MOdélisation OBjet

5 - Introduction aux patrons de conception

Valentin Honoré

valentin.honore@ensiie.fr

FISA 1A

Plan du cours

Quelles sont les propriétés d'un bon logiciel ? (1/2)

Des idées ? Rappel des cours précédents...

Quelles sont les propriétés d'un bon logiciel ? (1/2)

- Il marche !
- Pas de comportements inattendus
- Facile à maintenir
- Facile à modifier (ajout de fonctionnalités etc)
- Bonnes performances

Quelles sont les propriétés d'un bon logiciel ? (2/2)

Concentrons nous sur l'aspect "propreté"

Make it work

1

Make it clean

2

Make it fast

3

Plan du cours

Définitions

Cohésion et **Couplage** sont des concepts **abstraits** et **complémentaires**

- **Cohésion**
  - A quel point un ensemble de code est assemblé en un composant cohérent
  - A quel point une classe/méthode possède **une responsabilité (Single Responsibility)**
    - → Le moins vaut le mieux ! ! !
- **Couplage**
  - A quel point un ensemble de code dépend du code qui l'entoure
  - A quel point **changer une classe implique changer les autres**

Quel niveau de cohésion et couplage peut-on désirer dans un bon logiciel ?

Définitions

Cohésion et **Couplage** sont des concepts **abstraits** et **complémentaires**

- **Cohésion**
  - A quel point un ensemble de code est assemblé en un composant cohérent
  - A quel point une classe/méthode possède **une responsabilité (Single Responsibility)**
    - → Le moins vaut le mieux ! ! !
- **Couplage**
  - A quel point un ensemble de code dépend du code qui l'entoure
  - A quel point **changer une classe implique changer les autres**

Il faut viser une **forte cohésion** et un **faible couplage** pour avoir une bonne maintenance de logiciel !

Cohésion : exemple

Exemple : machine à café qui moud du café (versus 1 machine à café et une machine à moudre)

- **Avantages ?**

- **Inconvénients ?**

Cohésion : exemple

Exemple : machine à café qui moud du café (versus 1 machine à café et une machine à moudre)

- **Avantages ?**
  - Facile à utiliser (un seul bouton)
  - Autosuffisante (ne dépend pas d'une autre machine)
    - → Le moins vaut le mieux ! ! !
- **Inconvénients ?**

Cohésion : exemple

Exemple : machine à café qui moud du café (versus 1 machine à café et une machine à moudre)

- **Avantages ?**
  - Facile à utiliser (un seul bouton)
  - Autosuffisante (ne dépend pas d'une autre machine)
    - → Le moins vaut le mieux ! ! !
- **Inconvénients ?**
  - Grosse machine (coûte cher)
  - Difficile à réparer (risque de devoir tout changer si une partie casse)

Cohésion : code associé

- **Avantages**
  - Facile à utiliser (*appeler une méthode unique*)
  - Autosuffisante (*ne dépend pas d'une autre classe*)
- **Inconvénients**
  - Gros code (*long à écrire*)
  - Difficile au *refactoring* (peu modulaire)

```
public class AutomaticCoffeeMachine {
  public Coffee brew() {
    var water = this.dispenseWater();
    var hotWater = this.preHeat(water);
    var beans = this.getCoffeeBean();
    var ground = this.grindCoffee(beans);
    var puck = this.tamp(ground);
    hotWater = this.boil(hotWater);
    return this.infuse(hotWater, puck);
    }
  private Water dispenseWater() { /* ... */ }
  private Water preHeat() { /* ... */ }
  private Water boil() { /* ... */ }
  private Ground grindCoffee() { /* ... */ }
  private Puck tamp() { /* ... */ }
  private Coffe infuse() { /* ... */ }
}
```

Cohésion : bilan ?

- **Faible couplage**
  - Pas de dépendances :)

- **Faible cohésion :(**
  - Trop de fonctionnalités :(
  - La machine est trop compliquée :(

```
public class AutomaticCoffeeMachine {
  public Coffee brew() {
    var water = this.dispenseWater();
    var hotWater = this.preHeat(water);
    var beans = this.getCoffeeBean();
    var ground = this.grindCoffee(beans);
    var puck = this.tamp(ground);
    hotWater = this.boil(hotWater);
    return this.infuse(hotWater, puck);
    }
  private Water dispenseWater() { /* ... */ }
  private Water preHeat() { /* ... */ }
  private Water boil() { /* ... */ }
  private Ground grindCoffee() { /* ... */ }
  private Puck tamp() { /* ... */ }
  private Coffe infuse() { /* ... */ }
}
```

Cohésion : comment l'augmenter ?

- **Comment faire ?**

```
public class AutomaticCoffeeMachine {
  public Coffee brew() {
    var water = this.dispenseWater();
    var hotWater = this.preHeat(water);
    var beans = this.getCoffeeBean();
    var ground = this.grindCoffee(beans);
    var puck = this.tamp(ground);
    hotWater = this.boil(hotWater);
    return this.infuse(hotWater, puck);
    }
  private Water dispenseWater() { /* ... */ }
  private Water preHeat() { /* ... */ }
  private Water boil() { /* ... */ }
  private Ground grindCoffee() { /* ... */ }
  private Puck tamp() { /* ... */ }
  private Coffe infuse() { /* ... */ }
}
```

Cohésion : comment l'augmenter ?

- **Comment faire ?**
  - Diviser la classe en sous-classes
    → utiliser **l'aggregation** et la **composition**

```
public record AutomaticCoffeeMachine  (
  WaterSource watersource,
  Heater heater,
  BeanSource beanSource,
  Grinder grinder,
  Tamper tamper ) {

    public Coffee brew() {
      var water = waterSource.getWater();
      var hotWater = heater.preHeat(water);
      var beans = beanSource.getCoffeeBean();
      var ground = grinder.grind(beans);
      var puck = tamper.tamp(ground);
      hotWater = heater.boil(hotWater);
      return this.infuse(puck, hotWater);
    }
    private Coffee infuse() { /* ... */ }
  }
```

Cohésion : comment l'augmenter ?

- **Comment faire ?**
  - Diviser la classe en sous-classes
    → utiliser **l'aggregation** et la **composition**
  - Bouger le code de l'unité de chauffage (annotation)
    → ou le supprimer (**programmation par aspect**)

```
public record AutomaticCoffeeMachine  (
  WaterSource watersource,
  // Heater heater,
  BeanSource beanSource,
  Grinder grinder,
  Tamper tamper ) {

    @PreHeat()
    public Coffee brew() {
      var water = waterSource.getWater();
      // var hotWater = heater.preHeat(water);
      var beans = beanSource.getCoffeeBean();
      var ground = grinder.grind(beans);
      var puck = tamper.tamp(ground);
      // hotWater = heater.boil(hotWater);
      return this.infuse(puck, hotWater);
    }
    private Coffee infuse() { /* ... */ }
  }
```

Plan du cours

Exemple : faire du café avec une moulin à café + machine à café

- **Avantages**
  - Outils spécialisés
    - → Peuvent être réutilisés
    - → Faciles à régler
  - Parties peuvent être remplacées individuellement
    - → Faciles à faire évoluer

- **Inconvénients**
  - Difficile à utiliser
    - → Besoin de suivre une séquence précise
  - Risque plus élevé de ratage
    - → Si une machine casse, plus de café

```
// init components
var ws = new WaterSource();
var cbs = new CoffeeBeanSource();
var h = new Heater(95);
var g = new Grinder(0.35);
var t = new Tamper(400);
var cp = new CoffeePress(9.5, 60);

// prepare
var water = ws.getWater("arabica");
var beans = cbs.getCoffeBeans();
var ground = g.grind(beans);
var puck = t.tamp(ground);
water = h.preHeat(water);
var hotWater = h.boil(water);

// brew coffee
coffee = new cp.brew(hotWater, puck);
```

Exemple : faire du café avec une moulin à café + machine à café

- **Avantages**
  - **Forte cohésion**
    - → Petites classes
    - → Faciles à coder
    - → **Single responsibility**

- **Inconvénients**
  - Fort couplage
    - → Grosse dépendance entre tous les outils
    - → Forte chance qu'un changement sur l'un implique un changement sur l'autre

Comment réduire le couplage ?

- Comment faire ?

```
var machine = new CoffeeMachine();

// BAD ===============
// brew() is tightly coupled with specific
      ingredients.

coffee = machine.brew( hotWater, ground, milk,
      sugar, caramel, cacao);
```

Comment réduire le couplage ?

- Réduire le nombre de paramètres ?

```
// BAD ============
// Should not create new ingredients // for each
      coffee we make
var ws = new WaterSource();
var bs = new BeanSource();
var ms = new MilkSource();

// Recipe is tightly coupled with ingredients
var latteReceipe = new Receipe( ws.getWayer,
      bs.getBeans(), ms.getMilk(),
            null, // no sugar
            null, // no caramel
            null, // no cacao);

// not easy to brew a latte
var latteMacciato = machine.brew(latteReceipe);
```

Comment réduire le couplage ?

