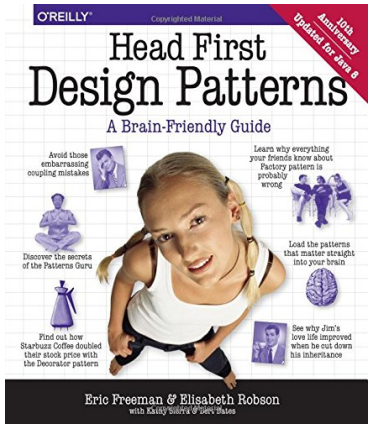
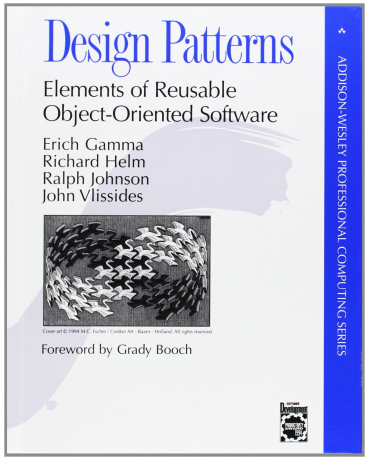
Génie Logiciel : Conception
Intro aux Patrons de Conception

Sébastien Mosser
INF-5153, Hiver 2019, Cours #5.1
UQÀM

Bibliographie



(le style est "discutable", mais le contenu raisonnable)



(référence, mais indigeste)

1 La danse des canards

2 Principes & Classification

3 Qui dit patron dit anti-patrons

La danse des canards

1

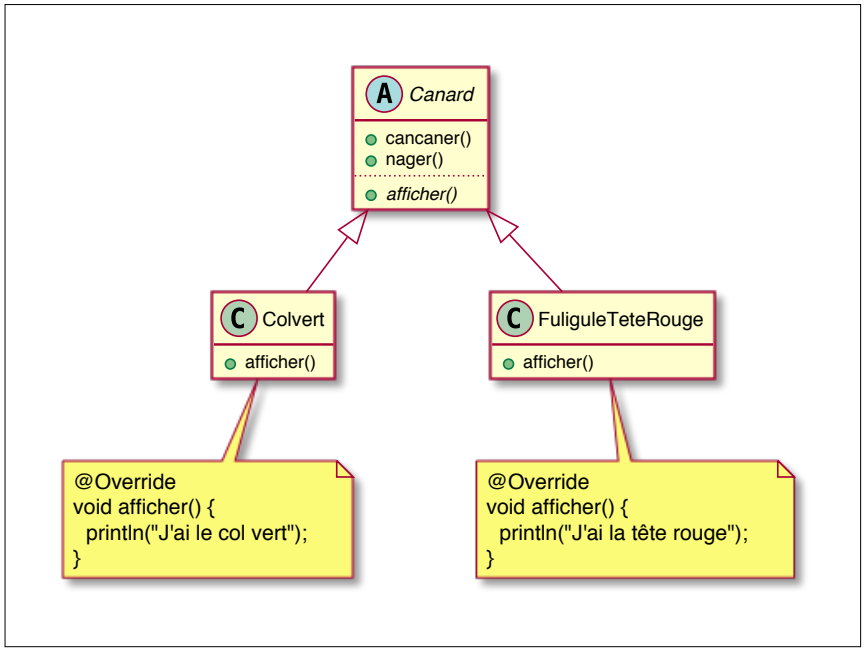
Jean-Michel Bruel
Professor of Software Engineering
University of Toulouse

5

Les canards savent ...

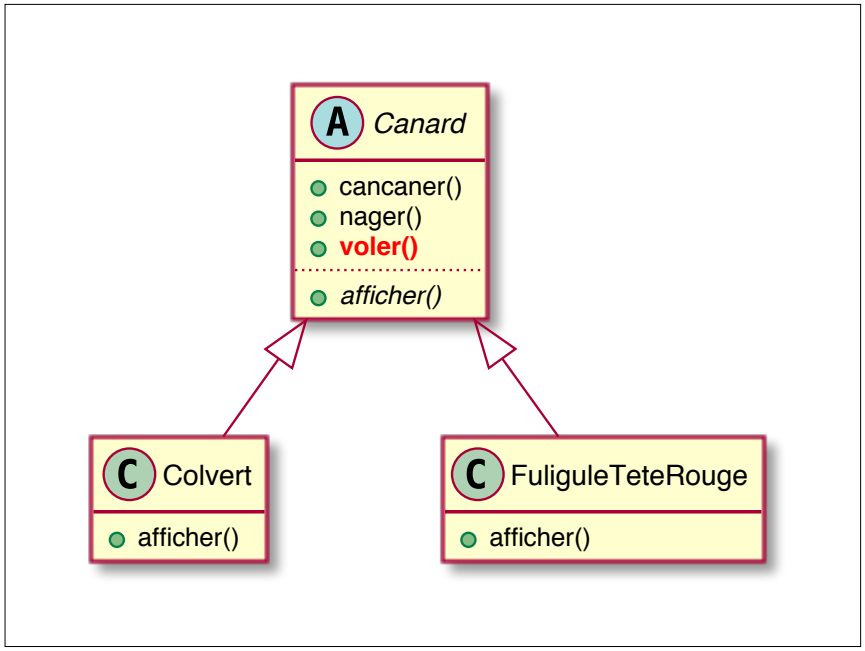
cancaner	Vrai pour toutes les espèces de canard
nager	
s'afficher	Dépend de l'espèce

