

1

Le meilleur outil de conception pour le développement de logiciels est un esprit bien éduqué sur les principes de conception. Ce n'est pas UML ou toute autre technologie

Craig Larman

3

Avertissement

INF-5153 N'EST PAS UN COURS D'UML
Même si on utilise intensivement UML
UML n'est qu'un Langage
(c'est même écrit dans le titre)

2

Crédits

UNIVERSITÉ CÔTE D'AZUR



Mireille Blay-Fornarino

Université Côte d'Azur
Laboratoire I3S, SPARKS

Imbattable au sprint, même avec des talons

<http://mireilleblayfornarino.i3s.unice.fr/>

4

1 Principes de GRASP

Les principaux patrons GRASP

3 SOLIDité du Code & de la conception

5

A pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

- Christopher Alexander, Professeur d'Architecture

Principes de GRASP

1

6

Patron de conception

🔗 Pour les articles homonymes, voir [Patron](#).

En [informatique](#), et plus particulièrement en [développement logiciel](#), un **patron de conception** (souvent appelé *design pattern*) est un arrangement caractéristique de modules, reconnu comme bonne pratique en réponse à un problème de conception d'un logiciel. Il décrit une solution standard, utilisable dans la conception de différents logiciels¹.

**Si on a des solutions,
Quels sont les problèmes ?**

7

8

G R A S P eneral esponsibility ssignment oftware attern

9

Conception dirigée par les Responsabilités

- **Anthropomorphisme :**
 - Attribution de caractéristiques humaines à des systèmes non-humains
- **Métaphore de la Conception Orientée Objet :**
 - Communauté d'objets responsables collaborant à la réalisation d'un objectif
- **Objectif :**
 - Penser le logiciel en terme de responsabilités par rapport à des rôles, à travers des collaborations

11

Patrons Logiciels Généraux d'Affectation des Responsabilités

10

Qu'est-ce qu'une responsabilité ?

- Ce n'est pas une classe
- Ce n'est pas une méthode
- Ce n'est pas un cas d'utilisation
- ...

12-1

Qu'est-ce qu'une responsabilité ?

- Ce n'est pas une classe
 - Ce n'est pas une méthode
 - Ce n'est pas un cas d'utilisation
 - ...
- C'est une ABSTRACTION d'un comportement
- Les méthodes s'acquittent des responsabilités

12-2



14

Objectifs de GRASP

- Réduire le décalage entre
 - La représentation "humaine" du problème
 - Sa représentation dans le système logiciel
- Exemple : Comment représenter un échiquier ?

13

Un objet doit "Savoir les choses"

- Connaitre les valeurs de ses propriétés
 - Par exemple : variables privées interne
- Connaitre les objets qui lui sont rattachés
 - Par exemple : références
- Connaitre les données qu'il peut dériver
 - Par exemple : taille d'une liste

15

Un objet doit “Faire les choses”

- Faire quelque chose
 - Par exemple : un calcul, créer un autre objet
- Déclencher une action sur un autre objet
 - Par exemple : “je ne sais pas faire, mais lui sait faire”
- Coordonner les actions des autres objets
 - Par exemple : Je demande à x de faire Y, et en fonction, je fais Z.

16

Modèles

- 1.1 Contrôleur
- 1.2 Createur
- 1.3 Une forte cohésion
- 1.4 Indirection
- 1.5 Spécialiste de l'Information
- 1.6 Le couplage faible
- 1.7 Le polymorphisme
- 1.8 Protégé des variations
- 1.9 Pure invention

9
patrons

18

Patrons

GRASP



17

Spécialiste de l'information

- Problème :
 - Quel est le principe général d'affectation des responsabilité des objets ?
- Solution :
 - Affecter la responsabilité à la classe spécialiste, c-a-d la classe qui possède les informations nécessaire pour s'en acquitter.

19

Au Monopoly ...

- Qui est responsable de l'accès à une case donnée du jeu ?

⊕ Sylvain Cherrier, Design Patterns

20-1

Expert (GRASP) : Discussion

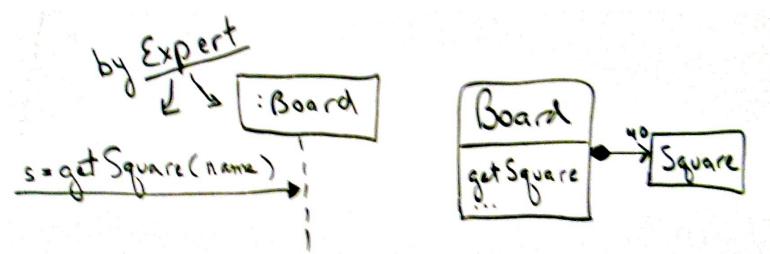
- Le plus utilisé de tous les patterns d'attribution de responsabilité
- Un principe de base en OO
- L'accomplissement d'une responsabilité nécessite souvent que l'information nécessaire soit répartie entre différents objets.