- Réduire le nombre de paramètres ?
- Utiliser des patrons de conception
  - *facade*, *builder*, *singleton* etc

```
// GOOD: Use design patterns ============
// Facade
class LatteMacciatoFacade {
  Coffee brew() {
    // Singletons
    var w = WaterSource.INSTANCE.getWater()
    var b = CoffeeSource.INSTANCE.getBeans();
    var m = MilkSource.INSTANCE.getMilk();

    // builder
    var rb = new ReceipeBuilder()
    .water(w)
    .beans(b)
    .milk(m);

    return this.machine.brew(rb.build());
  }
}

// BAD =======
// Depend on many method calls
grinder.setLevel(Level.FINE);
grinder.setBeans(beans);
grinder.start();
var weight = 0;

while (weight < CoffeeWeight.ESPRESSO) {
  weight += grinder.continue();
}
grinder.stop();

var ground = grinder.getCoffeeGround();
```

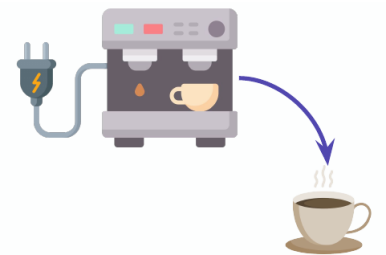
Comment réduire le couplage ?

- Réduire le nombre de paramètres ?
- Utiliser des patrons de conception
  - *facade*, *builder*, *singleton* etc
- Réduire le nombre de méthodes appelées

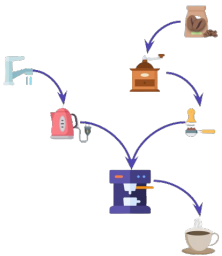
```
// GOOD ======
// Wrap dependents instructions
// into a single method
Ground grind(CoffeeBeans beans) {
  grinder.setLevel(Level.FINE);
  grinder.setBeans(beans);
  grinder.start();
  var weight = 0;
  while (weight < CoffeeWeight.ESPRESSO) {
    weight += grinder.continue();
  }
  grinder.stop();

  var ground = grinder.getCoffeeGround();
  return ground;
}
```

Au final, tout est une question de compromis !

- Faible couplage :)
- Faible cohésion :(



- Fort couplage :(
- Bonne cohésion :)



Le Graal : Forte cohésion et faible couplage

Ajuster entre cohésion et couplage est très souvent un équilibre de compromis !

- **Moins de classes**
  - → De plus grosses classes :(
  - → Moins de **cohésion** :(
  - → Faible **couplage** :)
- **Plus de classes**
  - → De plus petites classes :)
  - → Plus de **cohésion** :)
  - → Fort **couplage** :(
- L'utilisation des patrons de conception pour moins de **couplage** :
  - → *Builder*, *Singleton*, *Facade*, *Strategy etc* (cf la suite du cours)
- L'utilisation des patrons de conception pour plus de **cohésion** :
  - → *Proxy etc*

Plan du cours

S

O

L

I

D

Single Responsibility

O

L

I

D

Single Responsibility

Open / Close principle

L

I

D

Single Responsibility

Open / Close principle

Liskov Substitution

I

D

Single Responsibility

Open / Close principle

Liskov Substitution

Interface Segragation

D

Single Responsibility

Open / Close principle

Liskov Substitution

Interface Segragation

Dependency Inversion

Single Responsibility

- Une responsabilité par classe et méthode
  - → Meilleure cohésion :)
  - → Plus petites classes, plus facile à implanter

Listing – A ne pas faire...

```
class AutomaticCoffeeMachine {

  private Beans getBeans() { }
  private Ground grind(Beans beans) { }
  private Water getWater() { }
  private Water boil(Water water) { }
  private Puck tamp(Ground ground) { }
  public Coffee brew(Water w, Puck p) { }

}
```

Listing – Privilégier plutôt...

```
class Grinder {
  private Ground grind(Beans beans) { }
}
class Heater {
  private Water boil(Water water) { }
}
class Tamper {
  private Puck tamp(Ground ground) { }
}
class CoffeePress {
  public Coffee brew(Water w, Puck puck) { }
}
```

Open / Close Principle

- Laisser la possibilité de faire des extensions
  - → Il doit être facile d'ajouter de nouvelles fonctionnalités
- Fermé à la modification
  - → Ajouter de nouvelles fonctionnalités ne doit pas changer le code existant

Listing – A ne pas faire... (couplage ! !)

```
enum CoffeeType {
RISTRETTO, ESPRESSO , AMERICANO
}

class CoffeePress {
  public Coffee brew(CoffeeType type) {
    float duration;
    if (type == CoffeeType.RISTRETTO)
      duration = 20;
    if (type == CoffeeType.ESPRESSO)
      duration = 30;
    if (type == CoffeeType.AMERICANO)
      duration = 40;
  }
}
```

Listing – Privilégier plutôt...

```
enum CoffeeType {
  RISTRETTO (20),
  ESPRESSO (30),
  AMERICANO (40);

  private float d;
  CoffeeType(float d) { this.d = d; }
  float getDuration() { return this.d; }
}

class CoffeePress {
  public Coffee brew(CoffeeType type) {
    float duration = type.getDuration();
}
```

Liskov Substitution

- Soit T un type, et S un sous-type de T
- Si $q(T)$ est une propriété de T, cette propriété est aussi une de S
  $\rightarrow$ $q(T) \rightarrow q(S)$
- A tout moment, un type T peut être remplacé par un sous-type S sans casser le code

Listing – A ne pas faire... (mauvaise substitution)

```
interface Beans {}
class ArabicaBeans implements Beans {
}
class RobustaBeans implements Beans {
}
class FrenchBeans implements Beans {
}
class Grinder {
  Ground grind(Beans b) { /* ... */ }
}

new Grinder().grind(new FrenchBeans());
```

Listing – Privilégier plutôt... (meilleure abstraction)

```
interface CoffeeBeans extends Beans {}
class ArabicaBeans implements CoffeeBeans {
}
class RobustaBeans implements CoffeeBeans {
}
class FrenchBeans implements CoffeeBeans {
}
class Grinder {
  Ground grind(CoffeeBeans b) { /* ... */ }
}

newGrinder().grind(newArabicaBeans());
```

Interface Segragation

- Avec de grosses interfaces, beaucoup de fonctionnalités offertes
  - → Difficile à recomposer...
- Privilégier une composition de petites interfaces
- Avec de petites interfaces, beaucoup de fonctionnalités offertes
  - → Maintenance :)
  - → Testabilité :)
  - → Réutilisabilité :)
- Par exemple, un couteau suisse = couteau + tournevis + décapsuleur + etc

Interface Segragation

Listing – A ne pas faire...

```
interface CoffeeGround {
  void tamp();
  void infuse();
  void dose();
}


interface TeaBag {
  void infuse();
  void dose();
}
```

Listing – Privilégier plutôt... (interfaces réutilisables)

```
interface Tampable {
  void tamp();
}
interface Infusable {
  void infuse();
}
interface Dosable {
  void dose();
}

interface CoffeeGround extends Tampable, Infusable, Dosable {}

interface TeaBag extends Infusable, Dosable {}
```

Dependency Inversion

"Se baser sur des abstractions, et non des implantations"

- "J'ai besoin d'un marteau"
- et non
- "J'ai besoin du marteau en fer bleu de marque Parkside"

→ Faible couplage sur les détails d'implantation

→ Interchangeable avec d'autres implantations

Dependency Inversion

Listing – A ne pas faire... (dépendre des types concrets)

```
interface CoffeeBeans {}

class ArabicaBeans implements Beans { }
class RobustaBeans implements Beans { }

ArabicaBeans beans = new ArabicaBeans();
new Grinder().grind(beans);

class Grinder {
  Ground grind(ArabicaBeans beans) {/* ... */ }
}
```

Listing – Privilégier plutôt...

```
interface CoffeeBeans {}

class ArabicaBeans implements Beans { }
class RobustaBeans implements Beans { }

Beans beans = new ArabicaBeans();
new Grinder().grind(beans);

class Grinder {
  Ground grind(CoffeeBeans beans) {/* ... */ }
}
```

A retenir :

Single Responsibility

Open / Close principle

Liskov Substitution

Interface Segragation

Dependency Inversion

Plan du cours

DRY !

Une idée de ce que ça peut vouloir dire ?

DRY !

- **D**o not **R**epeat **Y**ourself
- Ne **jamais** copier-coller de (larges) portions de code
- Factoriser le code
  - Avec la composition
  - Avec l'héritage

Eviter de copier du code... mais pas que !

Listing – Ne pas faire...

```
class Kettle {
  // ...
  Water boil(Water water) {
    while(water.temperature < 100)
      water.temperature++;
  }
}

class CoffeeMachine {
  // ...
  Water boil(Water water) {
    while(water.temperature < 100)
      water.temperature++;
  }
}
```

Listing – Ne pas faire non plus...

```
class Kettle {
  // ...
  Water boil(Water water) {
    while(water.temperature < 100)
      water.temperature++;
  }
}

class CoffeeMachine extends Kettle {}
```

Mais alors que faire ?

