Synthèse des amphis

Brocart Guillaume

30 septembre 2017

Table des matières

Introduction

1 Paradigmes

Un paradigme de programmation correspond à la méthode employée pour résoudre un problème dans un langage de programmation. On distingue deux principaux paradigmes:

- La programmation impérative
- La programmation déclarative

La programmation impérative est la plus utilisée, du fait qu'elle regroupe la programmation procédurale (C, BASIC) ainsi que la programmation orientée objet (Java, Kotlin, C++, ...). Contrairement aux langages impératifs, où il faut décrire comment arriver au résultat voulu, les langages déclaratifs décrivent le résultat, mais non comment y arriver. Des langages comme le HTML et le LaTeX sont déclaratifs du fait que l'on ne spécifie pas comment afficher le texte en gras: on utilise seulement une balise pour arriver au résultat.

1.1 Caractérisation d'un langage

On caractérise un langage par trois aspects:

- Le modèle de calcul

Comment le langage résous un problème ?

- La modularité

Quels sont les outils disponibles ?

- Les types

Comment les types sont-ils gérés ?

Le paradigme de programmation le plus utilisé actuellement est la programmation orientée objet, du fait de sa simplicité d'utilisation.

2 Rappels de POO

2.1 Objet vs classe vs type

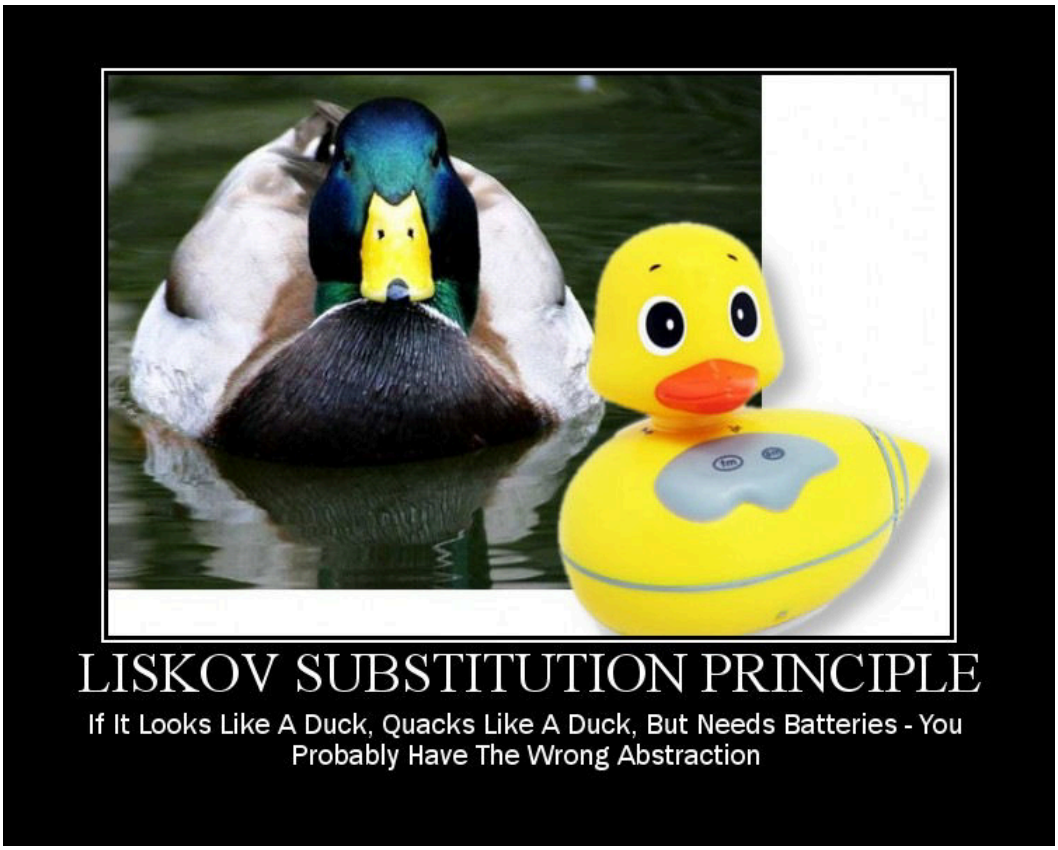
Un **type** est une interface représentant les fonctionnalités d'un objet, comment il peut interagir avec son environnement. Une **classe** définit comment les objets peuvent assurer leurs fonctionnalités. Un **objet** est donc une structure en mémoire constituée de méthodes et de données.

2.2 Héritage

En POO, on peut utiliser l'héritage pour permettre à une classe fille d'avoir accès aux méthodes et aux variables présentes dans une classe mère. On utilise le mot clé **extends** pour indiquer qu'une classe hérite d'une autre, et **super** pour faire référence à la classe mère.
Note: Impossible d'avoir plusieurs classes mères Java.