6



7

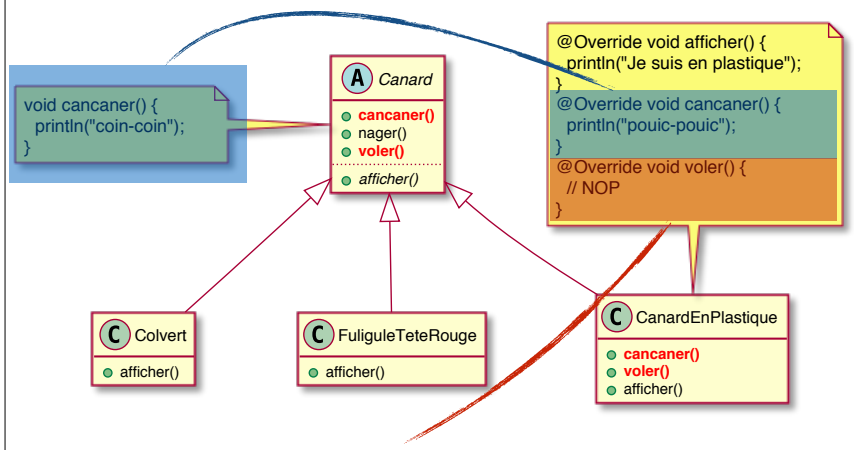
8



9

10

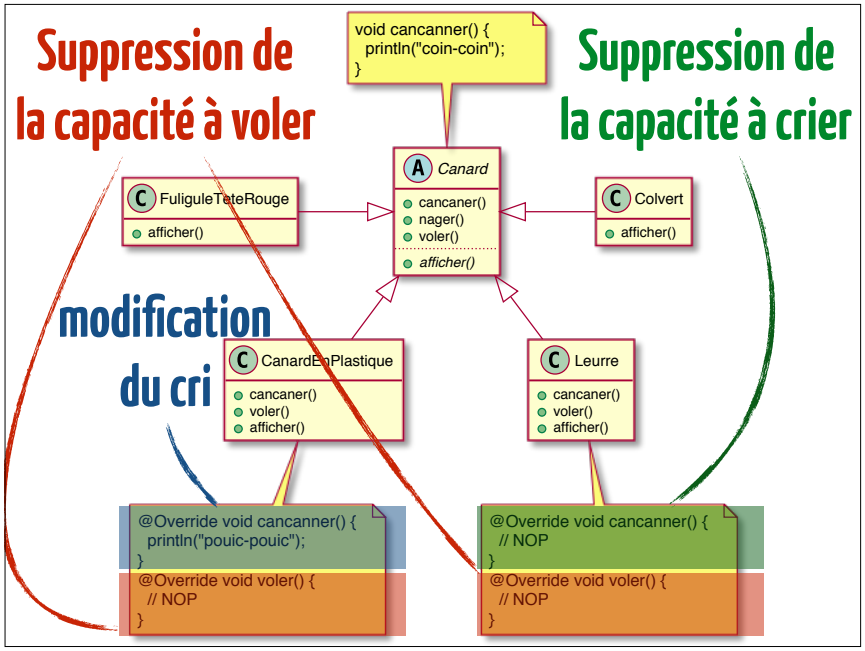
cri : “coin-coin” ≠ “pouic-pouic”



Suppression de la capacité à voler

11

12



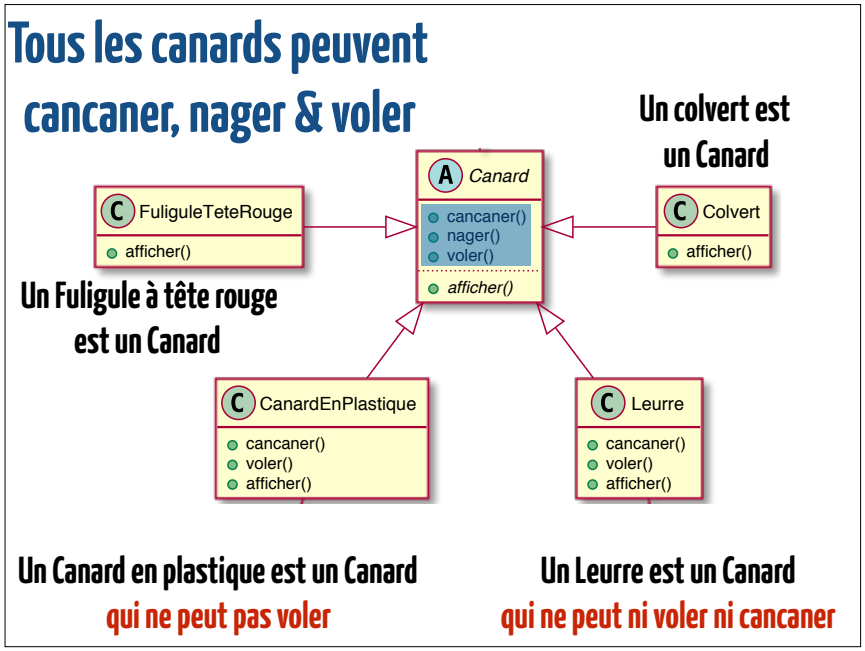
13

Est-ce raisonnable ?

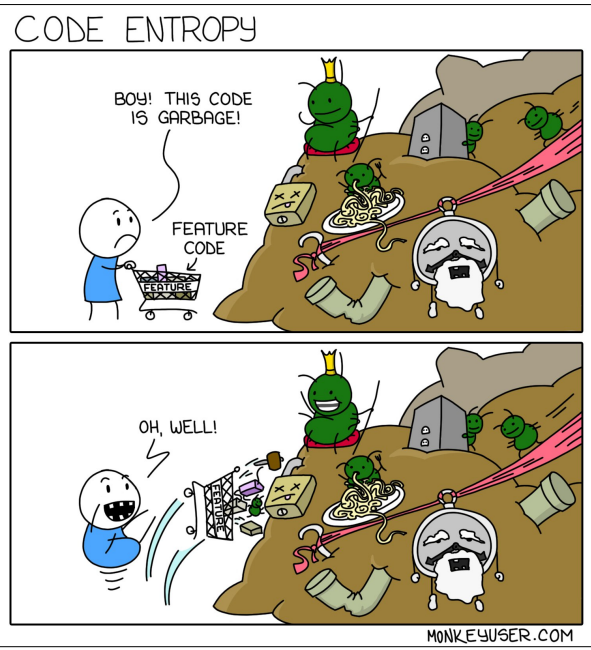
On respecte pourtant les piliers objets ...

