

Conception et programmation orientées objet avancées

Introduction

- Rappels sur la programmation objet
- Conception objet avancée

Thierry Charnois d'après les cours de Dominique Bouthinon (dépt info)
et certains exemples empruntés à :

<http://liris.cnrs.fr/laetitia.matignon/index/ISI32013/>

Rappel sur la programmation objet

Les concepts objets de base

Classes/objets

Les classes sont des modèles pour créer des objets qui communiquent entre eux par messages (appels de méthodes)

Encapsulation

On cache (*private*) la structure d'un objet et on ne révèle (*public*) que les fonctions (méthodes) nécessaires

Héritage

Une classe peut hériter des données et des méthodes d'autres classes

Polymorphisme (d'héritage)

Une méthode de même signature peut avoir des comportements (instructions) différents selon la classe où elle est (re)définie.

Intérêts des concepts objets de base

Sécurité

L'*encapsulation* permet de protéger un objet contre des modifications inappropriées

Souplesse

Le *polymorphisme* permet d'utiliser un même code sur des objets de différentes classes

Factorisation

L'*héritage* permet de factoriser des données et instructions

Réutilisation

Les notions de *classe* et d'*héritage* permettent de réutiliser de données et du code dans différents contextes.

Limites des concepts objets de base

Ils ne permettent pas de garantir la conception de programmes :

- maintenables
- extensibles
- fiables

Conception objet avancée

Objectifs

Concevoir des logiciels :

- (facilement) maintenables
- (facilement) extensibles
- fiables
- réutilisables

Mise en oeuvre

- **Programmation orientée objet SOLID**

[R. C. Martin, 2002]

- **Design patterns (patrons de conception)**

- [E. Gamma, R. Helm, R. Johnson et J. Vlissides, 1994]

Ouverture/fermeture (*Open/*C*losed*)

Ouverture/fermeture (*Open/*C*losed*)

Une classe doit être ouverte aux extensions (ajouts) mais fermée aux modifications (du code existant)

•Ouverture aux extensions :

- le comportement d'un module doit pouvoir être étendu
- exemple : ajout de nouvelles méthodes

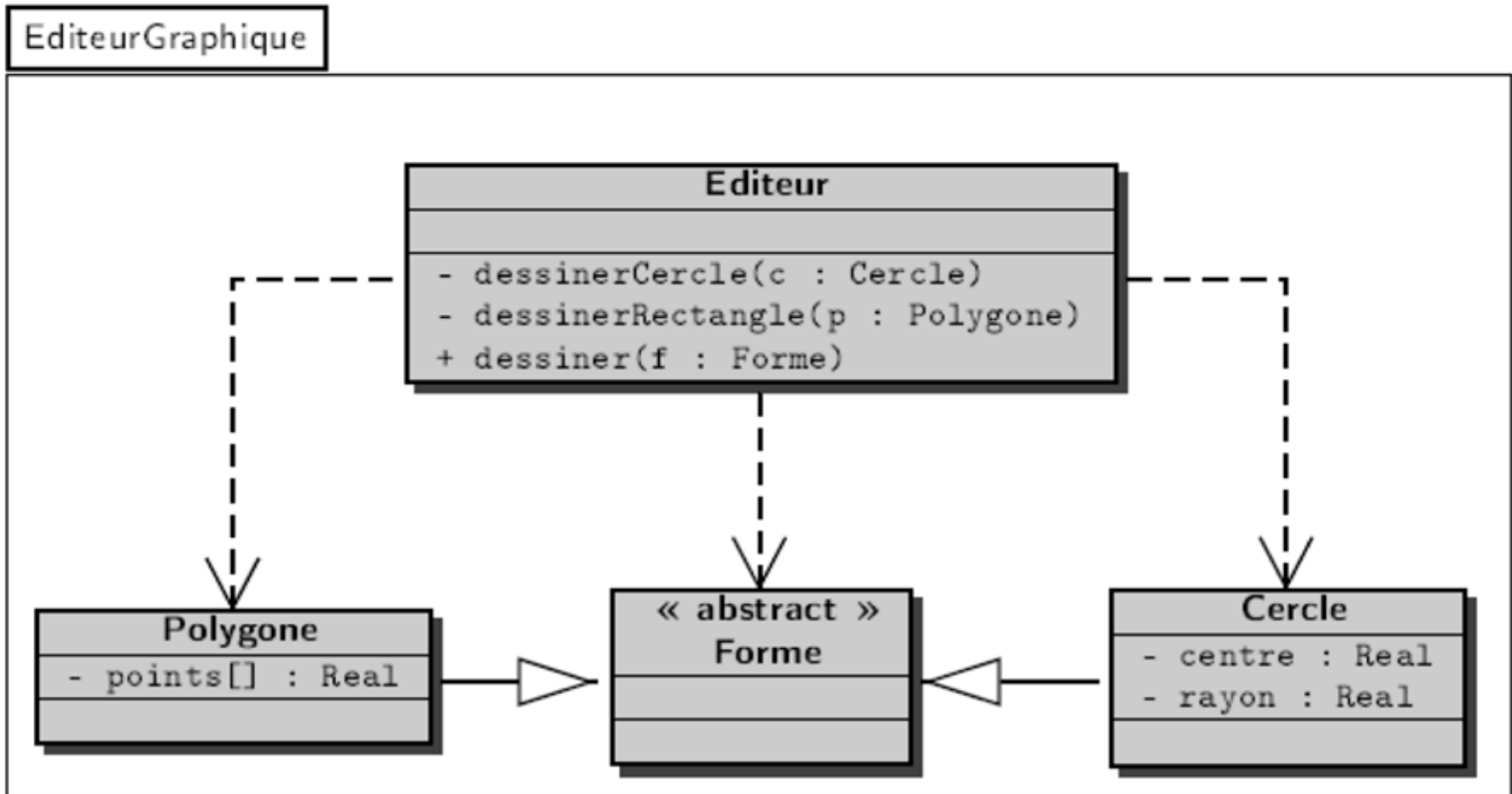
•Fermeture aux modifications :