⊕ Sylvain Cherrier, Design Patterns

21

Au Monopoly ...

- Qui est responsable de l'accès à une case donnée du jeu ?



⊕ Sylvain Cherrier, Design Patterns

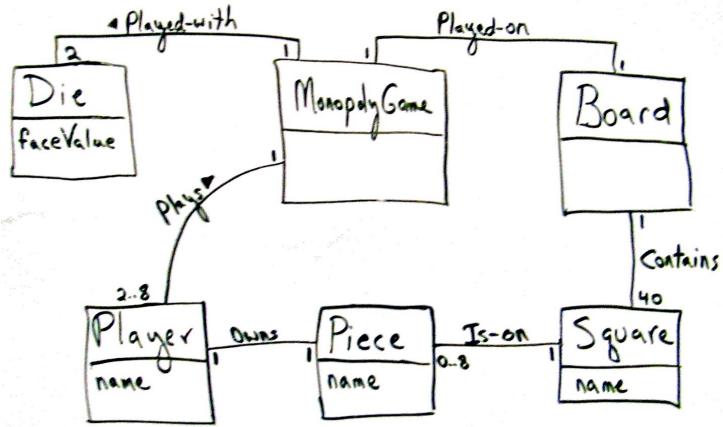
20-2

Créateur

- Problème :
 - Qui doit avoir la responsabilité de créer une nouvelle instance de la classe ?
- Solution :
 - Affecter à une classe B la responsabilité de créer une instance de A si
 - B contient ou agrège des objets A,
 - B utilise étroitement des objets A
 - B connaît les données utilisées pour initialiser les objets A

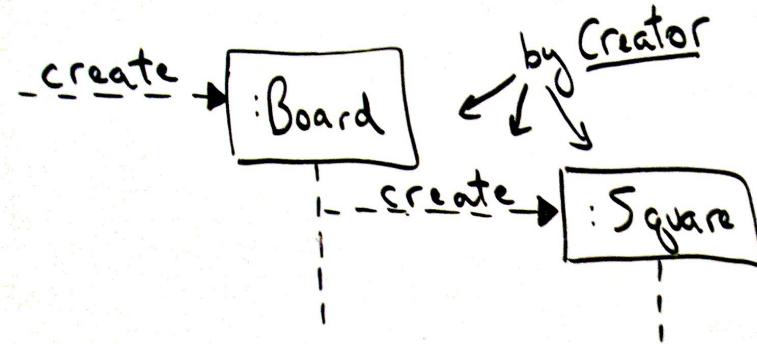
22

Qui créé les cases au Monopoly ?



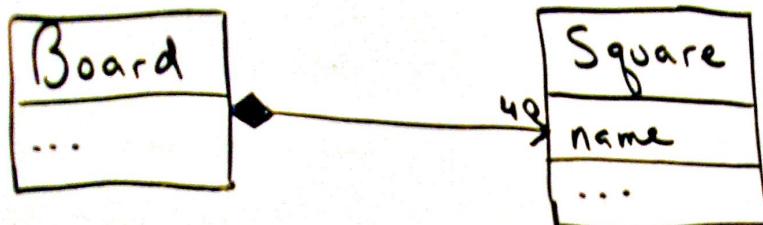
23

Diagramme de séquence à l'aide



24

L'association est donc une composition



25

Faible Couplage

- Problème :
 - Comment minimiser les dépendances, réduire l'impact des changements et augmenter la réutilisation ?
- Solution :
 - Mesurer le couplage “en continu”
 - Identifier les différentes solutions à l'affectation de responsabilité
 - Choisir la solution préférée en fonction de ce critère

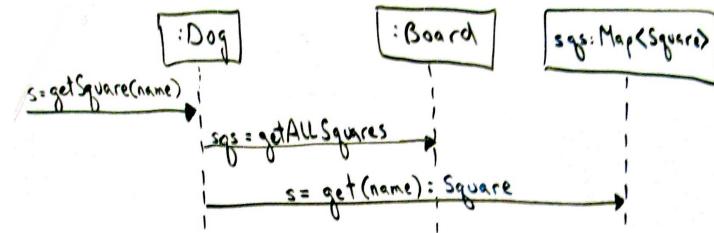
26

Couplage

- Exemples classiques de couplages de TypeX vers TypeY dans un langage OO
 - TypeX a un attribut qui référence à TypeY
 - TypeX a une méthode qui référence TypeY
 - TypeX est une sous-classe directe ou indirecte de TypeY
 - TypeY est une interface et TypeX l'implémente