Abstraction, Encapsulation, Polymorhpisme & Héritage

14



15



16

Et si notre problème, c'était en fait notre solution ?

Bref, utiliser l'héritage est-il la bonne approche ?

Interface plutôt qu'implémentation

Design Principle

Program to an interface, not an implementation.

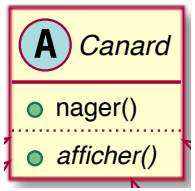
Trois Abstractions

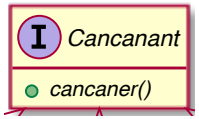
Structurelle :

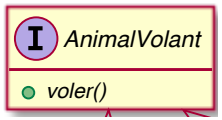
- **être un canard**

Fonctionnelle :

- **cancaner**
- **voler**





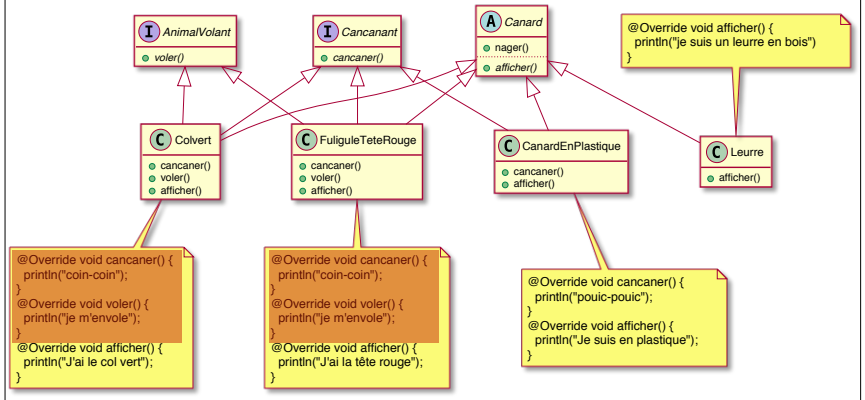


Création de canards à la carte !

- Colvert :
  - Canard ⊕ Cancanant ⊕ Animal Volant
- Fuligule à tête rouge :
  - Canard ⊕ Cancanant ⊕ Animal Volant
- Canard en plastique :
  - Canard ⊕ Cancanant
- Leurre :
  - Canard

```
public class Colvert
  extends Canard
  implements AnimalVolant, Cancanant { ... }

public class CanardEnPlastique
  extends Canard
  implements Cancanant { ... }

public class Leurre
  extends Canard { ... }
```

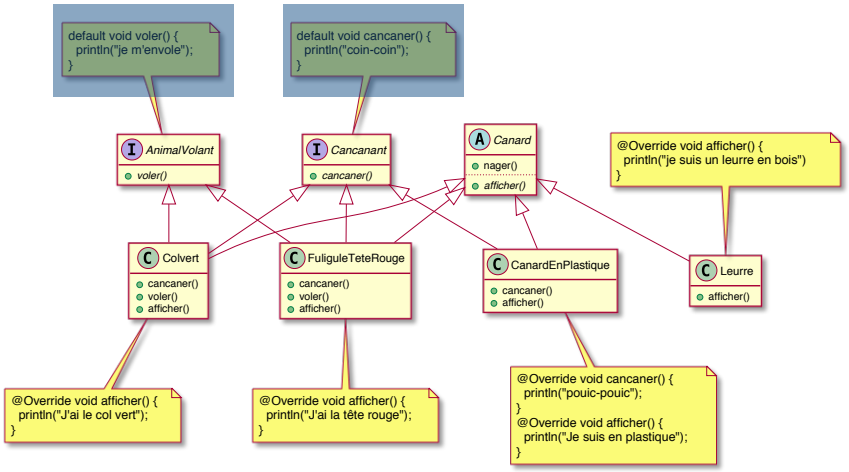
Au bout du compte ...



DRY: Don't Repeat Yourself

21

Java 8 propose des implémentation par défaut



22

On aurait pas juste déplacé le problème ?

```
Colvert colvert = new Colvert();
c.voler();

Canard unCanard = (Canard) new Colvert();
unCanard.voler() // Ne compile pas !!

if (unCanard instanceOf AnimalVolant) {
  ((AnimalVolant) unCanard).voler();
}
```

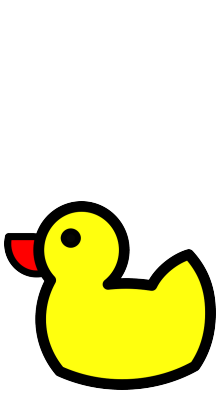
23

24

Identifier ce qui varie

Design Principle

Identify the aspects of your application that vary and separate them from what stays the same.

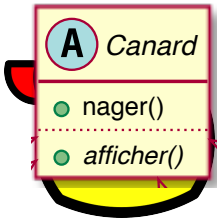


Être un Canard

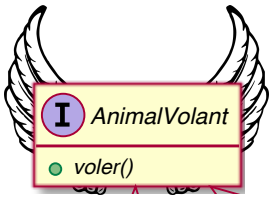
Voler

Faire du bruit

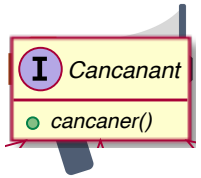
Wait, What ?



