Sous-typer avec classe

Yann **Régis-Gianas**
yrg@pps.irif.fr

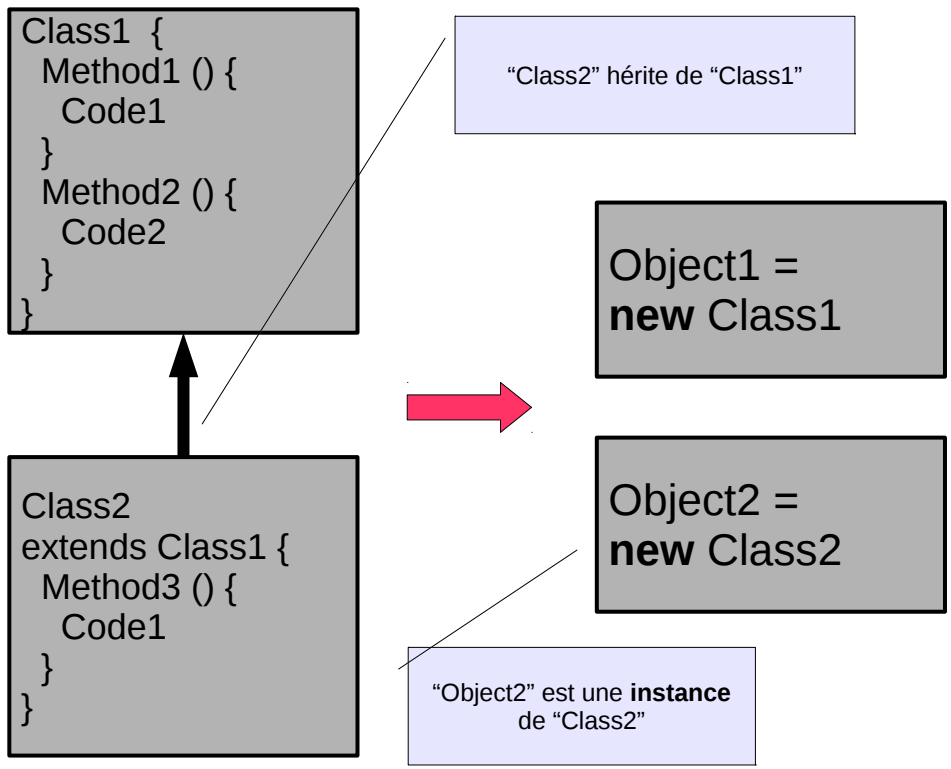
PPS - Université Denis Diderot – Paris 7

Sous-typer avec classe

Construire des classes par héritage

- Sous-typage
- Classes abstraites
- La classe abstraite comme une interface
- Problème de la classe de base fragile
- Problème de l'extensibilité fonctionnelle
- Problème de l'héritage multiple

Construction de classe par héritage



Mécanisme d'héritage

Héritage

L'héritage est un mécanisme de **réutilisation de code** visant à construire une classe, c'est-à-dire un générateur d'objets, à partir d'une autre en **l'étendant** ou en **redéfinissant** certaines parties de ses membres.

Sous-classe et Super-classe

Si une classe B hérite d'une classe A, alors B est une sous-classe de A et A est une super-classe ou classe mère de B.