Privilégier la composition à l'héritage ! [RAPPEL]

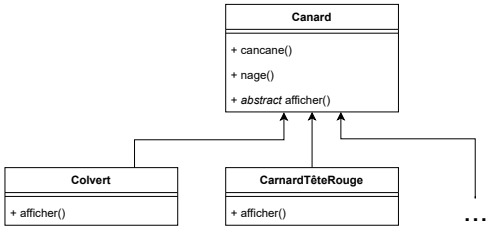
- Héritage :
  - La classe fille est très couplée à la classe mère
  - Héritage multiple génère des problèmes (cf juste après les patrons de conception)

- **Viser une forte cohésion**
- **La cohésion d'une classe/méthode a tendance a décroître quand sa taille augmente**

Plan du cours

Plan du cours

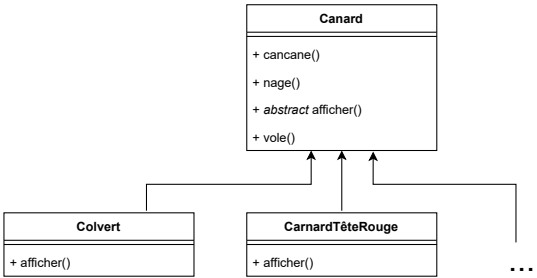
CanarApp1.0 : au pays des canards



- CanarApp : jeu (codé en Java) qui expose de nombreuses espèces (et pas que animales !) de canards qui nagent et cancannent
- une super-classe abstraite `Canard` dont toutes les espèces héritent
- Méthode abstraite `affiche()`qui permet d'afficher le carnard à l'écran
- Marcus a développé l'application dans sa startup `SimuDuck`

- Nouvelle version : Marcus doit faire voler les canards

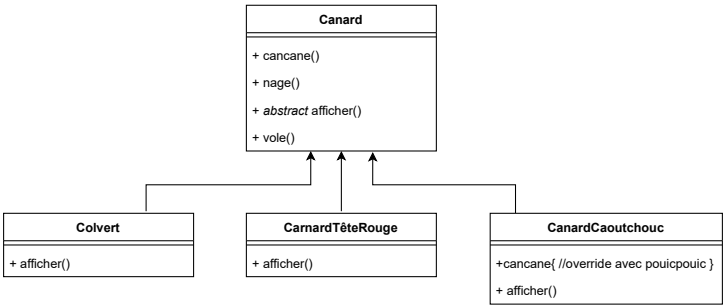
- Facile, il va juste ajouter une méthode `vole` dans la classe `Canard` !
  - ainsi, tous les canards en héritent !
  - utilisation des principes de la programmation objet



Mais il y a eu un gros problème lors de la démonstration de l'application...

- Des canards en caoutchouc qui volent, quelle diablerie !
  - Marcus réalise que toutes les sous-classes de canards de volent pas
  - Il y a des objets inanimés dans CanarApp

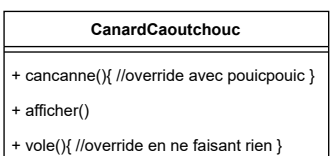
- Une mise à jour simple du code a causé des effets de bords
  - du principe de réutilisation avec l'héritage à un problème de maintenance du code :

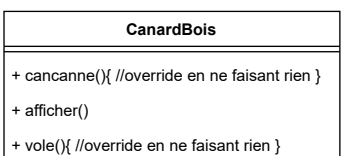


Que va faire Marcus ?

- Pourquoi ne pas *override* `vole()` ?
  - le faire pour la classe `CanardCaoutchouc`
  - comme pour la méthode `cancanne()`, mais en ne faisant rien pour `vole()`

- Mais que se passe-t-il quand on ajoute des canards de décoration en bois au programme ?
  - Ils ne volent pas :(
  - Ils ne cancannent pas :(
  - On doit *override* 2 méthodes pour ne rien faire





À cette étape, petite question :

- Quels sont les désavantages de l'héritage dans le cas de notre CanarApp ?
  1. Duplication de code entre sous-classes
  2. Des changements peuvent affecter des sous-classes de canard par effets de bord
  3. Les canards ne peuvent pas voler ni cancanner en même temps
  4. Difficile de modéliser le comportement de n'importe quel canard
  5. les changements de comportement à l'exécution sont difficiles à prévoir
  6. Les canards ne peuvent pas chanter

À cette étape, petite question :

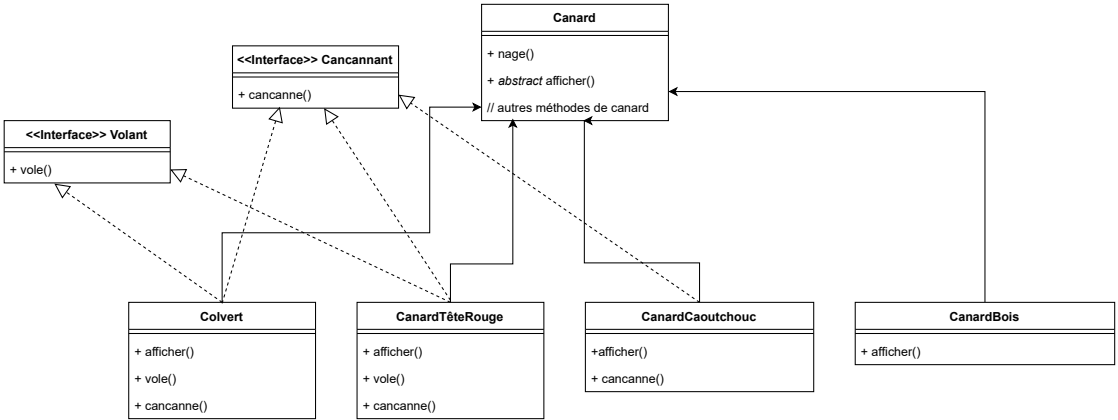
- Quels sont les désavantages de l'héritage dans le cas de notre CanarApp ?
  1. Duplication de code entre sous-classes
  2. Des changements peuvent affecter des sous-classes de canard par effets de bord
  3. Les canards ne peuvent pas voler ni cancanner en même temps
  4. Difficile de modéliser le comportement de n'importe quel canard
  5. les changements de comportement à l'exécution sont difficiles à prévoir
  6. Les canards ne peuvent pas chanter

Réponses : 1, 2, 4, 5

Retour au *brainstorming* de Marcus

- L'héritage ne semble pas être la bonne réponse
  - nouvelle contraite de son patron : mise à jour du produit tous les 6 mois
  - du coup, si la *spec* change, il va falloir verifier tous les *override* de chaque nouvelle sous-classe...
  - à jamais !
- Besoin d'avoir certains canards (mais pas tous) qui volent ou cancannent
  - **Idée :** super-type (="interface") `Volant` avec une méthode `vole()`
  - Comme ça, ceux qui volent n'auront qu'à la mettre en œuvre !
  - Idem avec un super-type `Cancannant` comme tous les canards ne cancannent pas...
- Voilà ce que propose Marcus !

Proposition de Marcus : CanarApp1.2



- **Qu'en pensez vous ?**

Que feriez-vous à la place de Marcus ?

- L'implantation des interfaces par les sous-classes
  - Plus de canards en caoutchouc volants et de canards en bois qui parlent
  - Mais quel est le plus gros problème qui est apparu ?

Que feriez-vous à la place de Marcus ?

- L'implantation des interfaces par les sous-classes
  - Plus de canards en caoutchouc volants et de canards en bois qui parlent
  - Mais quel est le plus gros problème qui est apparu ?

- Destruction complète de la réutilisabilité du code pour ces deux comportements !
  - chaque sous-classe réimplante une fonction qui est sûrement la même au final
  - donc re-cauchemar de maintenance :((exemple : petit changement dans la fonction `vole()`)

Que feriez-vous à la place de Marcus ?

- L'implantation des interfaces par les sous-classes
  - Plus de canards en caoutchouc volants et de canards en bois qui parlent
  - Mais quel est le plus gros problème qui est apparu ?

- Destruction complète de la réutilisabilité du code pour ces deux comportements !
  - chaque sous-classe réimplante une fonction qui est sûrement la même au final
  - donc re-cauchemar de maintenance :((exemple : petit changement dans la fonction `vole()`)

- Devant sa limonade, Marcus réfléchit...
  - il faudrait pouvoir contruire un logiciel qui, quand on doit le changer, l'impact sur le code serait minime
  - on passerait moins de temps à retravailler le code, et plus à ajouter des fonctionnalités supplémentaires

Aparté : développement logiciel

- **Quelle est la constante inévitable en développement logiciel ?**
  - TNEMEGNAHC

Aparté : développement logiciel

- **Quelle est la constante inévitable en développement logiciel ?**
  - `TNEMEGNAHC`

- Peu importe la qualité du développement de votre logiciel :
  - il va grossir
  - il va changer
  - ou... il va mourrir (RIP)

Aparté : développement logiciel

- **Quelle est la constante inévitable en développement logiciel ?**
  - `TNEMEGNAHC`

- Peu importe la qualité du développement de votre logiciel :
  - il va grossir
  - il va changer
  - ou... il va mourrir (RIP)

- Quelles peuvent-être les raisons de la nécessité de mise à jour d'un logiciel ?