Être un Canard

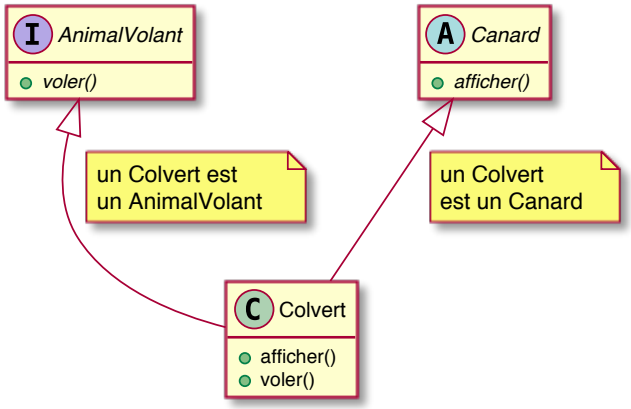


Voler



Faire du bruit

(Mauvaise) version avec des interfaces



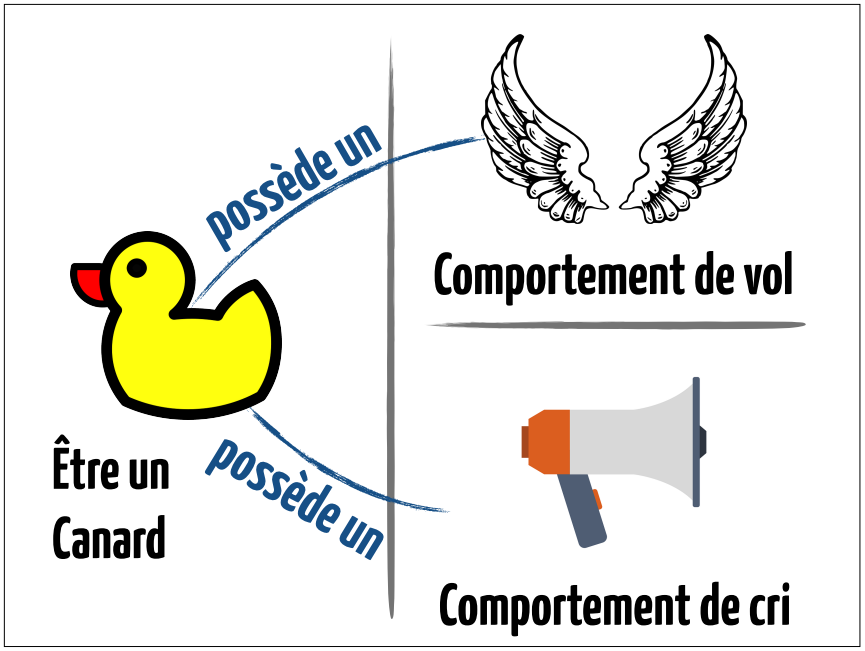
Héritage ≡ “est un”

Composition vs Héritage

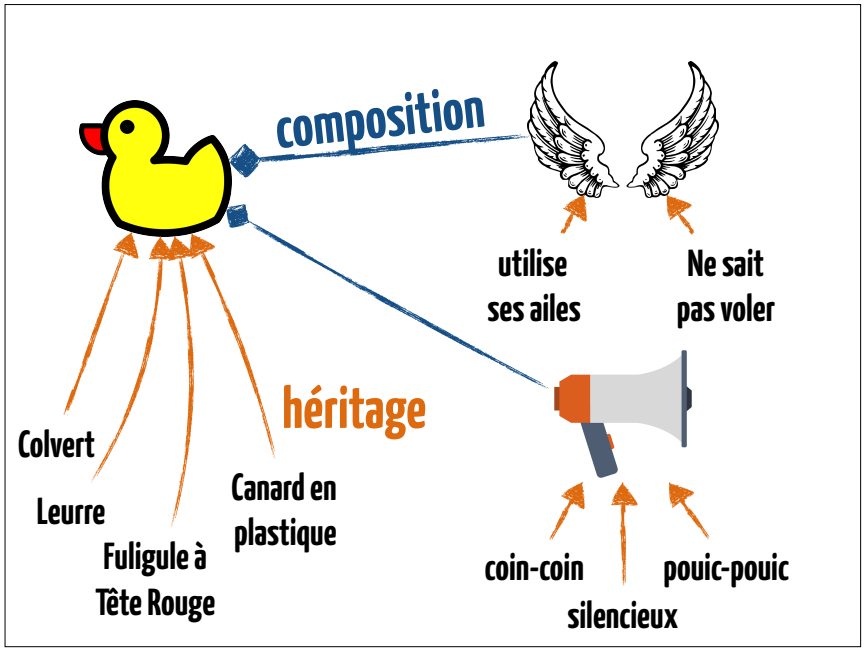
Design Principle

Favor composition over inheritance.