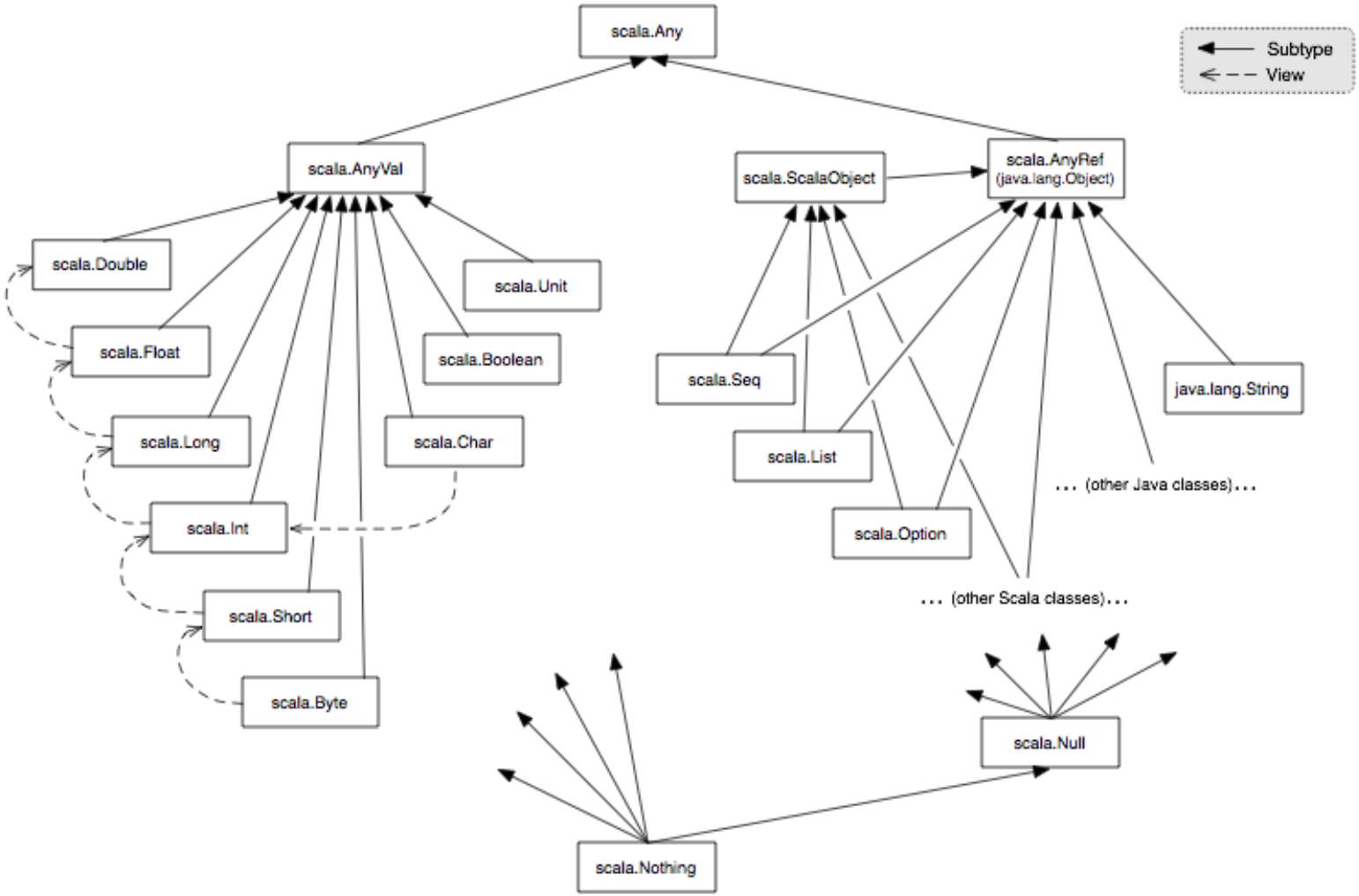
Hiérarchie de classes

La relation d'héritage forme un arbre appelé hiérarchie de classes.

L'héritage en Scala

```
01 class A {
02    def m1 = 0
03 }
04
05 class B extends A {
06    def m2 = 1
07 }
08
09 val ib = new B
10 val x = ib.m2
11 val y = ib.m1
```

La hiérarchie des types prédéfinis de Scala



Héritage et constructeur par défaut

```
01 class A (x : Int) {
02   def m1 = x
03 }
04
05 class B (x : Int, y : Int) extends A (x + y) {
06   def m2 = x
07 }
08
09 val ib = new B (1, 2)
10 val x = ib.m2
11 val y = ib.m1
```

Quelles sont les valeurs des variables `x` et `y` ?

Héritage et constructeurs auxiliaires

```
01 class A (x : Int) {
02   def m1 = x
03   def this (x : Int, y : Int) = this (x + y)
04 }
05
06 class B (x : Int, y : Int) extends A (x, y) {
07   def m2 = x
08   def this () = this (0, 0)
09 }
10
11 val ib = new B ()
12 val x = ib.m2
13 val y = ib.m1
```

Quelles sont les valeurs des variables `x` et `y` ?

Interdire l'héritage

Pour des raisons de sécurité ou pour améliorer l'efficacité de la compilation, on peut interdire l'introduction de nouvelles sous-classes à une hiérarchie. Pour cela, il suffit de préfixer la définition d'une classe `A` par le mot-clé `sealed`. Dès lors, il ne sera plus possible d'hériter de `A` sauf à l'intérieur du fichier source dans lequel cette classe `A` est définie.

Quand une classe est marquée `sealed`, le compilateur peut détecter les analyses par motifs sur des valeurs de cette classe qui ne sont pas **exhaustives**. Par exemple :

```
sealed class Color
case class Red () extends Color
case class Blue () extends Color
case class Green () extends Color

scala> def f (x : Color) =
  x match {
    case Red () => "red"
    case Blue () => "blue"
  }
<console>:15: warning: match is not exhaustive!
missing combination Color
missing combination Green
```

Le raisonnement par cas

La décomposition des différents cas d'une situation est essentielle dans la conception d'un système ou d'un algorithme. Grâce aux classes de cas scellées et aux analyses par cas, le compilateur de ScalaAvérifie pour vous que vous avez bien traités tous les cas.

Redéfinition de définitions en SCALA

```
01 class A {
02    def m : Int = 0
03 }
04
05 class B extends A {
06    override def m : Int = 1
07 }
08
09 val bi = new B
10 val x = bi.m
```

Redéfinition de définitions en Scala

```
01 class A {
02   def m : Int = 2 + 2
03 }
04
05 class B extends A {
06   override val m : Int = 2 + 2
07 }
08
09 val bi = new B
10 val x = bi.m
```

À quoi sert cette forme de redéfinition ?