Aparté : développement logiciel

- **Quelle est la constante inévitable en développement logiciel ?**
  - TNEMEGNAHC

- Peu importe la qualité du développement de votre logiciel :
  - il va grossir
  - il va changer
  - ou... il va mourrir (RIP)

- Quelles peuvent-être les raisons de la nécessité de mise à jour d'un logiciel ?
  - Ajout d'une nouvelle fonctionnalité
  - Changement de format de données/base(s) de données
  - *Refactoring* : tout casser et reprendre à zéro
  - Changements technologiques : protocoles, langage, compilateur etc

Revenons au pauvre Marcus

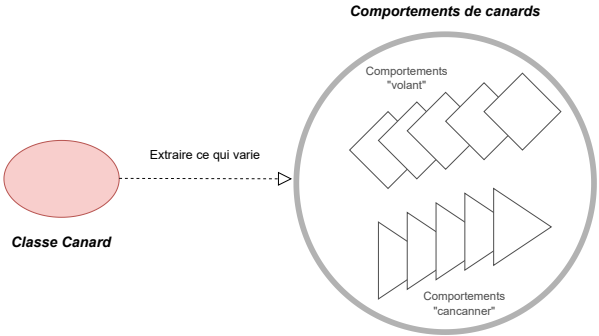
- Héritage ne marche pas (tout le monde hérite), les interfaces Java non plus (duplication possible de code)
  - Obligation de traquer les comportements à chaque changement
  - Déverminage à l'infini !

- Solution ? **Identifier les comportements qui varient et les séparer de ceux qui restent constants !**
  - Principe universel quand on développe un logiciel
  - Encapsuler ce qui varie pour que cela n'affecte pas le reste du code
  - Résultat ? moins d'effets de bord et plus de flexibilité ! !

- Base de tous les **patrons de conception** (*design patterns*)
  - Laisser varier des parties du système indépendamment des autres parties
  - Il est temps d'aider Marcus à extraire les comportements de canard des sous-classes de `Canard`

Séparer ce qui change de ce qui est immuable : par où commencer ?

- Classe `Canard` semble bien marcher à part les problèmes `vole()` et `cancanne()`
  - la classe `Canard` va rester telle quelle
  - besoin de rajouter des ensembles de classes en fonction de ce qui change
- Créer deux ensembles de classes (totalement décorrélées)
  - un pour le comportement "voler" & un pour le comportement "cancanner"
  - chaque ensemble sera en charge d'implanter leurs comportements respectifs
  - exemple : classe qui code "cancanner", une qui code "pouic" et une autre qui code le silence



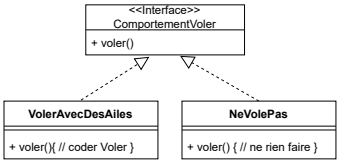
CanarApp2.0 : les comportements de canards (1/3)

- Comment modéliser l'ensemble des classes qui implantent les comportements "voler" et "cancanner" ?
  - Marcus veut pouvoir assigner un comportement à une instance de `Canard`, par exemple "voler"
  - *Exemple :* instancier un colvert et l'initialiser avec un type de comportement "voler" particulier, et même pouvoir en changer

- **Programmation par super-type, et non programmation par implantation**
  - Marcus va utiliser une interface pour chaque comportement (ex : "voler" et "cancanner")
  - chaque implantation d'un comportement va mettre en œuvre une de ces interfaces

- `Canard` ne s'occupe plus de coder les comportements "voler" et "cancanner"
  - les sous-classes de `Canard` utiliseront un comportement représenté par une interface
  - le code concret de ces comportements ne sera plus bloqué dans ces sous-classes de `Canard`

CanarApp2.0 : les comportements des canards (2/3)



- Les classes `Canard` n'ont besoin de connaître aucun détails d'implantation de leurs propres comportements !

- Programmation par super-type VS programmation par implantation
  - exploiter le polymorphisme par un super-type pour que le code d'un objet qui s'exécute ne soit pas "vérouillé"
  - "le type réel d'un objet devrait être un super-type (classe abstraite ou interface) afin que les objets assignés à ces variables puissent être de n'importe quelle implantation concrète de ce super-type"
  - la classe déclarante n'a pas besoin de connaître le type réel des objets !

CanarApp2.0 : les comportements de canards (3/3) [Exemple]

- Programmer par implantation

```
Chien d = new Chien();
d.aboyer(); // concrete implementation
```

- Programmer avec super-type

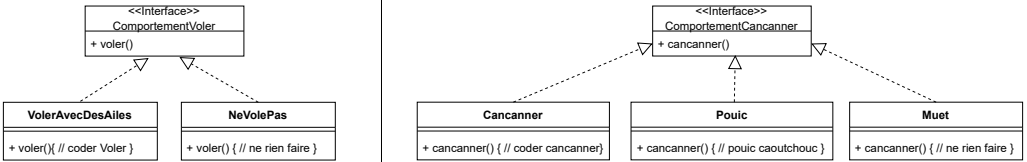
```
Animal animal = new Chien();
animal.cri(); // we can use polymorphism
```

- Encore mieux : assigner l'implantation concrète à l'exécution directement !

```
animal = getAnimal();
animal.cri(); // even better!
```

CanarApp2.0 : Coder les comportements des canards

- Deux interfaces : `ComportementVoler` & `ComportementCancanner`
  - avec les classes concrètes représentant chaque comportement réel



- D'autres types peuvent réutiliser les comportements "Voler" et "Cancanner"
  - ils ne sont plus cachés dans notre classe `Canard`
  - on peut ajouter autant de comportement que l'on veut sans modifier `Canard` + les autres classes de comportement

Quelques questions à cette étape

- Des classes qui représentent des comportements, un peu bizarre non ? Ne représentent pas d'état réel

Quelques questions à cette étape

- Des classes qui représentent des comportements, un peu bizarre non ? Ne représentent pas d'état réel
  - C'est vrai qu'elles ne représentent pas quelque chose de concret (pas d'attributs)
  - Mais un comportement pourrait avoir des attributs (par exemple la vitesse, les battements d'aile/min pour le vol)

Quelques questions à cette étape

- Des classes qui représentent des comportements, un peu bizarre non ? Ne représentent pas d'état réel
  - C'est vrai qu'elles ne représentent pas quelque chose de concret (pas d'attributs)
  - Mais un comportement pourrait avoir des attributs (par exemple la vitesse, les battements d'aile/min pour le vol)

- Canard devrait aussi devenir une interface ?

Quelques questions à cette étape

- Des classes qui représentent des comportements, un peu bizarre non ? Ne représentent pas d'état réel
  - C'est vrai qu'elles ne représentent pas quelque chose de concret (pas d'attributs)
  - Mais un comportement pourrait avoir des attributs (par exemple la vitesse, les battements d'aile/min pour le vol)

- Canard devrait aussi devenir une interface ?
  - Ici non (on va voir le code plus tard)
  - On veut bénéficier de l'héritage de méthodes et d'attributs avec la super-classe `Canard`

Quelques questions à cette étape

- Des classes qui représentent des comportements, un peu bizarre non ? Ne représentent pas d'état réel
  - C'est vrai qu'elles ne représentent pas quelque chose de concret (pas d'attributs)
  - Mais un comportement pourrait avoir des attributs (par exemple la vitesse, les battements d'aile/min pour le vol)

- Canard devrait aussi devenir une interface ?
  - Ici non (on va voir le code plus tard)
  - On veut bénéficier de l'héritage de méthodes et d'attributs avec la super-classe `Canard`

- Avec cette nouvelle architecture, que feriez vous pour ajouter un comportement "voler en jet-pack" ?

Quelques questions à cette étape

- Des classes qui représentent des comportements, un peu bizarre non ? Ne représentent pas d'état réel
  - C'est vrai qu'elles ne représentent pas quelque chose de concret (pas d'attributs)
  - Mais un comportement pourrait avoir des attributs (par exemple la vitesse, les battements d'aile/min pour le vol)

- Canard devrait aussi devenir une interface ?
  - Ici non (on va voir le code plus tard)
  - On veut bénéficier de l'héritage de méthodes et d'attributs avec la super-classe `Canard`

- Avec cette nouvelle architecture, que feriez vous pour ajouter un comportement "voler en jet-pack" ?
  - Créer une classe `VolerAvecJetPack` qui met en œuvre l'interface `ComportementVoler`

Quelques questions à cette étape

- Des classes qui représentent des comportements, un peu bizarre non ? Ne représentent pas d'état réel
  - C'est vrai qu'elles ne représentent pas quelque chose de concret (pas d'attributs)
  - Mais un comportement pourrait avoir des attributs (par exemple la vitesse, les battements d'aile/min pour le vol)

- Canard devrait aussi devenir une interface ?
  - Ici non (on va voir le code plus tard)
  - On veut bénéficier de l'héritage de méthodes et d'attributs avec la super-classe `Canard`

- Avec cette nouvelle architecture, que feriez vous pour ajouter un comportement "voler en jet-pack" ?
  - Créer une classe `VolerAvecJetPack` qui met en œuvre l'interface `ComportementVoler`

- Quelle autre classe pourrait avoir besoin d'utiliser le comportement "cancanner" mais qui n'est pas une espèce de canard ?