- le comportement d'un module doit pouvoir être étendu mais sans modification de son code source.
- exemple : mise en place d'une interface

•En d'autres termes, l'ajout de fonctionnalités doit se faire en ajoutant du code et non en éditant du code existant.

- Exemple : Une méthode privée d'une classe est fermée à la modification (aucun autre code ne peut la détraquer) ; mais on peut ajouter des méthodes publiques qui invoquent cette méthode privée pour étendre le comportement des méthodes privées sans les modifier.

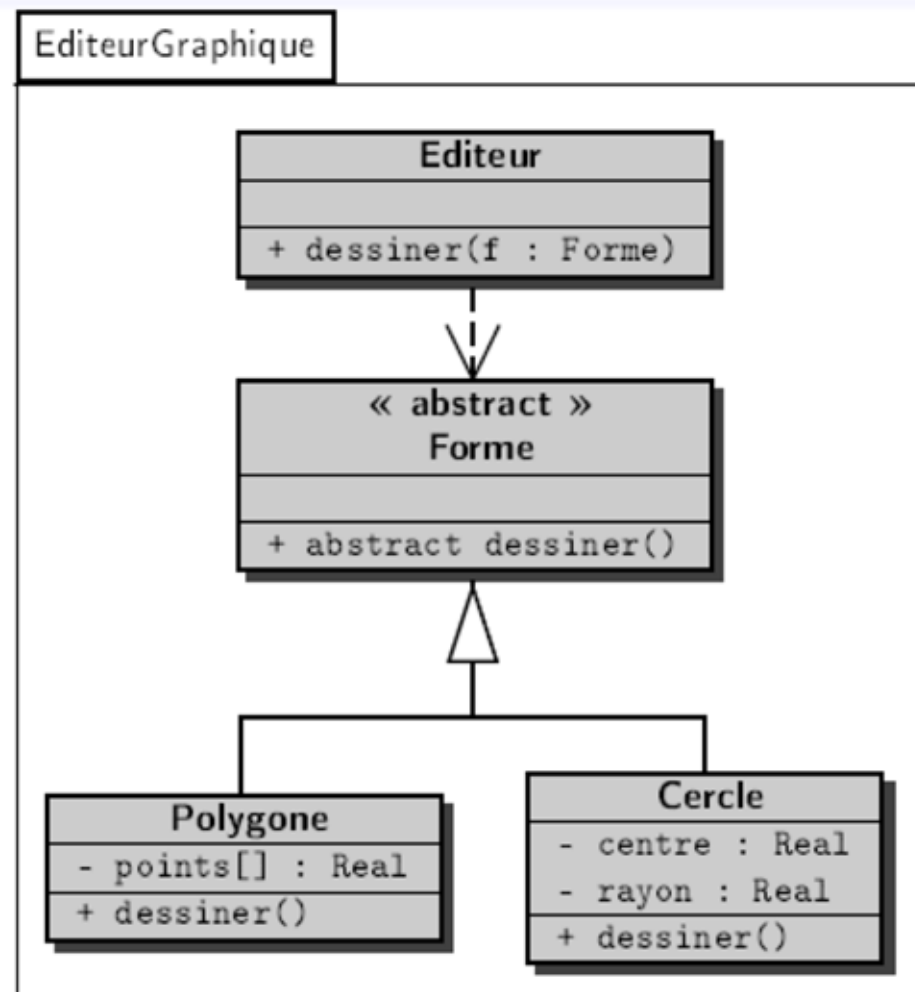
Ouverture/fermeture (*Open/Closed*)