♦ Sylvain Cherrier, Design Patterns

27

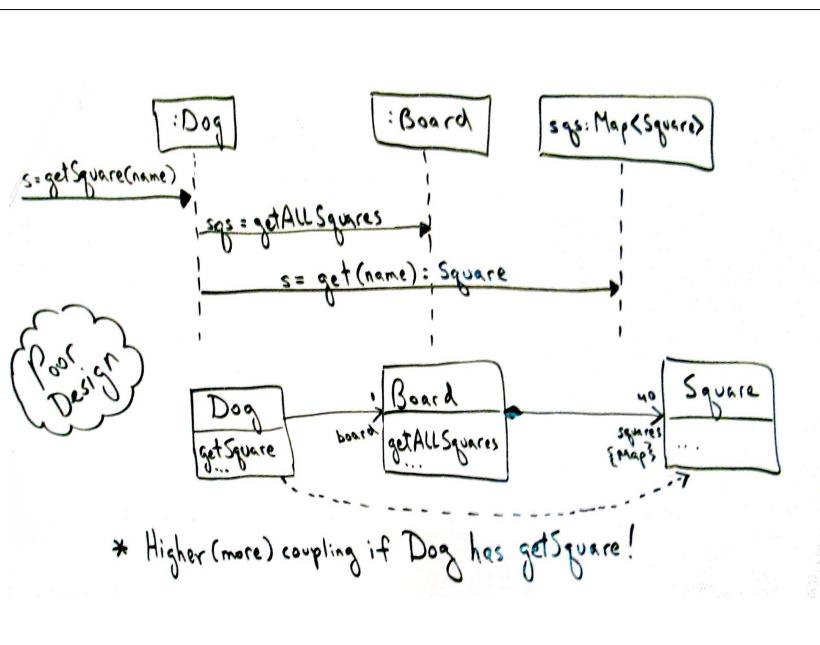


ISG - Grasp Solid - 23 January 2019

28-1

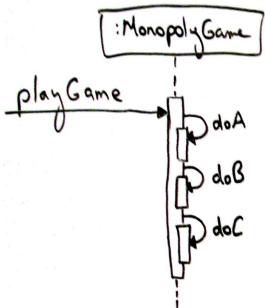
Forte Cohésion

- Problème :
 - Comment maintenir la complexité gérable ?
 - i.e., avoir des objets compréhensible et facile à gérer
- Solution :
 - Faire comme pour le faible couplage : mesurer, caractériser, choisir

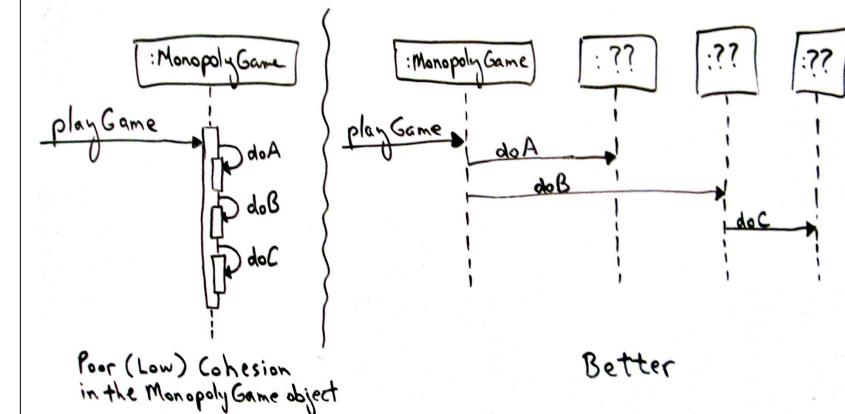


28-2

29



30-1

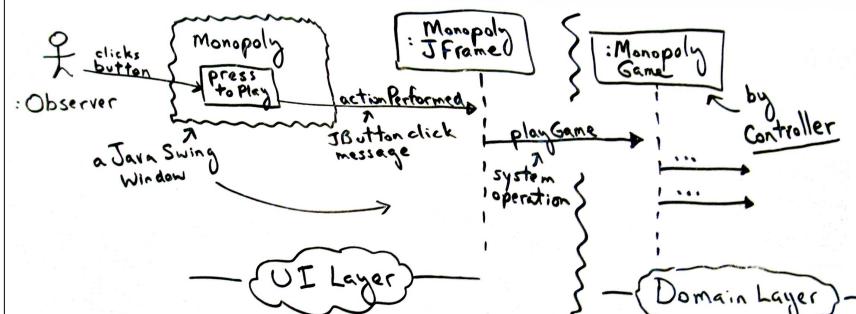


30-2

Contrôleur

- Problème :
 - Quel est le premier objet qui coordonne le système (après l'interface personne-machine)
- Solution :
 - Choisir (ou inventer) un objet qui endosse explicitement ce rôle