2.3 Substitution

Si une classe S est un sous-type d'une classe C, alors on peut remplacer C par S sans causer de problèmes. Cette propriété est connue sous le nom de "Principe de substitution de Liskov". C'est un des principes fondamentaux de la POO.
Un carré est un rectangle. Un triangle équilatéral est un triangle. Donc dire qu'un carré est un sous-type de Rectangle semble correcte, mais cette implémentation est erronée, parler de hauteur et de largeur pour un carré semble peu naturel. De plus, si la classe Rectangle contient des setters, on pourrait se retrouver avec un carré ayant des cotés non égaux... Ce principe s'applique aussi avec les canards :)



2.4 Dynamic dispatch

```
class A{
    void m1(){
        System.out.println("Hello World depuis A");
    }
}
class B extends A{
    @Override
    void m1(){
        System.out.println("Hello World depuis B");
    }
}
class C extends A{
    @Override
    void m1(){
        System.out.println("Hello World depuis C");
    }
}
```

```
class DispatchTest{
    public static void main(String args[]){
        A a = new A();
        B b = new B();
        C c = new C();

        // ref est de Type A
        A ref = a;
        // Hello World depuis A
        ref.m1();
        // B est un sous-type de A
        ref = b;
        // Hello World depuis B
        ref.m1();
        // C est un sous-type de A
        ref = c;
        // Hello World depuis C
        ref.m1();
    }
}
```

On voit que c'est la méthode du sous-type le plus spécifique qui est appelée, même si ref est déclaré comme étant de type A. En effet, B et C sont des sous-types, plus spécifiques que A: il est correct d'écrire ref = b. Toutefois, on aurait une erreur de compilation si on voulait mettre dans ref un sur-type de A.

Factorisation et spécifications

1 Factorisation

La factorisation permet de répondre à la question suivante: quel code dans quelle classe ? Cela permet de faciliter la lisibilité du code ainsi que son entretien. On peut décrire une factorisation par trois critères: la responsabilité, la cohésion et le couplage

1.1 Responsabilités

Lorsque l'on écrit une classe, on définit ses responsabilités. Une classe doit connaître ses attributs, qui constituent son identité; ses méthodes, qui constituent les actions qu'elle peut effectuer et sa visibilité par rapport aux autres classes.

1.2 Cohésion

La cohésion est le degré d'interaction et d'interdépendance des données et des méthodes à l'intérieur d'une classe. La cohésion d'une classe doit être **forte**.

1.3 Couplage

Le couplage est le degré d'interaction et d'interdépendance des de données et méthodes entres classes. Le couplage d'une classe doit être **faible**.

2 Spécifications

Une spécification est un contrat pour l'appel d'une méthode, permettant de découpler et de séparer les responsabilités entre différentes méthodes. Les spécifications sont constituées de deux types de conditions:

- Les pré-conditions

C'est à dire les contraintes pour l'appel de la méthode.

- Les post-conditions

C'est à dire les garanties que la méthode peut remplir une fois son exécution terminée. La post-condition n'est donc pas déterministe: on peut avoir plusieurs implémentations qui satisfont une même post-condition.

2.1 Critères

Les spécifications doivent être **brèves**, suffisamment **fortes**, pour être précises mais aussi suffisamment **faibles** pour rester faisables.

Les sous-types

1 Les types

Tout langage de programmation a des types. Ces derniers sont le plus souvent connus dès la compilation. On parle de typage statique, comme en Java, en C, ... Des langages plus modernes comme le Python ou le Javascript ont un typage dynamique. C'est à dire que c'est le compilateur qui va décider du type approprié pour une variable.

1.1 Utilisations

Pourquoi avoir créé des types ?
Les types sont nécessaires pour savoir combien de mémoire va être allouée pour une certaine variable. Un char prendra 1 octets, un int 4 octets, un double 8, ... Les types permettent aussi le type safety. Cela signifie que l'on ne peut pas faire une opération incompatible avec un type. Par exemple, on ne pourra pas multiplier deux String ensemble, alors qu'on peut très bien multiplier deux int.

2 Les sous-types

2.1 Sous-types structurels

Sous-types ayant le minimum nécessaire pour permettre la substitution.

- Les types de constantes doivent être **covariants**.
- Les types de variables doivent être **invariants**.
- Les types de retour doivent être **covariants**.
- Les types de paramètres doivent être **contravariants**.

2.2 Sous-types comportementaux

Les sous-types comportementaux vérifient les conditions des sous-types structurels, mais poussent le concept encore plus loin, car **toute propriété vraie dans un type l'est aussi dans son sous-type** (principe de substitution de Liskov).

2.3 Généricité et sous-types

La généricité et les sous-types sont généralement incompatibles. En effet, les paramètres ne sont pas contravariants, et le type de retour n'est pas covariant.

```
List<E> list = new List<E>();
boolean list.add(E e);
E list.pop();
```

Note: Si S sous-type de T, alors S[ ] sous-type de T[ ].

Sources

- Thomas Nowak
- https://stackoverflow.com/questions/56860
- http://enwp.org/Covariance_and_contravariance_(computer_science)
- https://www.javatpoint.com/runtime-polymorphism-in-java