28

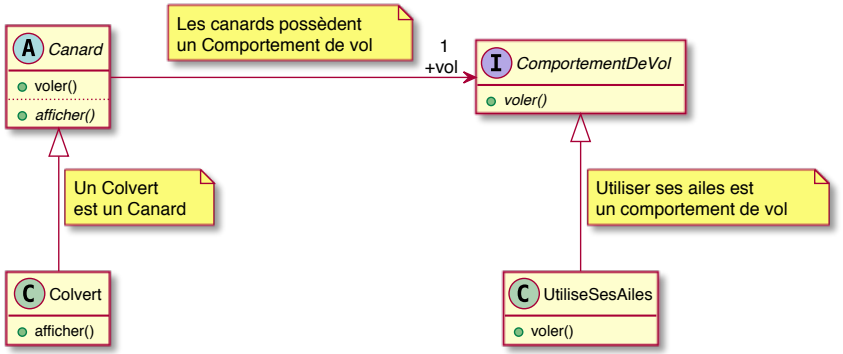


29



30

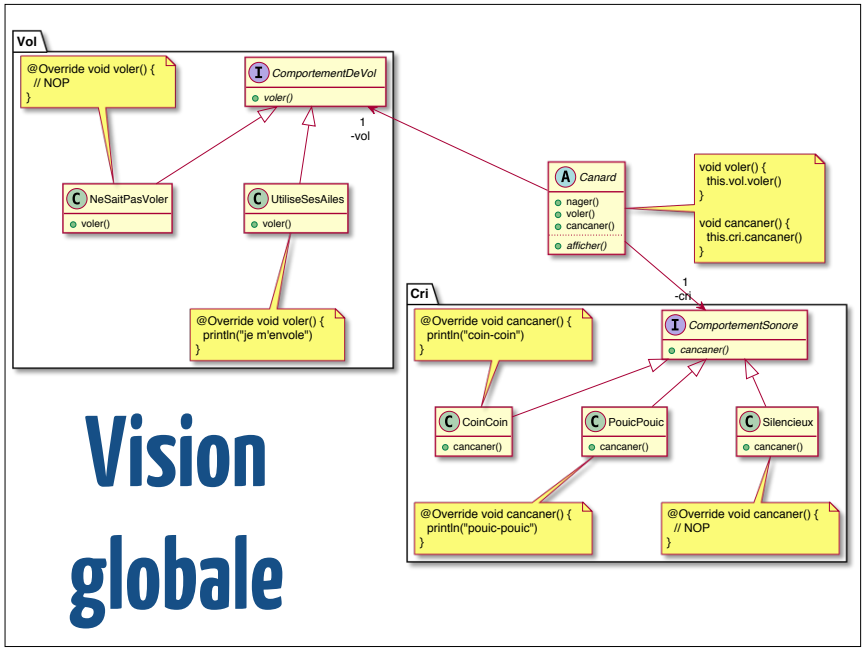
```
public abstract class Canard {
  private ComportementDeVol vol;
  public void voler() { this.vol.voler(); }
}
```



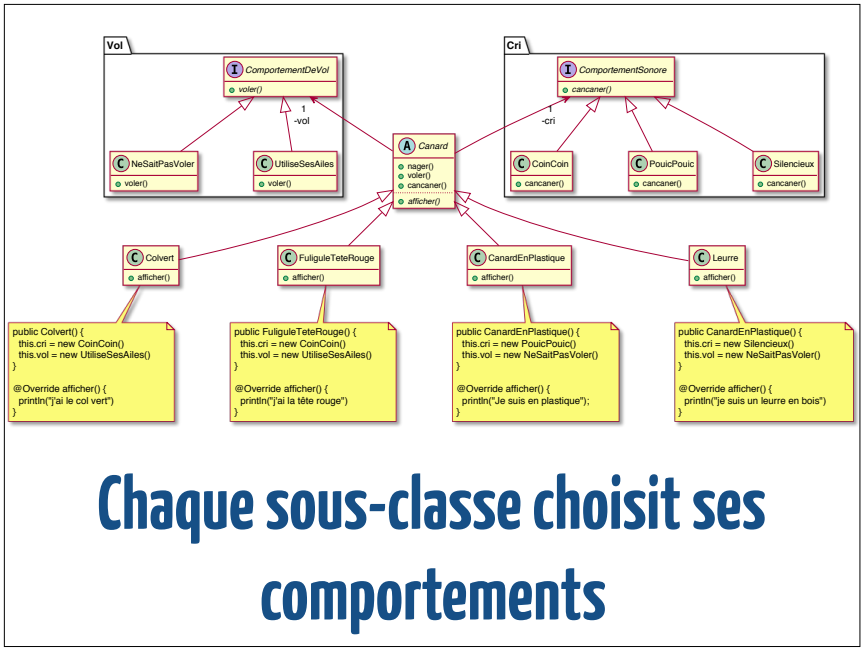

```
public class Colvert extends Canard {
  public Colvert() { this.vol = new UtiliseSesAiles(); }
}
```

UNIFIED MODELING LANGUAGE™

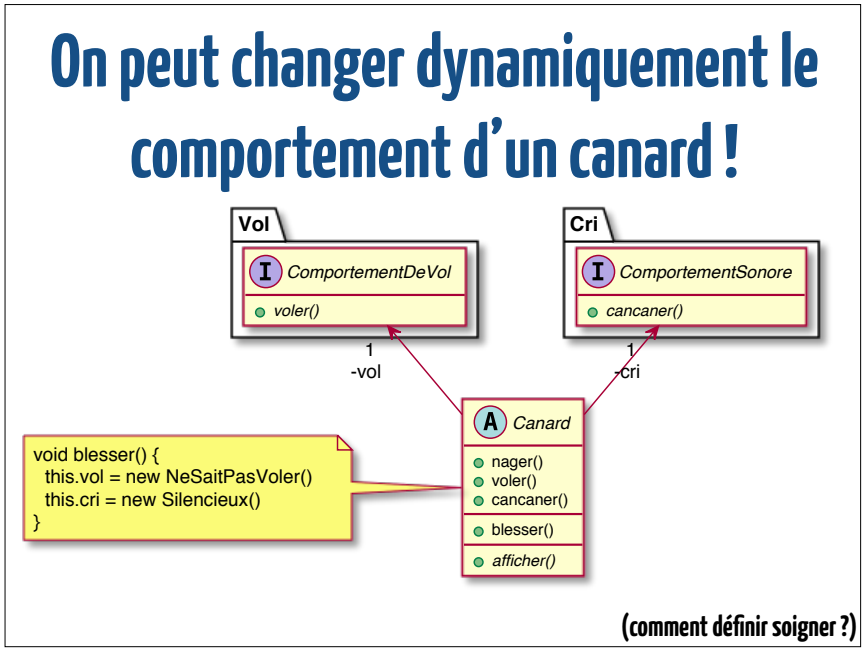
31


Vol
@Override void voler() {
// NOP
}
ComportementDeVol
voler()
1
-vol
NeSaitPasVoler
voler()
UtiliseSesAiles
voler()
@Override void voler() {
println("je m'envole")
}
Canard
nager()
voler()
cancaner()
afficher()
void voler() {
this.vol.voler()
}
void cancaner() {
this.cri.cancaner()
}
1
-cri
Cri
@Override void cancaner() {
println("coin-coin")
}
ComportementSonore
cancaner()
CoinCoin
cancaner()
PouicPouic
cancaner()
Silencieux
cancaner()
@Override void cancaner() {
println("pouic-pouic")
}
@Override void cancaner() {
// NOP
}
Vision globale