31

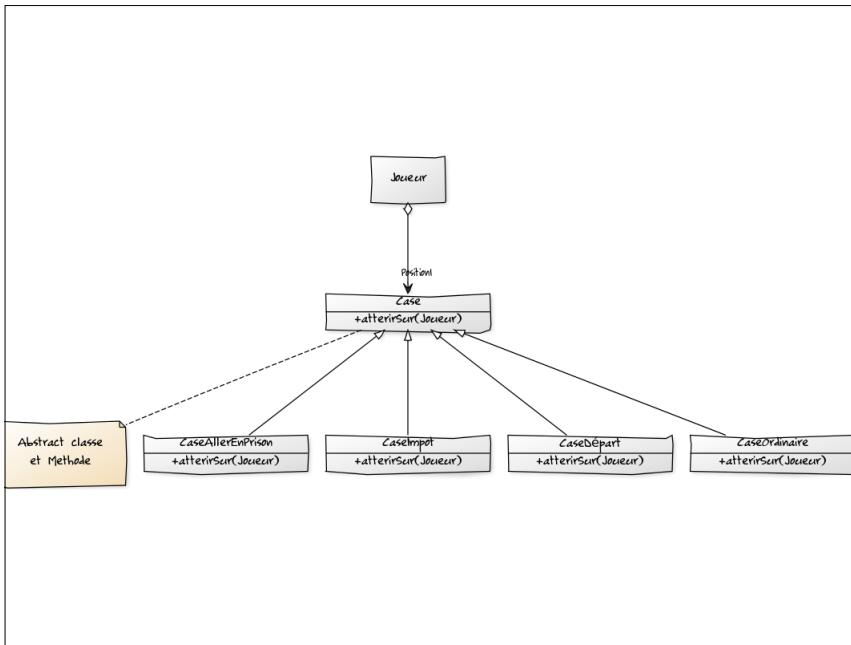


32

Polymorphisme

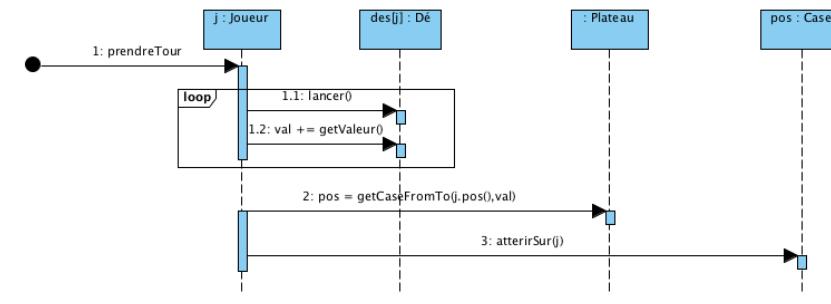
- Problème :
 - Comment créer des alternatives dépendantes des types ?
 - Comment créer des composants logiciels “enfichables” (puzzle)
- Solution :
 - Quand les fonctions varient en fonction du type, affecter les responsabilités au point de variation (pas de if/then/else sur des instanceof).