Quelques questions à cette étape

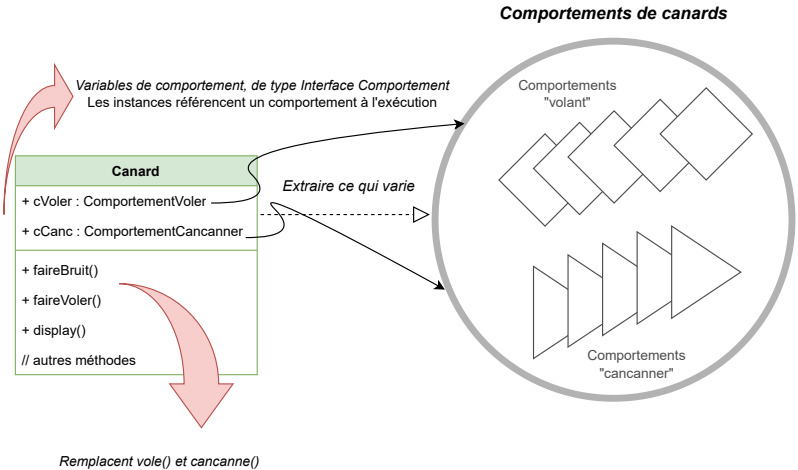
- Des classes qui représentent des comportements, un peu bizarre non ? Ne représentent pas d'état réel
  - C'est vrai qu'elles ne représentent pas quelque chose de concret (pas d'attributs)
  - Mais un comportement pourrait avoir des attributs (par exemple la vitesse, les battements d'aile/min pour le vol)

- Canard devrait aussi devenir une interface ?
  - Ici non (on va voir le code plus tard)
  - On veut bénéficier de l'héritage de méthodes et d'attributs avec la super-classe `Canard`

- Avec cette nouvelle architecture, que feriez vous pour ajouter un comportement "voler en jet-pack" ?
  - Créer une classe `VolerAvecJetPack` qui met en œuvre l'interface `ComportementVoler`

- Quelle autre classe pourrait avoir besoin d'utiliser le comportement "cancanner" mais qui n'est pas une espèce de canard ?
  - un appeau à canard

CanarApp2.0 : Intégrer les comportements à la classe Canard



- Utilisation du polymorphisme pour les variables de type Comportement
- On va voir maintenant comment coder `faireBruit()` et `faireVol()`

CanarApp2.0 : Déclaration de faireBruit()

```
public class Canard {
  ComportementCancanner cCanc;
  // etc

  public void faireBruit() {
    cCanc.bruit();
  }
}
```

- Chaque canard référence un objet qui met en œuvre l'interface ComportementCancanner
- Délégation par Canard du comportement à l'objet référencé
  - peu importe quel type réel d'objet, on veut simplement qu'il sache émettre un bruit !
- Même principe pour le comportement "voler"

CanarApp2.0 : Comment affecter les variables de comportement dans Canard

- Regardons la classe Colvert
  - hérite des variables de comportement de Canard
  - initialisation avec les comportements désirés

```
public class Colvert extends Canard {
  public Colvert() {
    cCanc = new Cancanner(); // inheritage
    vComp = new VolerAvecDesAiles(); // inheritage
  }

  public void afficher() {
    System.out.println("Je suis un vrai colvert!");
  }
}
```

Mais, on fait de la programmation par implantation au final ? ? ?

- Dans le constructeur
  - Créations d'instances concrètes de `ComportementCancanner`
  - Pareil pour `ComportementVoler`

- À cette étape oui, mais on peut changer ça facilement
  - avec le polymorphisme, on peut assigner différents comportements lors de l'exécution
  - on va voir ça un peu plus loin
  - vous pouvez déjà réfléchir à comment on pourrait faire ça !

CanarApp2.0 : Code de l'application (1/5) [Canard.java]

```
public abstract class Canard {
  ComportementCancanner cCanc;
  ComportementVoler cVoler;

  public Canard() {
  }
  public abstract void afficher();

  public void faireVoler() {
    cVoler.voler();
  }
  public void faireBruit() {
    cCanc.bruit();
  }

  public void nager() {
    System.out.println("Tous les canards nagent!!!");
  }
}
```

CanarApp2.0 : Code de l'application (2/5) (comportement "Voler")

```java
public interface ComportementVoler {
  public void voler();
}
```

```java
public class VolerAvecDesAiles implements ComportementVoler {
  public void voler() {
    System.out.println("Je vole!!");
  }
}
```

```java
public class NeVolePas implements ComportementVoler {
  public void voler() {
    System.out.println("Je ne peux pas voler...");
  }
}
```

CanarApp2.0 : Code de l'application (3/5) (comportement "cancanner")

```java
public interface ComportementCancanner {
  public void bruit();
}
```

```java
public class Cancanner implements ComportementCancanner {
  public void bruit() {
    System.out.println("Couin couin");
  }
}
```

```java
public class Muet implements ComportementCancanner {
  public void bruit() {
    System.out.println("..........");
  }
}
```

CanarApp2.0 : Code de l'application (4/5) [Colvert.java]

```
public class Colvert extends Canard {
        public Colvert() {
                cVoler = new VolerAvecDesAiles();
                cCanc = new Cancanner();
        }

        public void afficher() {
                System.out.println("Je suis un vrai colvert!");
        }
}
```

CanarApp2.0 : Code de l'application (5/5) [CanarApp.java]

```
public class CanarApp {
  public static void main(String[] args) {
    Canard c_colvert = new Colvert();
    c_volvert.afficher();
    c_colvert.faireBruit();
    c_colvert.faireVoler();
  }
}
```

```
% java CanarApp.java
Couin couin
Je vole!!!
```

Pour finir : définir le comportement dynamiquement (1/4)

- Ajouter deux nouvelles méthodes à la classe `Canard`
- Permettent de changer le comportement d'un canard "à la volée" !

```
public void setComportementVoler(ComportementVoler cv) {
  cVoler = cv;
}

public void setComportementCancanner(ComportementCancanner cc)
    {
  cCanc = cc;
}
```

Pour finir : définir le comportement dynamiquement (2/4)

- Définir un nouveau modèle de Canard [ModeleCanard.java]
  - Notre prototype ne peut pas voler de base mais peut cancanner

```
public class ModeleCanard extends Canard {

  public ModeleCanard() {
    cVoler = new NeVolePas();
    cCanc = new Cancanner();
  }

  public void afficher() {
    System.out.println("Je suis un prototype de canard :)");
  }

}
```

Pour finir : définir le comportement dynamiquement (3/4)

- Définir un nouveau comportement "voler" [JetPack.java]

```
public class JetPack implements ComportementVoler {
  public void voler() {
    System.out.println("Je vole en jet-pack, la classe!!");
  }
}
```

Pour finir : définir le comportement dynamiquement (4/4)

- Ajouter le nouveau modèle et comportement [CanarAppBis.java]

```
public class CanarAppBis {
  public static void main(String[] args) {
    Canard c_colvert = new Colvert();
    c_colvert.faireBruit();
    c_colvert.faireVoler();

    Canard modele = new ModeleCanard();
    modele.faireVoler();
    modele.setComportementVoler(new JetPack());
    modele.faireVoler();
  }
}
```

```
% java CanarApp.java
Couin couin
Je vole!!!
Je ne peux pas voler...
Je vole en jet-pack, la classe!!
```

Bilan : vue globale

- Des canards qui étendent la classe `Canard` avec
  - des comportements "voler" qui mettent en œuvre `ComportementVoler`
  - des comportements "cancanner" qui mettent en œuvre `ComportementCancanner`

- On peut voir ces comportements comme des algorithmes
  - ils représentent différentes actions que les canards peuvent faire (différentes manières de voler, chanter etc)
  - ils sont interchangeables !
  - on pourrait faire la même chose pour des ensembles de classes qui décrivent différentes manières de calculer une dérivée en fonction de la forme de la fonction

- Petite leçon : **Préférer la composition à l'héritage !**
  - chaque Canard a un comportement "voler" + "cancanner" = composition
  - au lieu de les hériter, les canards obtiennent leurs comportements par composition avec l'objet de comportement désiré
  - la composition est utilisée dans de nombreux patrons de conception !

En parlant de patrons de conception...

- FELICITATIONS ! ! Vous venez d'appliquer votre premier *design pattern* : *Strategy*
  - définit une famille d'algorithmes
  - encapsule chacune d'eux
  - les rend interchangeable

- *Strategy* laisse l'algorithme varier indépendamment des clients qui l'utilisent

Plan du cours

Motivation

- Concevoir un logiciel est difficile
  - Bien décomposer le problème
  - Créer de bonnes abstraction (structuration du code)
  - Flexibilité, extensibilité, modularité et élégance
- Aspect reutilisabilité du code (difficile $+++$)
- Pour cela, des conceptions existent !
  - avec des caractéristiques récurrentes (comme on l'a vu avec *Strategy*)
  - mais toutes différentes les unes des autres
- Objectifs
  - Disposer de briques de conception réutilisables
  - Pouvoir produire du code plus rapidement & de meilleur qualité

Patrons de conception : définition

Définition

Un *design pattern* décrit une solution à un problème général et récurrent de conception dans un contexte particulier.

