Les bases de la POO

ICAM – JP Gouigoux – 09/2012

Un peu d’histoire

- Programmation séquentielle : l’ordinateur déroule une liste d’instructions et les exécute pas à pas
- Programmation procédurale : des procédures, contenant du code, peuvent elles-mêmes être appelées par du code, suivant des enchaînements contrôlés => on abstrait des méthodes
- Programmation orientée objet : des objets définissent des fonctions et des contenus, et peuvent s’appeler les uns les autres => on abstrait des méthodes, mais aussi des contenus
- Programmation évènementielle : au lieu de suivre un déroulement d’appels (entre procédures ou entre objet), on laisse en partie la main à un orchestrateur qui déclenche un traitement à partir d’un évènement produit depuis une entité
- Programmation fonctionnelle : définir des comportements par des règles, et laisser un orchestrateur composer les fonctions

L’idée de base de la POO

- Une classe définit des champs et des méthodes
  - Les champs contiennent l’état
  - Les méthodes définissent les interactions
- Une classe peut être instanciée
  - L’instance est une concrétisation en mémoire d’un modèle défini par la classe
- La modélisation par classe permet une abstraction du domaine à modéliser
  - Attention à la fausse POO, où les instances ne font que simuler une programmation procédurale
  - On revient à des approches plus pures de la POO par le Domain Driven Design, où la conception est basée sur une modélisation propre du métier, avec une classe pour chaque notion, et un schéma objet parlant aux intervenants fonctionnels

Un exemple

- Classe « Voiture »
  - « Vitesse » est un membre de type numérique
  - « Accelerer(float) » est une méthode affectant le membre « Vitesse »
- Une variable « a » pointe sur une instance de la classe « Voiture »
  - On peut lire la vitesse en appelant a.Vitesse
  - L'état interne de l'instance peut être modifié en appelant a.Accelerer(25.0)
- En Java, on écrira :
  - Voiture a = new Voiture();
  - a.Accelerer(25.0);
  - float b = a.Vitesse;

Les concepts de la POO

- Expressivité : concepts métier portés par le code
- Encapsulation : l'état d'une instance est principalement manipulable par cette instance, évitant ainsi les risques d'erreur
- Polymorphisme : une instance peut se présenter sous la forme de plusieurs classes
  - Exemple : l'instance a est une Ferrari, mais également une voiture, par héritage
  - Exemple : l'instance a est une voiture, mais remplit également le contrat IPesable, car elle propose une propriété de lecture de sa masse

Les relations en POO

- Composition : un membre d'une classe est du type représenté par une autre classe
  - C'est une relation « has a »
- Identité : une instance est identifiable comme appartenant au type défini par une classe
  - C'est une relation « is a » (rare)
- Contractuelle : une instance suit un contrat de fonctionnement défini par une interface
  - C'est une relation « works as a » (plus courant)

Concept d'encapsulation

- Idéalement, l'état d'un objet est modifiable par lui-même et aucun autre
- On peut pousser jusqu'à encapsuler les lectures de l'état depuis l'extérieur

```
public class Voiture
{
    private float _Vitesse = 0;

    public void Accelerer(float delta)
    {
        _Vitesse += delta;
    }

    public float getVitesse()
    {
        return _Vitesse;
    }
}
```

Intérêts de l'encapsulation

- Centraliser les contrôles sur l'état de l'objet
- Imposer des règles et limitation de visualisation ou de manipulation de l'objet

```
public class Voiture
{
    private float _Vitesse = 0;

    public void Accelerer(float delta) throws Exception
    {
        // Il y a une limite à l'accélération
        if (delta > 10) throw new Exception("Accélération trop forte");
        _Vitesse += delta;
        // Il y a également une limite à la vitesse
        if (_Vitesse > 130) _Vitesse = 130;
    }

    // Inutile d'embêter les utilisateurs de la classe avec les virgules
    public int getVitesse()
    {
        return (int)_Vitesse;
    }
}
```