33



35

Effet des cases au Monopoly

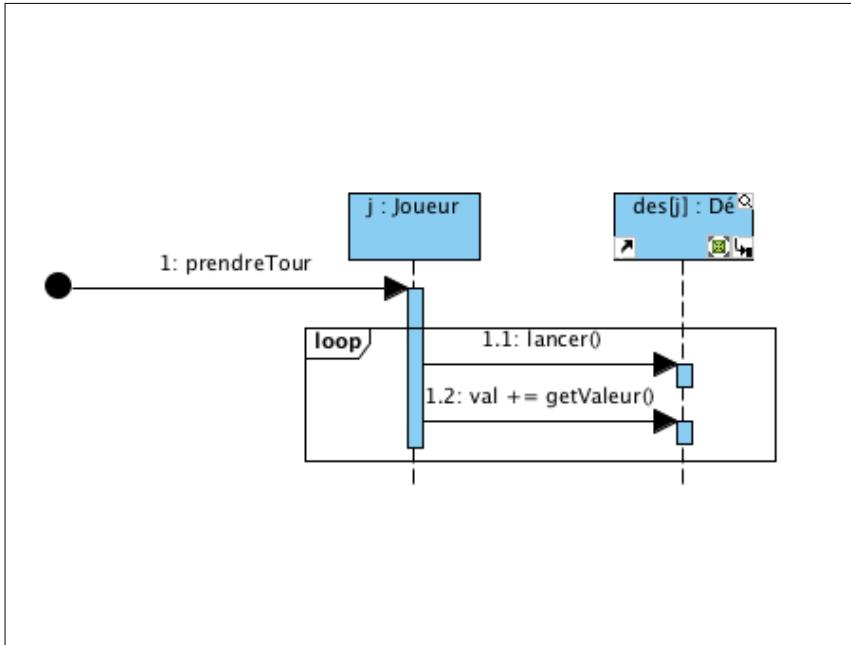


34

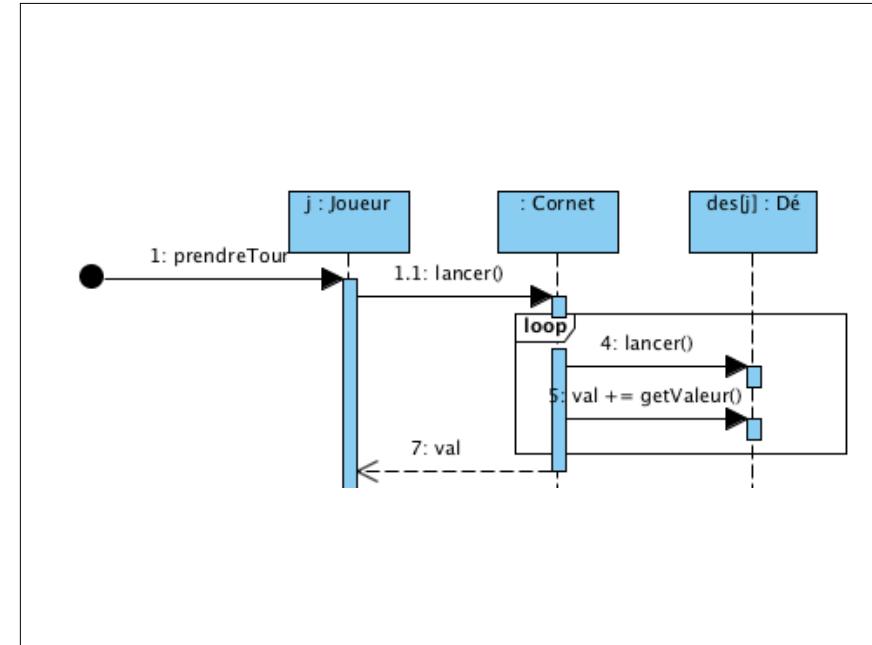
Invention pure

- Problème :
 - Que faire quand les éléments du monde réel ne sont pas utilisable vis à vis du respect d'un faible couplage et d'une forte cohésion ?
- Solution :
 - Fabriquer de toutes pièces une entité fortement cohésive et faiblement couplé

36



37



38



39



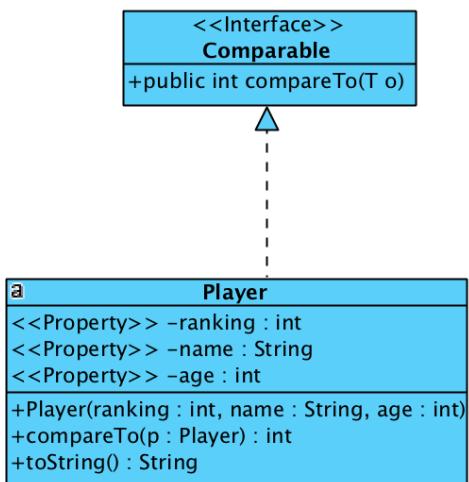
40

S.O.L.I.D : l'essentiel !

- **Single responsibility principle (SRP)** : une classe n'a qu'une seule responsabilité (ou préoccupation).
- **Open/closed principle (OCP)** : une classe doit être ouverte à l'extension (par héritage, par exemple) mais fermé à la modification (attributs privés, par exemple).
- **Liskov substitution principle (LSP)** : les objets d'un programme doivent pouvoir être remplacés par des instances de leurs sous-types sans «casser» le programme.
- **Interface segregation principle (ISP)** : il vaut mieux plusieurs interfaces spécifiques qu'une unique interface générique.
- **Dependency inversion principle (DIP)** : il faut dépendre des abstractions, pas des réalisations concrètes.

Simon Urli 2014

41



43

Responsabilité Unique

42

```
public class Player implements Comparable<Player> {

    private int ranking;

    private String name;

    private int age;

    public Player(int ranking, String name, int age) {
        this.ranking = ranking;
        this.name = name;
        this.age = age;
    }

    @Override
    public int compareTo(Player p) {
        return name.compareTo(p.name);
    }
}
```

44

Comment comparer sur autre chose ?