- Les patrons de conception ne sont pas :
  - des classes ou bibliothèques (listes ou tables d'association)
  - des conceptions complètes et concrètes, ni une implantation
- Les patrons de conception sont :
  - des description abstraites de **solutions récurrentes** sur comment résoudre des problèmes **communs**
  - des composants logiques décrits indépendamment d'un langage donné (ici bien sûr on s'intéresse à leur version Java)
  - un moyen d'offrir un vocabulaire de conception commun (documentation & communication)

Patrons de conception : éléments structurels

- Nom
  - Concis et explicite
- Problème résolu
  - Quand utiliser ce patron ?
  - Définition du problème & du contexte
- Solution
  - présentée sous la forme d'un schéma (= diagramme UML)
  - liste des acteurs (classes & objets) et leurs relations
- Avantages et inconvénients
  - Impact sur la réutilisation, le réutilisabilité etc
  - Peuvent varier fortement en fonction des variations du patron

Catégories de *Patterns*

- **Création**
  - Description de la manière dont un objet ou un ensemble d'objets peuvent être créés, initialisés, et configurés
  - Isolation du code relatif à la création, à l'initialisation afin de rendre l'application indépendante de ces aspects
- **Structure**
  - Description de la manière dont doivent être connectés des objets de l'application afin de rendre ces connections indépendantes des évolutions futures de l'application
  - Découplage de l'interface et de l'implantation de classes et d'objets
- **Comportement**
  - Description de comportements d'interaction entre objets
  - Gestion des interactions dynamiques entre des classes et des objets

Les patrons de création

Catégorie	*Design Pattern*	Aspect(s) qui peuvent varier
Création	*Abstract Factory*	familles d'objets
	Factory	sous-classe d'un objet qui est instancié
	Singleton	l'unique instance de la classe
	Prototype	la classe de l'objet qui est instancié
	Builder	comment un objet composite est instancié

On va parler *Singleton* et *Factory*

Les patrons de structure

Catégorie	*Design Pattern*	Aspect(s) qui peuvent varier
Structure	*Adapter*	interface pour un objet
	Bridge	implantation d'un objet
	Facade	interface pour un sous-système
	Proxy	façon d'accéder à un objet, et sa localisation
	Decorator	responsabilités d'un objet sans sous-classes
	...	...

Lisez des références si vous êtes curieux à propos de l'un d'entre eux

Les patrons de comportement

Catégorie	*Design Pattern*	Aspect(s) qui peuvent varier
Comportement	*Iterator*	comment une aggrégation d'éléments est accédée & traversée
	Strategy	un algorithme
	Template Methode	une étape d'un algorithme
	Visitor	les opérations appliquées à un(des) objet(s) sans changer leur(s) classe(s)
	...	...

On a déjà vu *Strategy*

Plan du cours

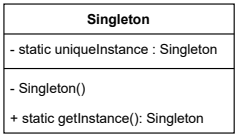
Le patron "singleton"

- Intention :
  - garantir qu'une classe ne peut avoir qu'une et une seule instance
  - fournir un point d'accès global à cette instance

- Motivations
  - Une variable globale est facilement accessible MAIS ça ne garantit pas l'unicité de l'instanciation de l'objet
  - Rendre la classe elle-même responsable de gérer son unique instance
  - Exemple : Client/serveur avec un serveur unique
  - Exemple : L'ENSIIE n'a qu'un seul directeur

- Application
  - Quand il ne doit y avoir qu'une seule instance d'une classe, et et elle doit être accessible aux "clients" à partir d'un point d'accès déterminé

Le patron "singleton" : Structure



- Une variable statique pour stocker notre unique instance
- Le constructeur est privé
  - seulement `Singleton` peut instancier cette classe !
- `getInstance()` permet d'instancier et retourner l'instance unique
- On peut ajouter évidemment d'autres méthodes et attributs

Plan du cours

Le patron "singleton" : implantation

```
public class Singleton {
  private static Singleton uniqueInstance;

  // other useful instance variables here

  private Singleton() {}

  public static Singleton getInstance() {
    if (uniqueInstance == null) {
      uniqueInstance = new Singleton(); /* lazy instantiation
          */
    }
  return uniqueInstance;
  }

  // other useful methods here
}
```

Exemple : contrôleur pour une chaudière à chocolat

```
public class ChocolateBoiler {
  private boolean empty;
  private boolean boiled;

  private ChocolateBoiler() {
    empty = true;
    boiled = false; // when the boiler is empty
  }

  public void fill() {
    if (isEmpty()) {  /* Must be empty and, once full, set
       flags */
      empty = false;
      boiled = false;
      // fill the boiler with a milk/chocolate mixture.
    }
  }

  public void drain() {
    if (!isEmpty() && isBoiled()) {
      // drain the boiled milk and chocolate
      empty = true; /* set back empty when drained */
    }
  }

  public void boil() {
    if (!isEmpty() && !isBoiled()) { /* must be full and not
       already boiled */
      // bring the contents to a boil
      boiled = true;
    }
  }

  public boolean isEmpty() {
    return empty;
  }

  public boolean isBoiled() {
    return boiled;
  }
}
```

Des soucis, encore des soucis...

- On peut suspecter que si on déclare 2 instances du contrôleur `ChocolatBoiler` d'une même machine, ça peut mal se passer
  - pourquoi ?

Des soucis, encore des soucis...

- On peut suspecter que si on déclare 2 instances du contrôleur `ChocolatBoiler` d'une même machine, ça peut mal se passer
  - pourquoi ?
  - Si elles se détachent en terme de comportement
  - L'un peut appeler `fill()` alors que l'autre est en train d'appeler de chauffer avec `boil()`

Maintenant, on va transformer la classe en Singleton

Exemple : singleton pour les chaudières à chocolat

```
public class ChocolateBoiler {
  private boolean empty;
  private boolean boiled;
  private static ChocolateBoiler uniqueInstance; /* unique
     instance */

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

  public static ChocolateBoiler getInstance() { /* init and
     access to unique instance */
    if (uniqueInstance == null) {
      uniqueInstance = new ChocolateBoiler();
    }
  return uniqueInstance;
  }

  public void fill() {
    if (isEmpty()) {
      empty = false;
      boiled = false;
      // fill the boiler with a milk/chocolate mixture
    }
  }

  // rest of ChocolateBoiler code...
}
```

C'est bon, on est sauvés ! (vraiment ?)

- Seule unique instance possible pour une chaudière
  - donc on peut plus avoir le problème d'avant
- Maintenant, pour optimiser les performances, l'entreprise alloue 2 *threads* au contrôleur
  - on va aller deux fois plus vite !
- **HOUSTON, WE HAVE A PROBLEM**
  - la méthode `fill()` a pu remplir la chaudière pendant qu'un jeu de chocolat et lait était en train de chauffer ! !
  - résultat : 1000L de mélange perdu :(
- Quelle est cette diablerie ?
  - on a une unique instance de la chaudière
  - tous les appels à `getInstance()` doivent renvoyer la même instance ?

C'est bon, on est sauvés ! (vraiment ?)

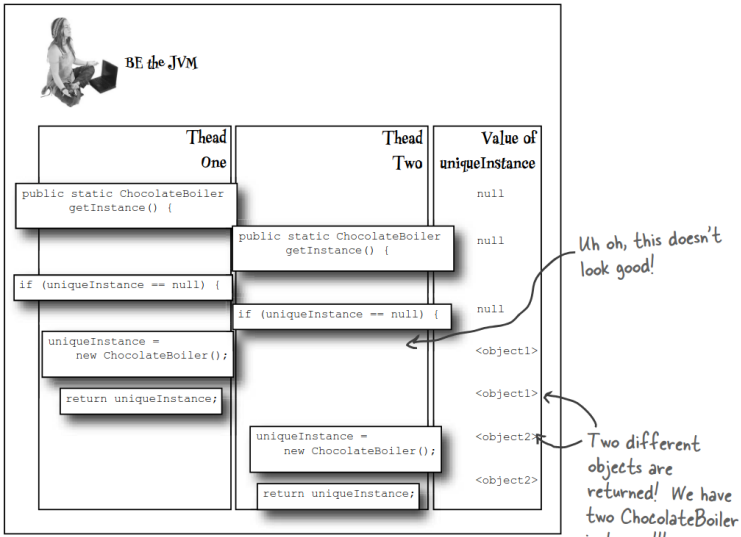
- Seule unique instance possible pour une chaudière
  - donc on peut plus avoir le problème d'avant
- Maintenant, pour optimiser les performances, l'entreprise alloue 2 *threads* au contrôleur
  - on va aller deux fois plus vite !
- **HOUSTON, WE HAVE A PROBLEM**
  - la méthode `fill()` a pu remplir la chaudière pendant qu'un jeu de chocolat et lait était en train de chauffer ! !
  - résultat : 1000L de mélange perdu :(
- Quelle est cette diablerie ?
  - on a une unique instance de la chaudière
  - tous les appels à `getInstance()` doivent renvoyer la même instance ?

Voyons voir ce qu'il se passe au niveau de la JVM (1/2)

```
ChocolateBoiler boiler = ChocolateBoiler.getInstance();
fill();
boil();
drain();
```

- 2 *threads* qui exécutent ce code
- Déroulons sur ce code ce qu'il se passe dans la JVM

Voyons voir ce qu'il se passe au niveau de la JVM (2/2)



Source : Head First Design Patterns

Plan du cours

Comment gérer le *multithreading* ?