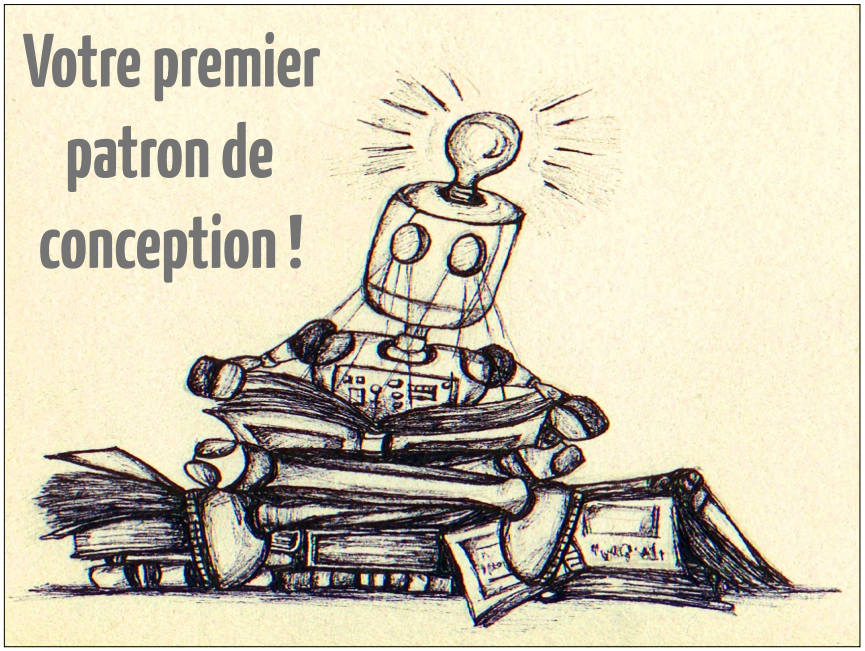
32


Vol
ComportementDeVol
voler()
1
-vol
NeSaitPasVoler
voler()
UtiliseSesAiles
voler()
Cri
ComportementSonore
cancaner()
1
-cri
CoinCoin
cancaner()
PouicPouic
cancaner()
Silencieux
cancaner()
Canard
nager()
voler()
cancaner()
afficher()
Colvert
afficher()
FuligulleTeteRouge
afficher()
CanardEnPlastique
afficher()
Leurre
afficher()
public Colvert() {
this.cri = new CoinCoin()
this.vol = new UtiliseSesAiles()
}
@Override afficher() {
println("j'ai le col vert")
}
public FuligulleTeteRouge() {
this.cri = new CoinCoin()
this.vol = new UtiliseSesAiles()
}
@Override afficher() {
println("j'ai la tête rouge")
}
public CanardEnPlastique() {
this.cri = new PouicPouic()
this.vol = new NeSaitPasVoler()
}
@Override afficher() {
println("Je suis en plastique");
}
public CanardEnPlastique() {
this.cri = new Silencieux()
this.vol = new NeSaitPasVoler()
}
@Override afficher() {
println("je suis un leurre en bois")
}
Chaque sous-classe choisit ses comportements

33


On peut changer dynamiquement le comportement d'un canard !
Vol
ComportementDeVol
voler()
1
-vol
Cri
ComportementSonore
cancaner()
1
-cri
Canard
nager()
voler()
cancaner()
blesser()
afficher()
void blesser() {
this.vol = new NeSaitPasVoler()
this.cri = new Silencieux()
}
(comment définir soigner ?)

34


Votre premier patron de conception !

35

STRATEGY — Object Behavioral

Intent

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Also Known As

Policy

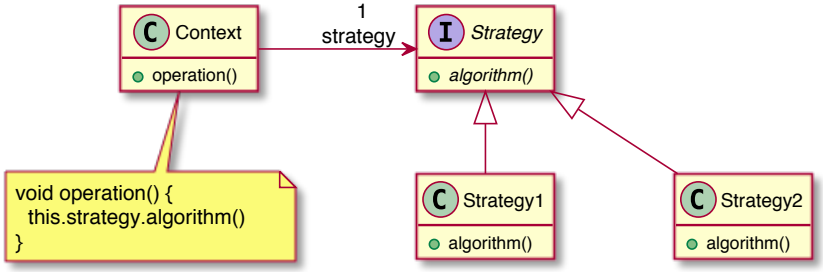
Motivation

Many algorithms exist for breaking a stream of text into lines. Hard-wiring all such algorithms into the classes that require them isn't desirable for several reasons:

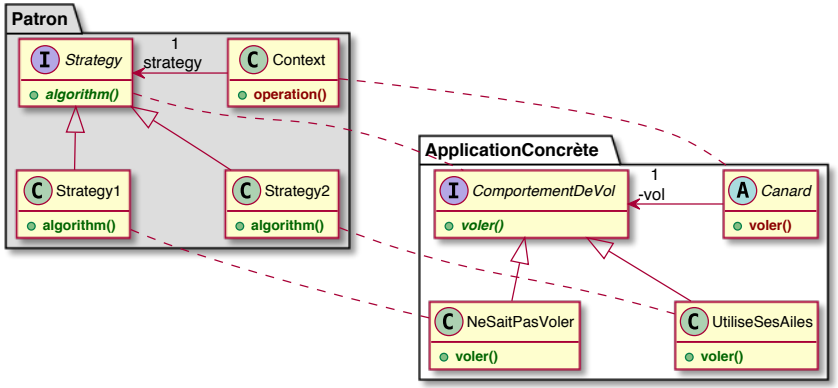
- Clients that need linebreaking get more complex if they include the linebreaking code. That makes clients bigger and harder to maintain, especially if they support multiple linebreaking algorithms.
- Different algorithms will be appropriate at different times. We don't want to support multiple linebreaking algorithms if we don't use them all.
- It's difficult to add new algorithms and vary existing ones when linebreaking is an integral part of a client.