45

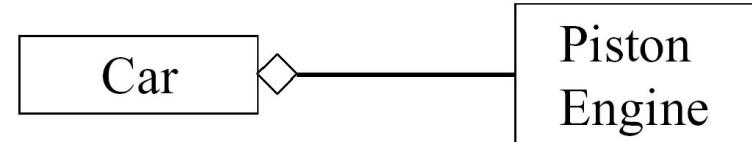
Ouvert
Fermé

47

```
class ByName implements Comparator<Student> {  
  
    public int compare(Student a, Student b) {  
        return a.name.compareTo(b.name);  
    }  
  
}
```

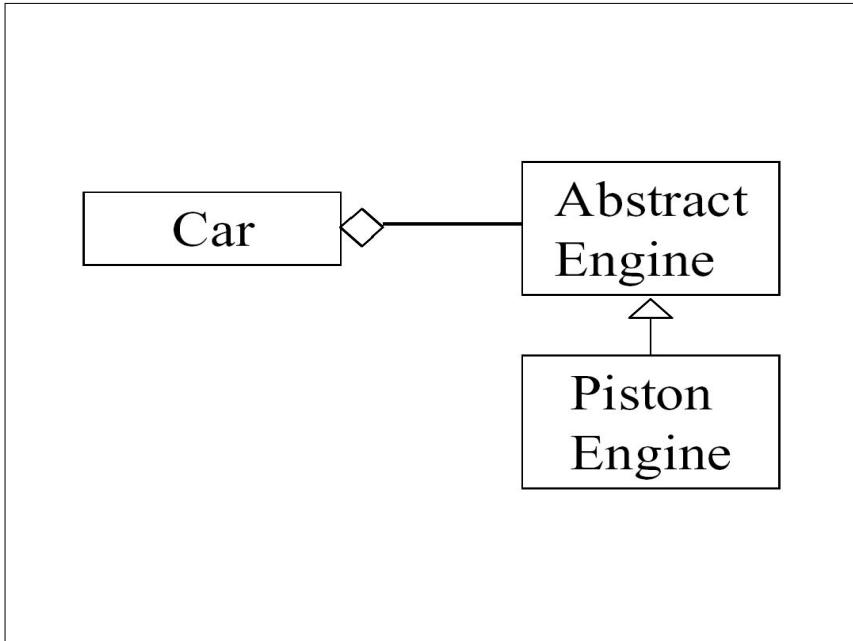
```
class BySection implements Comparator<Student> {  
  
    public int compare(Student a, Student b) {  
        return a.section - b.section;  
    }  
  
}
```