Principe "open-closed" : on doit pouvoir facilement ajouter des formes à l'application sans modifier les classes existantes → principe respecté ?

NON : éditeur non fermé aux modifications si on ajoute une forme

Ouverture/fermeture (*Open/Closed*)



On peut étendre le comportement sans modifier le code : Editeur est fermée aux modifications car la méthode `dessiner()` ne change pas si l'on ajoute une nouvelle `Forme`. Editeur est ouverte aux extensions : toutes les sous-classes de `Forme` peuvent changer le comportement de `dessiner()`

Substitution de Liskov

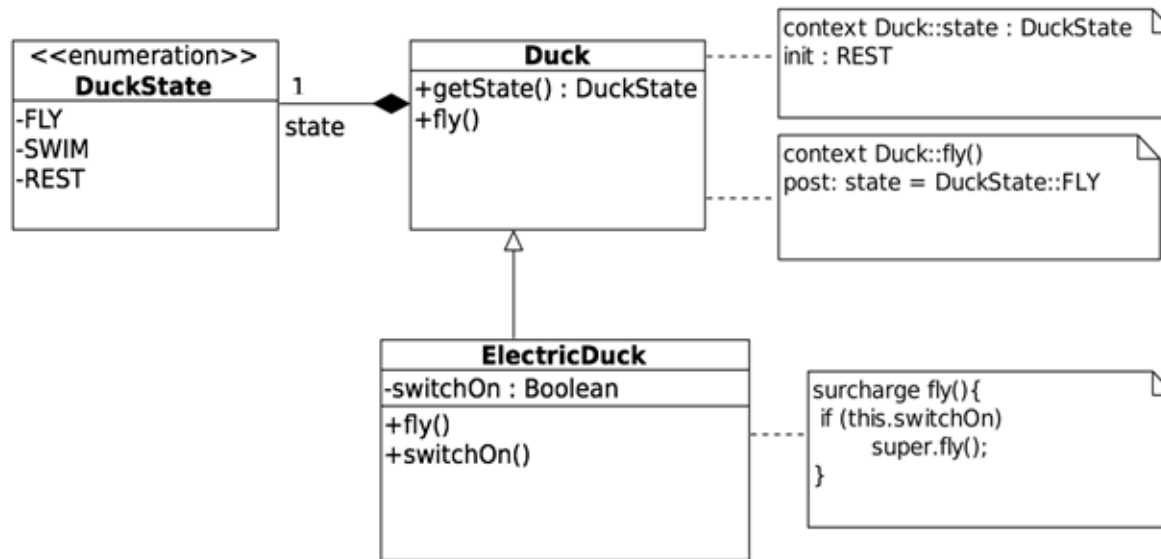
Un instance d'une classe doit pouvoir être substituée sans modification par une instance d'une sous-classe (et sans que le programme ne soit altéré dans son comportement)

Autrement dit :

Si une classe B hérite d'une classe A, tout programme écrit pour traiter des instances de A doit pouvoir traiter des instances de B sans même savoir que ce ne sont pas des instances directes de A.