We can avoid these problems by defining classes that encapsulate different linebreaking algorithms. An algorithm that's encapsulated in this way is called a **strategy**.

Le patron "Stratégie"



Des Canards Stratèges !



Principes & Classification

2

Principe des patrons de conceptions

- Standardisation des concepts de modélisation ;
- Capturer l'experience de conception ;
- Réutiliser des solutions élégantes & efficace pour des problèmes récurrents ;
- Améliorer la documentation ;
- Faciliter la maintenance.

Le cri des canards ? C'est une "Stratégie"

Capturer l'experience de conception

Stratégie {

Design Principle

Favor composition over inheritance.

Design Principle

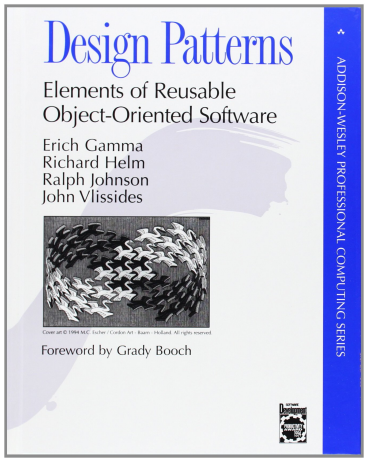
Identify the aspects of your application that vary and separate them from what stays the same.

Design Principle

Program to an interface, not an implementation.

Historique

- Livre "fondateur" publié en 1994
- Les auteurs sont le "Gang des Quatres"
- Proposition de 23 patrons de conceptions
- Séparation en 3 familles :
  - "Créationels"
  - Structurels
  - Comportementaux



Définition

Un patron de conception nomme, abstrait et identifie les aspects essentiels d'une structuration récurrente, ce qui permet de créer une modélisation orientée objet réutilisable

Dimension de classification

Portée	Objectif: Création	Objectif: Structure	Objectif: Comportement
classe			
objet			

Dimension de classification

Portée	Objectif: Création	Objectif: Structure	Objectif: Comportement
classe			
objet			Stratégie

Portée	Objectif: Création	Objectif: Structure	Objectif: Comportement
classe	Factory	Adapter	Interpreter Template Method
objet	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

INF-5153

Portée	Objectif: Création	Objectif: Structure	Objectif: Comportement
classe	**Factory**	**Adapter**	~~Interpreter~~ **Template Method**
objet	**Abstract Factory** **Builder** **Prototype** **Singleton**	**Adapter** ~~Bridge~~ **Composite** **Decorator** **Facade** ~~Flyweight~~ **Proxy**	~~Chain of Responsibility~~ **Command** ~~Iterator~~ ~~Mediator~~ ~~Memento~~ **Observer** **State** **Strategy** **Visitor**

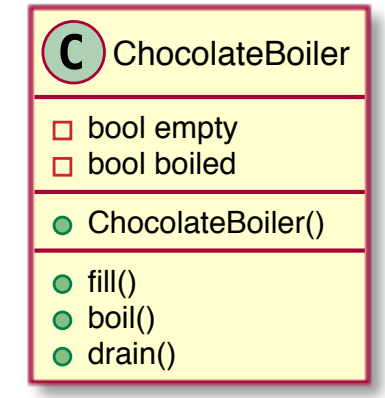
(Anti-)Patrons

3

47

```
public class ChocolateBoiler {

  private boolean empty;
  private boolean boiled;

  public ChocolateBoiler() {
    empty = true;
    boiled = false;
  }

  public void fill() {
    if (isEmpty()) {
      empty = false;
      boiled = false;
    }
  }

  public void drain() {
    if (!isEmpty() && isBoiled()) {
      empty = true;
    }
  }

  public void boil() {
    if (!isEmpty() && !isBoiled()) {
      boiled = true;
    }
  }
```

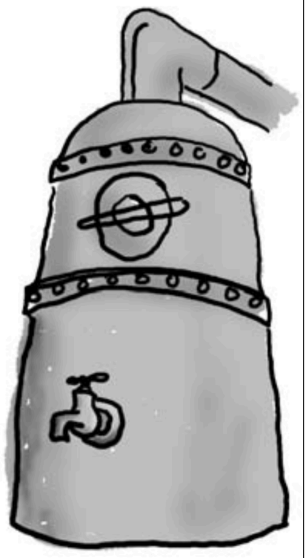
La fabrique de Chocolat



48

La fabrique de Chocolat

```
class ChocolateFactory {

  private ChocolateBoiler theBoiler;

  public ChocolateFactory() {
    this.theBoiler = new ChocolateBoiler();
  }

  public void makeChocolateBars() {
    // ... preprocessing
    if (theBoiler.isEmpty()) {
      theBoiler.fill();
      theBoiler.boil();
      theBoiler.drain();
    } else {
      throw new RuntimeException("Busy Boiler!");
    }
    // ... postprocessing
  }

}
```



49

```
public void makeChocolateTruffles() {
  // ... preprocessing
  theBoiler = new ChocolateBoiler();
  if (theBoiler.isEmpty()) {
    theBoiler.fill();
    theBoiler.boil();
    theBoiler.drain();
  } else {
    throw new RuntimeException("Busy Boiler!");
  }
  // ... postprocessing
}
```

50-1

```
public void makeChocolateTruffles() {
  // ... preprocessing
  theBoiler = new ChocolateBoiler();
  if (theBoiler.isEmpty()) {
    theBoiler.fill();
    theBoiler.boil();
    theBoiler.drain();
  } else {
    throw new RuntimeException("Busy Boiler!");
  }
  // ... postprocessing
}
```

Erreur de débutant

Manque de documentation

... whatever ...