46

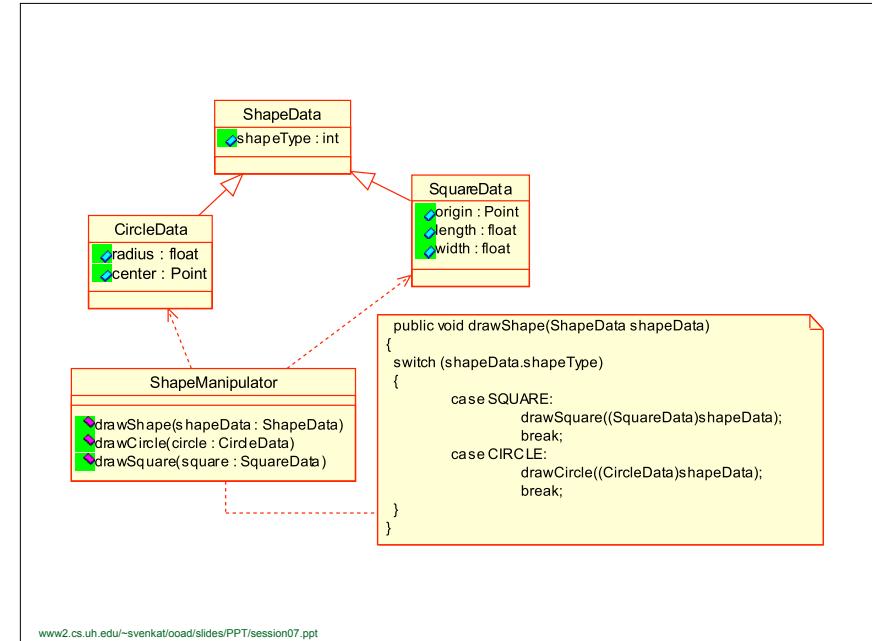


**Comment passer à
un moteur électrique ?**

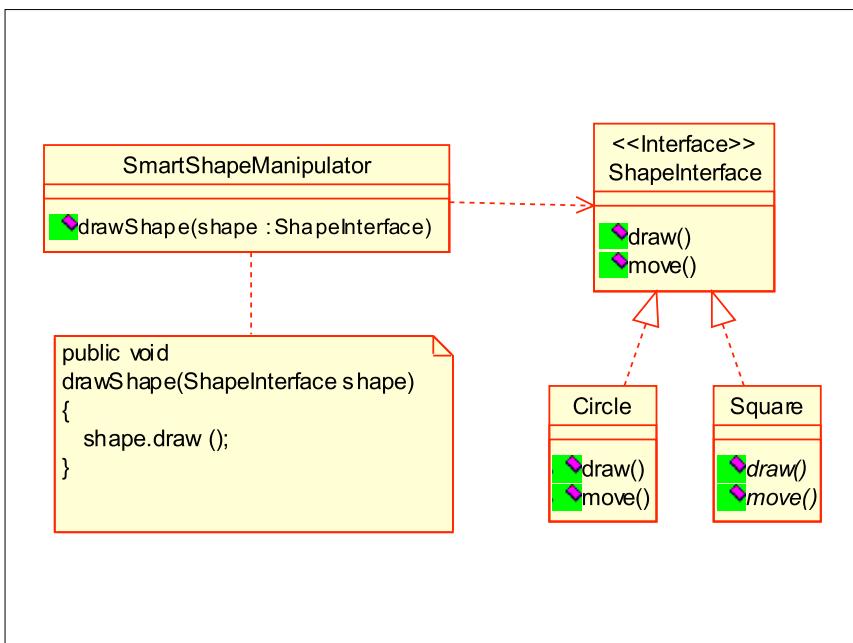
48



49



50



51



52

Principe de substitution de Liskov

Les instances d'une classe

doivent être **remplaçables** par des instances de leurs **sous-classes** sans altérer le programme.

53

Exemple : Les oiseaux volent ...

```
class Bird {                      // has beak, wings, ...
    public: virtual void fly();   // Bird can fly
};

class Parrot : public Bird {      // Parrot is a bird
    public: virtual void mimic(); // Can Repeat words...
};

// ...
Parrot mypet;
mypet.mimic();      // my pet being a parrot can Mimic()
mypet.fly();        // my pet "is-a" bird, can fly
```

55

Autrement dit

« Si une propriété P est vraie pour une instance x d'un type T, alors cette propriété P doit rester vraie pour tout instance y d'un sous-type de T »

54

Mais ni les autruches, ni les pingouins ...

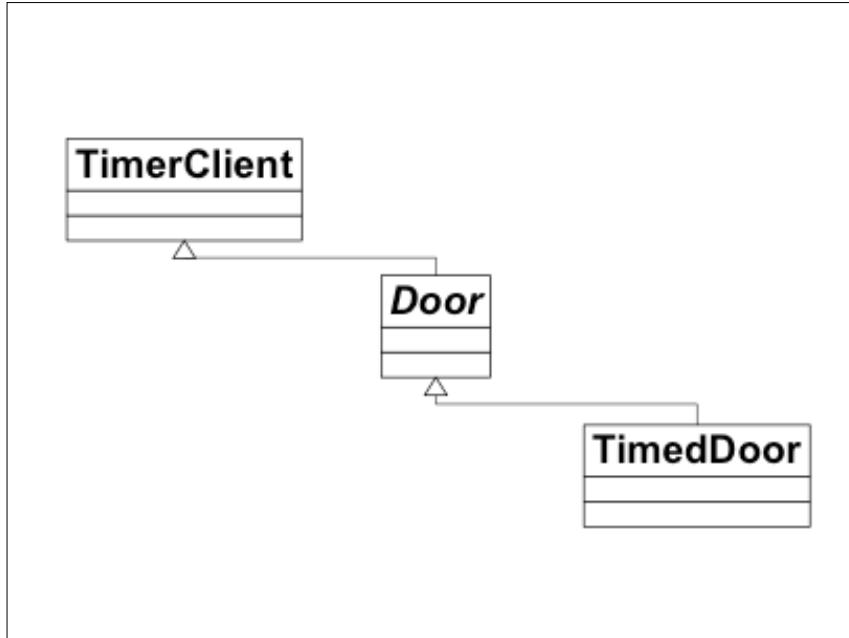
```
class Penguin : public Bird {
    public: void fly() {
        error ("Penguins don't fly!"); }
};
```

