
    if (termine)
        throw new IllegalStateException("Le␣score␣ne␣peut␣plus␣évoluer");
    if (aDomicile)
        pointsEquipeDomicile = pointsEquipeDomicile+3;
    else
        pointsEquipeExterieur = pointsEquipeExterieur+3;
}

public void enregistreEssai(boolean aDomicile){
    if (termine)
        throw new IllegalStateException("Le␣score␣ne␣peut␣plus␣évoluer");
    if (aDomicile)
        pointsEquipeDomicile = pointsEquipeDomicile+5;
    else
        pointsEquipeExterieur = pointsEquipeExterieur+5;
}

public void enregistreEssaiTransforme(boolean aDomicile){
    if (termine)
        throw new IllegalStateException("Le␣score␣ne␣peut␣plus␣évoluer");
    if (aDomicile)
        pointsEquipeDomicile = pointsEquipeDomicile+7;
    else
        pointsEquipeExterieur = pointsEquipeExterieur+7;
}
```

2.5 Leçons à retenir de cet exemple

Cet exemple nous montre plusieurs choses.

— Il faut s'assurer que l'on ne crée que des nouveaux objets corrects. Cela nécessite de bien vérouiller les constructeurs dans nos classes.

— Il faut s'assurer qu'un objet correct ne devienne pas incorrect à la suite d'une modification. Cela impose de faire attention aux méthodes modificatrices.

— Les méthodes `setXXX` qui permettent de donner n'importe quelle valeur à un attribut ne doivent pas devenir un réflexe conditionné. Il est rare que l'on puisse fixer librement la valeur d'un attribut. C'est l'exception plus que la règle.

3 Exceptions liées à la cohérence

Il y a deux exceptions qu'on utilise fréquemment pour signaler un problème d'incohérence dans un objet. On peut utiliser ces exceptions dans le constructeur pour éviter de construire un objet incohérent et dans les modificateurs pour éviter de rendre incohérent un objet existant.

— `IllegalArgumentException` : pour signaler qu'un paramètre quoique du bon type, a une valeur qui ne convient pas pour la méthode ou le constructeur concerné. Par exemple, pour

la méthode `deposer` d'un compte bancaire, un nombre négatif n'est pas une valeur possible pour le montant à déposer sur le compte.

— `IllegalStateException` : pour signaler qu'une méthode ne peut pas s'appliquer dans l'état actuel d'un objet. Nous avons vu un exemple, celui des matchs terminés pour les scores desquels les méthodes d'ajout de points ne doivent plus être appelées. Cette exception n'a pas de sens dans un constructeur : il faut que l'objet existe et soit dans un certain état pour s'appliquer.

— Dans les deux cas traités par ces exceptions, on peut vouloir être plus précis en créant une exception spécifique à une erreur pour notre application. Dans ce cas, on peut dériver cette nouvelle exception de celle des deux qui va bien. Par exemple, on peut créer l'exception `ModificationScoreMatchTermineException` à partir de `IllegalStateException` avec le code suivant :

```
\item public class ModificationScoreMatchTermineException extends
  IllegalStateException{}
```