- Solution triviale : faire de `getInstance()` une méthode synchronized
  - chaque thread attend son tour avant d'accéder à la méthode
  - **très couteux** :(
  - **synchronisation seulement utile au premier appel à la méthode (passer de null à une instance)** :(

```
public class Singleton {
  private static Singleton uniqueInstance;
  // other useful instance variables here

  private Singleton() {}

  public static synchronized Singleton getInstance() {
    if (uniqueInstance == null) {
      uniqueInstance = new Singleton();
    }
    return uniqueInstance;
  }
  // other useful methods here
}
```

Un meilleur *multithreading* : 1) Ne rien faire

- Si la synchronisation n'est pas un problème dans notre logiciel
  - on peut laisser comme ça (avec synchronized)
  - à savoir : peut réduire les performance d'un facteur 100

- S'il y a énormément d'appels à cette méthode, ne rien faire n'est peut-être pas la meilleure solution...

Un meilleur *multithreading* : 2) Ne plus utiliser de *lazy instantiation*

- Si le logiciel crée et utilise toujours une instance du singleton, ou que le coût de la création d'une instance est bas
  - pas à une instanciation directe
  - JVM se charge de la création au chargement de la classe **AVANT** qu'un *thread* puisse y accéder

---

```
public class Singleton {
  private static Singleton uniqueInstance = new Singleton();

  private Singleton() {}

  public static Singleton getInstance() {
    return uniqueInstance;
  }
}
```

Un meilleur *multithreading* : 3) Utiliser du *"double-check looking"*

- Principe du *"double-check looking"* [pas dispo avant Java 2, version 5]
  1. vérifier si une instance est déjà créée
  2. si ce n'est pas le cas, alors on synchronise avant de la créer (donc une seule fois)
- Utilisation du mot-clé volatile, meilleur choix pour implantation *thread-safe*

```
public class Singleton {
  private volatile static Singleton uniqueInstance;
  /* volatile = ensures threads handle uniqueInstance
     correctly when it is being initialized */

  private Singleton() {}

  public static Singleton getInstance() {
    if (uniqueInstance == null) {
      synchronized (Singleton.class) {
        if (uniqueInstance == null) {
          uniqueInstance = new Singleton();
        }
      }
    }
    return uniqueInstance;
  }
}
```

Plan du cours

Le patron "Factory"

- Intention :
  - définir une interface pour créer un objet mais...
  - déléguer aux sous-classes les décisions d'instanciation

- Quand l'utiliser ?
  - quand une classe ne peut pas anticiper la classe des objets à créer
  - quand une classe veut que sa sous-classe spécifie les objets qu'elle crée

Plan du cours

Partons cette fois sur un exemple culinaire : la pizzeria !

```
public class PizzaStore {
  Pizza orderPizza() {
    Pizza pizza = new Pizza();
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
  }
}
```

- Mais on a plus d'un type de pizzas...
  - solution : une nouvelle `orderPizza(String type)` pour déterminer le bon type de pizza à préparer

La pizzeria avec plusieurs pizzas c'est mieux !

```
public class PizzaStore {
  Pizza orderPizza(String type) {
    Pizza pizza;
    if (type.equals("cheese")) {
      pizza = new CheesePizza();
    } else if (type.equals("greek") {
      pizza = new GreekPizza();
    } else if (type.equals("pepperoni") {
      pizza = new PepperoniPizza();
    }
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
  }
}
```

Mais à chaque fois qu'on veut changer la carte, Bérézina !

```
public class PizzaStore {
  Pizza orderPizza(String type) {
    Pizza pizza;

    if (type.equals("cheese")) {
      pizza = new CheesePizza();
    } else if (type.equals("greek") {
      pizza = new GreekPizza();
    } else if (type.equals("pepperoni") {
      pizza = new PepperoniPizza();
    } else if (type.equals("clam") { /* new pizzas!! */
      pizza = new ClamPizza();
    } else if (type.equals("veggie") {
      pizza = new VeggiePizza();
    }

    /* below does not change */
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
  }
}
```

On en revient à l'encapsulation !

- Problème : le code n'est pas fermé à la modification
  - pour changer le catalogue, il faut tout reparcourir...
  - du code qui varie + du code qui change pas = ça va pas !

- Mais on sait ce qui varie et ne varie pas
  - il est temps de refaire de l'encapsulation !

- Comment va-t-on faire ?
  - Extraire le code de création d'objet en dehors de `orderPizza()`
  - On place ce code dans un objet qui va seulement se charger de créer les pizzas (une *Factory* !)
  - `orderPizza()` devient alors un client de cette *Factory* !

Une *Factory* à pizza !

```
public class SimplePizzaFactory { /* notre usine */
  public Pizza createPizza(String type) { /* méthode appelée
     par tous les "clients" */
    Pizza pizza = null;

    if (type.equals("cheese")) {
      pizza = new CheesePizza();
    } else if (type.equals("pepperoni")) {
      pizza = new PepperoniPizza();
    } else if (type.equals("clam")) {
      pizza = new ClamPizza();
    } else if (type.equals("veggie")) {
      pizza = new VeggiePizza();
    }
    return pizza;
  }
}
```

Quels avantages à celà ? On a juste repoussé le problème...

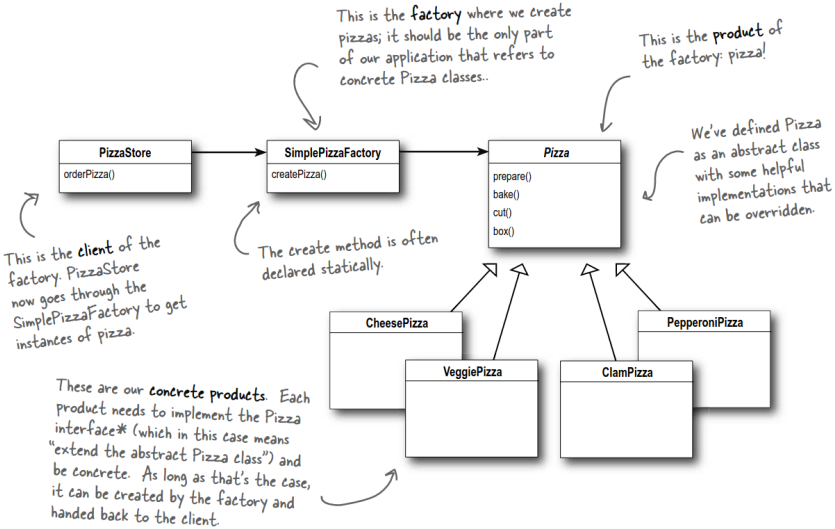
- `SimplePizzaFactory` pourrait avoir plusieurs "clients"
  - un menu, une classe livraison etc
  - grâce à l'encapsulation, on a juste à modifier le code à un endroit !
  - retirer les instanciations du code des clients (= `PizzaStore`)

- Justement, passons maintenant au code des "clients" !

Retravailler la classe PizzaStore

```
public class PizzaStore {
  SimplePizzaFactory factory;

  public PizzaStore(SimplePizzaFactory factory) {
    this.factory = factory;
  }

  public Pizza orderPizza(String type) {
    Pizza pizza;
    pizza = factory.createPizza(type); /* new operator
        replaced by create method! */
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
  }

  // other methods here
}
```

Nous venons de définir une Simple Factory (idiome)



Source : Head First Design Patterns

Plan du cours

Ouvrons des franchises !

- Gros succès : on veut se développer dans d'autres villes !
  - on veut réutiliser le code précédent
  - mais comment gérer les différences régionales ?
    → chaque franchise a sa propre version des pizzas à la carte

- Idée : composer `PizzaStore` avec la bonne *factory* et une franchise

```
NYPizzaFactory nyFactory = new NYPizzaFactory(); /* NY style
   pizzas */
PizzaStore nyStore = new PizzaStore(nyFactory); /* PizzaStore
   referenced to nyFactory */
nyStore.order("Veggie");

ChicagoPizzaFactory chicagoFactory = new ChicagoPizzaFactory();
PizzaStore chicagoStore = new PizzaStore(chicagoFactory);
chicagoStore.order("Veggie")
```

Une contrainte en plus : garder un certain contrôle sur les pizzas

- On souhaiterait contraindre les franchises (utilisation du protocole de `orderPizza`))
  - par exemple : garder le même *packaging*, imposer un type de four etc
  - solution : **créer un cadre qui contraint les magasins et la création de pizza ensemble** (tout en restant flexible bien sûr !)

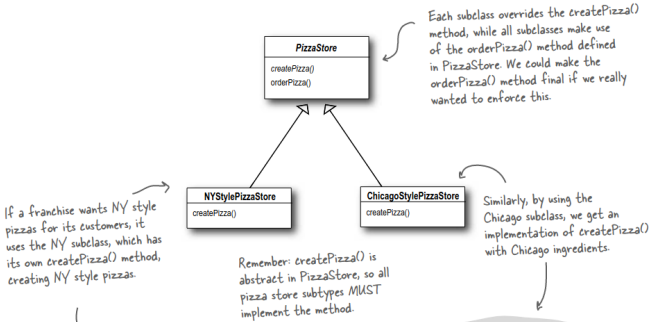
- Avec `SimplePizzaFactory`, le processus de fabrication des pizzas était très lié à `PizzaStore` mais ça n'était pas flexible :(
  - Trouvons un moyen de faire mieux !