**Ceci ne modélise pas
“les pingouins ne savent pas voler”**

56

Ségrégation des interfaces

57

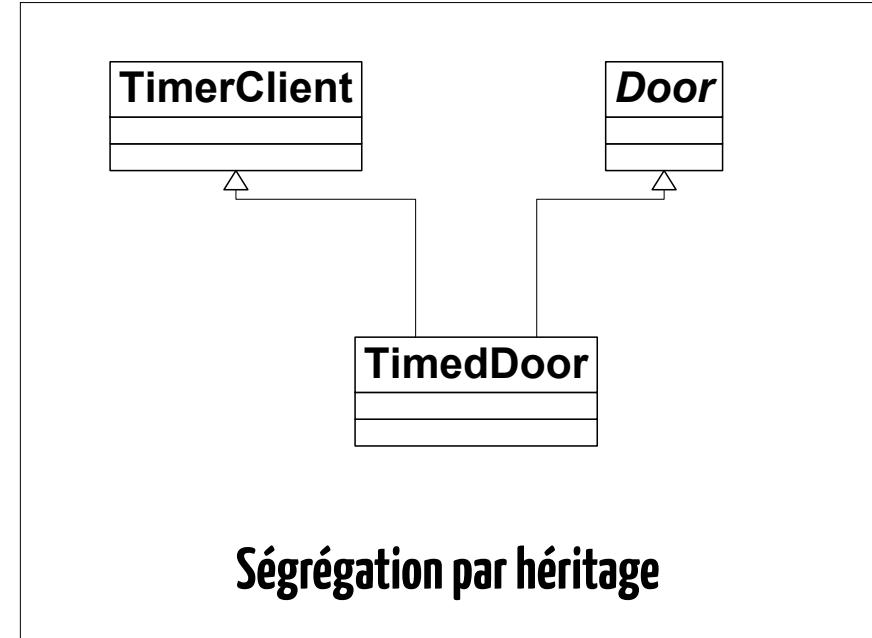


59

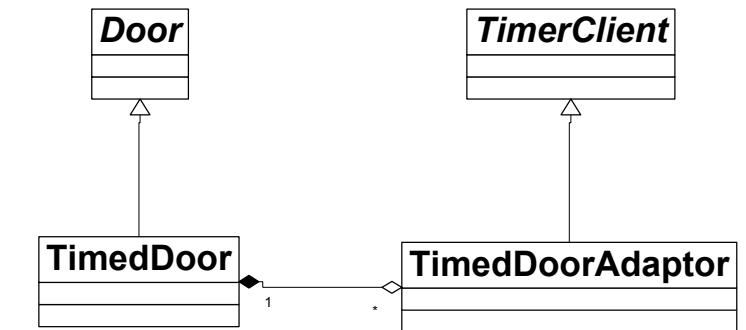
```
public interface Door {
    void lock();
    void unlock();
    void open();
    void close();
    boolean isOpen();
}
```

CSE 403, Spring 2008, Alverson

58



60



Ségrégation par composition

61

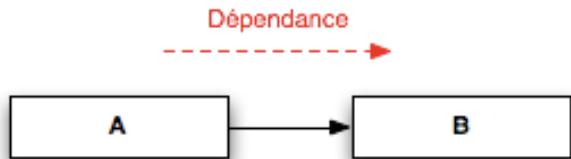
```

public class Drinker {

    public void DrinkBeer(FavoriteBar bar) {
        Beer beer = bar.OrderBeer();
        this.RaiseElbow();
        this.Drink(beer);
    }

}

```



63

Inversion des Dépendances

62

```

public interface Bar {
    Beer orderBeer();
}

public class Favorite implements Bar {
    public Beer orderBeer() { ... }
}

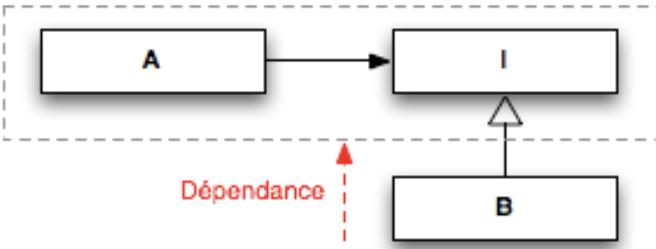
public class Drinker {

    public void DrinkBeer(Bar bar) {
        Beer beer = bar.OrderBeer();
        this.RaiseElbow();
        this.Drink(beer);
    }

}

```

64



65

```

public class Drinker {

    public void drink(OrderSupport bar,
                      BeverageType type) {
        Beverage bvg = bar.order(type);
        this.RaiseElbow();
        this.Drink(bvg);
    }

    public void drinkBeer(OrderSupport bar) {
        this.drink(bar, BeverageType.BEER);
    }
}
  
```

67

```

public interface OrderSupport {
    Beverage order(BeverageType type);
}

public class Favorite implements OrderSupport {
    public Beverage order(BeverageType type) {
        switch(type) {
            case BeverageType.BEER: ...
            default: throw new Exception();
        }
    }
}
  
```

66

Quid du principe de Liskov si on considère maintenant les bars à smoothies ?

68