50-2

Problème

N'importe qui peut créer une instance de ChocolateBoiler.

Pourtant il n'existe physiquement qu'un seul dispositif.

C'est au développeur de faire attention.

51

C'est au développeur de faire attention.

52-1

C'est au développeur de faire attention.

52-2

SINGLETON — Object Creational

Intent

Ensure a class only has one instance, and provide a global point of access to it.

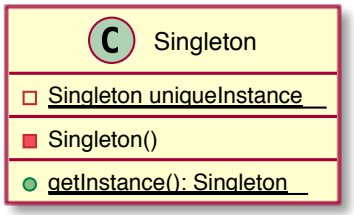
Motivation

It's important for some classes to have exactly one instance. Although there can be many printers in a system, there should be only one printer spooler. There should be only one file system and one window manager. A digital filter will have one A/D converter. An accounting system will be dedicated to serving one company.

How do we ensure that a class has only one instance and that the instance is easily accessible? A global variable makes an object accessible, but it doesn't keep you from instantiating multiple objects.

A better solution is to make the class itself responsible for keeping track of its sole instance. The class can ensure that no other instance can be created (by intercepting requests to create new objects), and it can provide a way to access the instance. This is the Singleton pattern.

53



Instance statique
Constructeur
} privé

Accesseur
} public

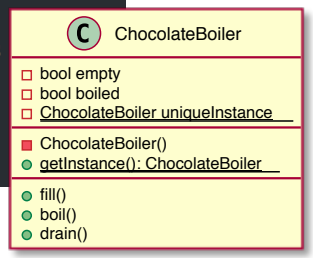
54

```
class Singleton {

  private static Singleton uniqueInstance = null;

  private Singleton() { ... }

  public static Singleton getInstance() {
    if (uniqueInstance == null)
      uniqueInstance = new Singleton();
    return uniqueInstance;
  }

      Singleton s = new Singleton();
}
```

```
Singleton s = Singleton.getInstance();
```

55

```
public class ChocolateBoiler {

  private boolean empty;
  private boolean boiled;

  private static ChocolateBoiler uniqueInstance = null;

  private ChocolateBoiler() {
    empty = true;
    boiled = false;
  }

  public static ChocolateBoiler getInstance() {
    if (uniqueInstance == null)
      uniqueInstance = new ChocolateBoiler();
    return uniqueInstance;
  }

}
```



56

```
class ChocolateFactory {

  private ChocolateBoiler theBoiler;
  public ChocolateFactory() {
    // this.theBoiler = new ChocolateBoiler();
    this.theBoiler = ChocolateBoiler.getInstance();
  }

  public void makeChocolateBars() {
    // ... preprocessing
    if (theBoiler.isEmpty()) {
      theBoiler.fill(); theBoiler.boil();
      theBoiler.drain();
    } else {
      throw new RuntimeException("Busy Boiler!");
    }
    // ... postprocessing
  }

  public void makeChocolateTruffles() {
    // ... preprocessing
    // theBoiler = new ChocolateBoiler();
    theBoiler = ChocolateBoiler.getInstance();
    if (theBoiler.isEmpty()) {
      theBoiler.fill(); theBoiler.boil();
      theBoiler.drain();
    } else {
      throw new RuntimeException("Busy Boiler!");
    }
    // ... postprocessing
  }
}
```

Même en cas d'erreur de programmation, on parle toujours à la même instance de ChocolateBoiler !

Il faut faire un tout petit peu plus attention en cas de multi-threading

57

STUPID code, seriously?

This may hurt your feelings, but you have probably written STUPID code already. I have too. But, what does that mean?

- Singleton
- Tight Coupling
- Untestability
- Premature Optimization
- Indescriptive Naming
- Duplication

Singleton, bon ou mauvais ?

https://williamdurand.fr/2013/07/30/from-stupid-to-solid-code/

58

Mais au fond quelle différence y a t il entre le bon et le mauvais chasseur ?

Bon y faut expliquer tu vois y'a le mauvais chasseur, y voit un truc qui bouge y tire. Le bon chasseur y voit un truc y tire, mais c'est un bon chasseur.

https://www.youtube.com/watch?v=QuGcoOJKXT8

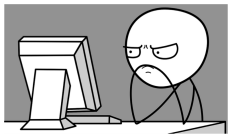
59

Singleton : un anti-patron ?

- Le bon singleton ? Il implémente une instance unique, mais c'est un bon singleton… NON
  - **Gestion d'un seule instance avec une responsabilité unique**
  - **Pas d'état, sur la gestion de l'instance**
  - Exemple : formatter, cache, logger, interface d'accès à du matériel
- Mauvais singleton ?
  - **Représentation d'un utilisateur qui vient de se logger**
  - **Représentation d'un plateau de jeu partagé**
  - **Facilité d'accès des valeurs dans plusieurs zones/couches de l'application**

60

Why singletons suck…



- **Graphe de dépendances entre objets caché**
- **Difficiles à tester**
  - En fait, c'est un couplage fort : en étant globaux, c'est tout leur environnement qui doit gérer leur état
  - Ils sont difficiles à « mocker », on doit écrire du code spécifique pour les tester
  - Ils ne sont pas vraiment extensibles par héritage
- **Pas bon pour la concurrence**
  - Ou pas *thread-safe*
  - Ou goulot d'étranglement en cas d'accès multiples et concurrents
- **Solution : Injection de dépendances**
  - cours [redacted] (ceci est un message publicitaire de Sébastien Mosser)

MGL 7361, A19