Un cadre pour le PizzaStore (1/3)

```
public abstract class PizzaStore {
  public Pizza orderPizza(String type) {
    Pizza pizza;

    pizza = createPizza(type); /* back in PizzaStore */

    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
  }

  abstract Pizza createPizza(String type); /* Factory object
     in this method */
}
```

- Localiser le processus de création dans `PizzaStore` + donner liberté aux franchises
- `createPizza()` en **méthode abstraite** dans `PizzaStore` (sous-classe pour chaque franchise régionale qui décide de la recette des pizzas)
- Rappel : on veut que toutes les franchises utilisent la procédure dans `orderPizza`

Un cadre pour le PizzaStore : déléguer aux sous-classes (2/3)



Source : Head First Design Patterns

Un cadre pour le PizzaStore : implantation (3/3)

```
public class NYPizzaStore extends PizzaStore {
  Pizza createPizza(String item) {
    if (item.equals("cheese")) {
      return new NYStyleCheesePizza();
    } else if (item.equals("veggie")) {
      return new NYStyleVeggiePizza();
    } else if (item.equals("clam")) {
      return new NYStyleClamPizza();
    } else if (item.equals("pepperoni")) {
      return new NYStylePepperoniPizza();
    } else return null;
  }
}
```

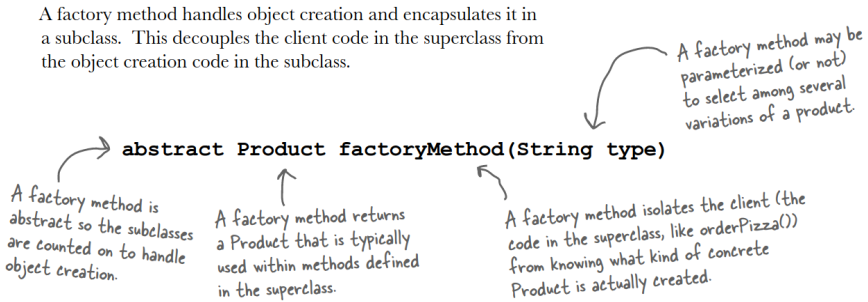
- `createPizza()` crée une pizza (implantation obligatoire de la méthode)
- On crée ici nos classes concrètes de pizzas : pour chaque type de pizza, on crée la version NY-style
- Notez que `orderPizza()` dans la super-classe ne sait pas quelle pizza sera créée ; seulement qu'elle peut la préparer, cuire, couper et mettre en boîte !

Déclarer une méthode *Factory* (1/2)

```
public abstract class PizzaStore {
  public Pizza orderPizza(String type) {
    Pizza pizza;

    pizza = createPizza(type); /* HERE */

    pizza.prepare();
    // etc
    return pizza;
  }

  protected abstract Pizza createPizza(String type); /* HERE */
}
```

- Toute la responsabilité pour instancier des pizzas a été délégué à UNE SEULE méthode qui agit comme une Factory

Déclarer une méthode *Factory* (2/2)



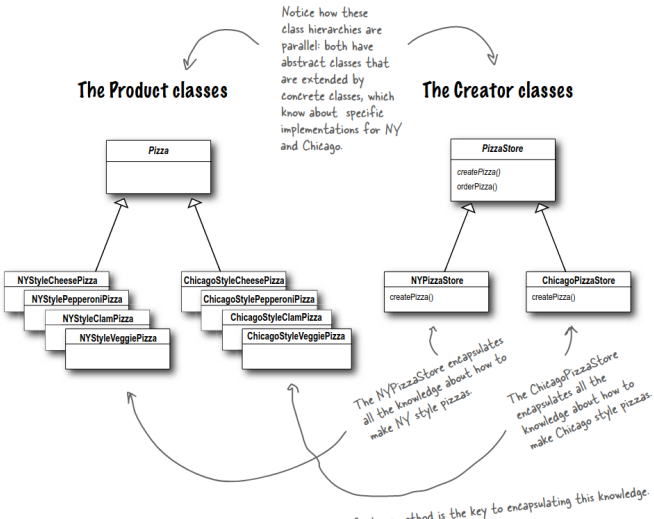
Source : Head First Design Patterns

- Une méthode *Factory* crée des objets et les encapsule dans une sous-classe
- Découple le code client dans la super-classe du code de création d'objet dans la sous-classe

On passe aux tests !

- Neo veut commander une *cheese* pizza à partir d'un `PizzaStore` de New-York
- Trinity veut quand à elle une *cheese* pizza de Chicago. Même processus de commande, mais une pizza différente !
- Comment faire ça ?
  1. Chacun à besoin de son instance "locale" de `PizzaStore`
  2. Chacun appelle ensuite `orderPizza` et indique quel type de pizza il veut (veggie etc)
  3. Appel à `createPizza` définie dans `NYPizzaStore` (resp. `ChicagoPizzaStore`)
     - `NYPizzaStore` instancie des classes "NY style Pizza" (idem pour `ChicagoPizzaStore`)
  4. `orderPizza()` ignore quel type de pizza a été fabriqué, mais elle sait que c'est une pizza qui a été préparée, cuite etc

Voici donc notre *Factory pattern* avec les pizzas



Source : Head First Design Patterns

Il nous manque une chose essentielle, les pizzas ! !

```
public abstract class Pizza { /* abstract class */
  String name;
  String dough;
  String sauce; /* pizza's attributes */
  ArrayList toppings = new ArrayList();

  void prepare() { /* procedure for all pizzas */
    System.out.println("Preparing " + name);
    System.out.println("Tossing dough...");
    System.out.println("Adding sauce...");
    System.out.println("Adding toppings: ");

    for (int i = 0; i < toppings.size(); i++) {
      System.out.println(" " + toppings.get(i));
    }
  }

  void bake() {
    System.out.println("Bake for 25 minutes at 350");
  }

  void cut() {
    System.out.println("Cutting the pizza into diagonal
       slices");
  }

  void box() {
    System.out.println("Place pizza in official PizzaStore
       box");
  }

  public String getName() {
    return name;
  }
}
```

Et maintenant... les sous classes concrètes !

```
public class NYStyleCheesePizza extends Pizza {
  public NYStyleCheesePizza() {
    name = "NY Style Sauce and Cheese Pizza"; /* special NY
       cheese pizza */
    dough = "Thin Crust Dough";
    sauce = "Marinara Sauce";

    toppings.add("Grated Reggiano Cheese");
  }
}
```

```
public class ChicagoStyleCheesePizza extends Pizza {
  public ChicagoStyleCheesePizza() {
    name = "Chicago Style Deep Dish Cheese Pizza"; /* special
       Chicago cheese pizza */
    dough = "Extra Thick Crust Dough";
    sauce = "Plum Tomato Sauce";

    toppings.add("Shredded Mozzarella Cheese");
  }

  /* override the cut method */
  void cut() {
    System.out.println("Cutting the pizza into square slices");
  }
}
```

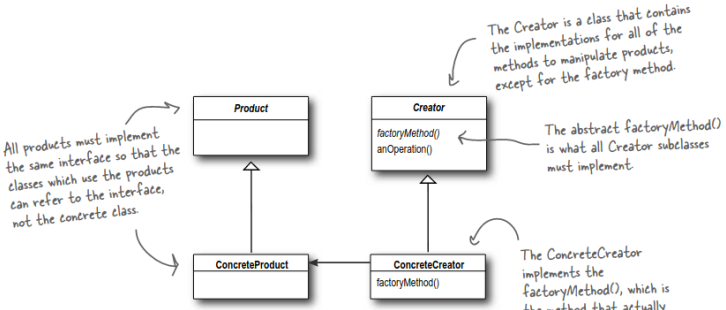
Ca y est, on y est : les pizzas ! ! !

```
public class PizzaTestDrive {
  public static void main(String[] args) {
    PizzaStore nyStore = new NYPizzaStore();
    PizzaStore chicagoStore = new ChicagoPizzaStore();

    Pizza pizza = nyStore.orderPizza("cheese");
    System.out.println("Neo ordered a " + pizza.getName() +
        "\n");

    pizza = chicagoStore.orderPizza("cheese");
    System.out.println("Trinity ordered a " + pizza.getName()
        + "\n");
  }
}
```

Pour conclure sur *Factory*

Définition

Le *Factory pattern* définit une interface pour créer des objets, mais laisse les sous-classes décider quelle classe instancier. La *Factory method* permet à une classe de déférer les instanciations aux sous-classes.



Source : Head First Design Patterns

Conclusion

- Patrons de conception : bonne manière de résoudre un problème particulier
- Nous en avons vu que 3, il y en a évidemment bien plus !
- Pensez-y quand vous concevrez un code, logiciel ou autre : il y a sûrement des patrons qui seront bénéfiques à votre code ! !
- Enormément de littérature à ce sujet !

Sources

- Le livre "Design Patterns" chez **O'Reilly - Head First** [PDF]
- Le livre "Design Patterns, Elements of Reusable Object-Oriented Software" chez **Addison-Weesley Professional Computing Series**