Conséquences

Ne pas introduire de modifications dans le fonctionnement de B qui rend inutilisable tout objet de type B utilisé comme un objet de type A.

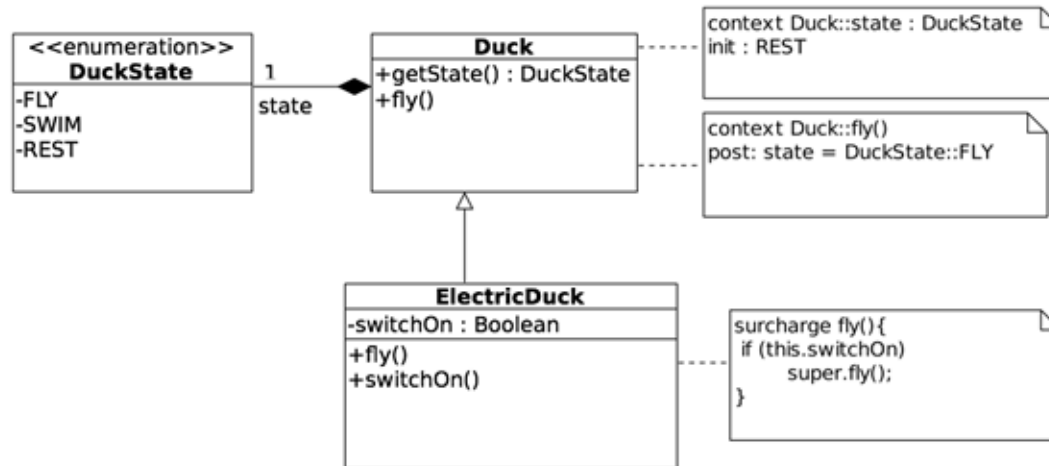


```

public class Test {
    public void test(List<Duck> myList ) {
        int nbFlyingDucks = 0;
        for (Duck duck : myList) {
            duck.fly(); //on fait voler les canards
            if (duck.getState() == DuckState.FLY) nbFlyingDucks++; }
        if (nbFlyingDucks != myList.size()) //on verifie qu'ils sont tous en train de voler
            throw new Exception("Some_ducks_not_flying");}
    public static void main(String[] args){
        List<Duck> myList = new ArrayList<Duck>();
        myList.add(new Duck()); myList.add(new Duck());
        new Test().test(myList); //OK
        myList.add(new ElectricDuck());
        new Test().test(myList); }}
    //Exception levee: le canard electrique n a pas pu decoller car non allume
  
```

Principe de Liskov non respecté : la méthode Test qui utilise des instances de la classe Duck doit pouvoir utiliser des instances de la classe dérivée ElectricDuck sans que le programme ne soit altéré dans son comportement.

Que penser de cette solution ?



Solution :

```
public class Test{
    public void test(List<Duck> myList ) {
        int nbFlyingDucks = 0;
        for (Duck duck : myList) {
            if (duck instanceof ElectricDuck) { ((ElectricDuck) duck).
                switchOn();}
            duck.fly();
            if (duck.getState() == DuckState.FLY) nbFlyingDucks++; }
        if (nbFlyingDucks != myList.size()) {
            throw new Exception("Some_ducks_not_flying"); } } }
```

Liskov respecté mais la méthode `Test` a besoin de connaître le type réel des objets pour pouvoir les traiter correctement : module non-fermée aux modifications
→ principe open closed non respecté.... solution bientôt

Design patterns

Les *design patterns* (patrons de conception) décrivent des procédés généraux et réutilisables pour concevoir des logiciels

Les *design patterns* mettent en œuvre les principes SOLID

Un *design pattern* décrit une bonne pratique et une solution standard en réponse à un problème de conception d'un logiciel

Classification des design patterns

Création (créer des objets)

Fabrique, Fabrique abstraite, Prototype, Singleton, Monteur

Structure (organiser les classes)

Décorateur, Adapteur, Façade, Pont, Objet composite, Poids-mouche, Proxy

Comportement (organiser la collaboration entre objets)

Observateur, Stratégie, Commande, Médiateur, Chaîne de responsabilité, Itérateur, Interpréteur, Etat, Patron de méthode, Visiteur, Fonction de rappel