Concept d’héritage

- On peut créer des lignées de classes héritant chacune d’une classe mère (l’héritage multiple est rarement supporté)
- Les membres et méthodes de la classe mère sont accessibles sur la classe fille (si encapsulation public ou protected)

```
public class Ferrari extends Voiture
{
    public void AccelererFort(float delta) throws Exception
    {
        if (delta > 30) throw new Exception("Accélération trop forte");
        _Vitesse += delta; // Possible, car _Vitesse déclaré en protected dans Voiture
        if (_Vitesse > 250) _Vitesse = 250;
    }
}
```

Polymorphisme

- Une méthode peut remplacer la méthode de sa classe mère

```
public class Ferrari extends Voiture
{
    public void Accelerer(float delta) throws Exception
    {
        if (delta > 30) throw new Exception("Accélération trop forte");
        _Vitesse += delta; // Possible, car _Vitesse déclaré en protected dans Voiture
        if (_Vitesse > 250) _Vitesse = 250;
    }
}
```

```
public static void main(String[] args) throws Exception {
    Voiture a = new Voiture();
    a.Accelerer((float) 9.8);
    System.out.println(a.getVitesse());

    Voiture b = new Ferrari(); // Manifestation de polymorphisme
    b.Accelerer((float) 28.2); // Surcharge de la méthode
    System.out.println(b.getVitesse());
}
```

Limitation de l'héritage

- La relation « is a » est en fait assez rare, et souvent confondue avec « works as a » (effet « Canada Dry »)
- Exemple : la classe Carre hérite-t-elle de Rectangle ?
  - Un carré est bien (is a) un type particulier de rectangle
  - Mais du coup, un carré aura une largeur et une longueur
  - Que faire si du code modifie l'un et pas l'autre ?
  - La notion de prototype peut être plus proche : un rectangle spécialise un carré en lui ajoutant un membre supplémentaire de longueur, en plus de la largeur

Gestion par interface

- La solution à cette limitation est d'utiliser la contrainte d'un « works as a » plutôt qu'un « is a »
- En termes techniques, il s'agit d'une interface
- L'interface agit comme un contrat

```
public interface IPilotable
{
    public void Accelerer(float delta) throws Exception;
    public int getVitesse();
}

public class Voiture implements IPilotable
{
    protected float _Vitesse = 0;

    public void Accelerer(float delta) throws Exception
    {
```

Programmation contractuelle

- Les interfaces sont la base des contrats de fonctionnement entre composants
- Comme en droits, les contrats permettent d'éviter les conflits et incompréhension, résultant en un fonctionnement erratique
- Une expertise de la POO se base sur ces concepts, et a fourni la programmation SOLID

SOLID : OOP made right

- Single Responsibility : une classe remplit une fonction et une seule
- Open Closed : une classe est ouverte à l'extension, mais fermée aux modifications
- Liskov Substitution Principle : lorsqu'une classe se substitue à une autre, son contrat de fonctionnement est respecté
- Interface Segregation : on crée des interfaces séparées pour les différents contrats
- Dependency Inversion : le contrat n'étant pas porté par le module contenant la classe qui l'implémente, mais par le module ayant besoin de cette classe, la dépendance est inversée par rapport au sens commun

Inversion de dépendance

- Une voiture a besoin d'un moteur
  - Voiture dépend de Moteur
- A mieux y réfléchir, une voiture a besoin d'une classe quelconque qui lui fournisse de l'énergie
- Bref, elle a besoin d'une instance d'une classe implémentant IFournisseurEnergie
- Si le module contenant Voiture expose IFournisseurEnergie, il peut se garantir que les modules l'implémentant le feront de manière compatible à ses propres besoins, quel que soit leur nombre
- La classe Moteur doit maintenant implémenter IFournisseurEnergie
- Moteur a donc besoin du module contenant Voiture
  - Inversion de dépendance par rapport au sens initialement envisagé