Redéfinition de valeurs en SCALA

```
01 class A {
02    def m : Int = 0
03 }
04
05 class B extends A {
06    override def m : Int = 1
07 }
08
09 val bi = new B
10 val x = bi.m
```

Redéfinition de valeurs en SCALA

```
01 class A {
02   val m : Int = 0
03 }
04
05 class B extends A {
06   override def m : Int = 1
07 }
08
09 // error: overriding value m in class A of type Int;
10 // method m needs to be a stable, immutable value
```

D'après vous, pourquoi une telle limitation ?

Redéfinition des variables

```
01 class A {
02    var x = 0
03 }
04 class B extends A {
05    override var x = 1
06 }
07 class C extends A {
08    override val x = 1
09 }
10 class D extends A {
11    override def x = 1
12 }
13
14 // error: overriding variable x in class A of type Int;
15 // variable x cannot override a mutable variable
```

Récursion ouverte

Lorsque l'on écrit une classe objet, il faut s'imaginer que tous les appels de méthodes peuvent faire appel à des redéfinitions de ces méthodes dans les sous-classes de la classe en cours de définition. On dit que l'on travaille sous l'hypothèse d'un monde ouvert.

En d'autres termes, on peut voir l'ensemble des définitions des membres d'une classe comme un ensemble de **définitions mutuellement récursives** sur lesquelles on se donne la liberté de revenir dans le futur en les redéfinissant dans une classe fille. Cette propriété (méta-linguistisque) des objets s'appelle la **récursion ouverte**.

Par quel mécanisme la récursion ouverte est-elle implémentée ?

Récursion ouverte

Lorsque l'on écrit une classe objet, il faut s'imaginer que tous les appels de méthodes peuvent faire appel à des redéfinitions de ces méthodes dans les sous-classes de la classe en cours de définition. On dit que l'on travaille sous l'hypothèse d'un monde ouvert.

En d'autres termes, on peut voir l'ensemble des définitions des membres d'une classe comme un ensemble de **définitions mutuellement récursives** sur lesquelles on se donne la liberté de revenir dans le futur en les redéfinissant dans une classe fille. Cette propriété (méta-linguistisque) des objets s'appelle la **récursion ouverte**.

Par quel mécanisme la récursion ouverte est-elle implémentée ? Grâce au fait que tout appel de méthode s'effectue à travers `this`. Comme `this` ne correspond pas à une instance de la classe en cours de définition mais à une **instance d'une de ses sous-classes**, une version redéfinie de la méthode peut être exécutée.

Fermer la récursion

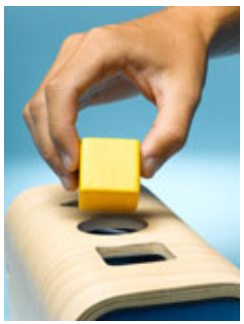
Il est possible d'interdire la redéfinition d'un membre à l'aide du mot-clé `final`. Ce mécanisme a deux intérêts :

- Il simplifie le raisonnement sur la classe : on connaît exactement le code executé par un appel à une méthode marquée `final` puisque c'est nécessairement celui de la classe en cours de définition.
- Le compilateur peut optimiser un appel à une méthode `final`.

Plan de la partie *"Construire des objets"*

Sous-typage induit

Quand on définit une relation d'héritage entre deux classes `A` et `B`, une relation de sous-typage entre les types (nommés) `A` et `B` est déclarée dans le système.



Sous-typage

Un type B est un sous-type d'un type A si dans tout contexte où une valeur de type A est attendue, une valeur de type B peut être utilisée.

Nous avons vu dans précédemment que SCALA propose du typage nominal et du typage structurel. La relation d'héritage déclarée entre deux classes augmente seulement la relation de sous-typage **nominale**.

Nous noterons

$$T \prec: U$$

pour la propriété « le type T est un sous-type de U ».

Covariance

Propriété

Si $A \prec: A'$ et $B \prec: B'$ alors $A \times B \prec: A' \times B'$.

Le type des paires formées de deux types A et B qui sont des sous-types de A' et B' est un sous-type des paires formées des deux types A' et B'. On dit que l'opérateur de type de formation des paires est covariant.

On peut clairement généraliser cette propriété aux n-uplets. De même, les enregistrements [1], qui sont des n-uplets nommés, vérifient aussi cette propriété.

1. Les enregistrements sont aussi appelés "structures" en C.

Le type des fonctions

Soient A, B, A', B'. À quelles conditions a-t-on $A \to B \prec: A' \to B'$?

```
01 class A { var x = 0 }
02 class B extends A { var y = 1 }
03 class C extends B { var z = 2 }
04
05 def g (f : B => B) = { val b = f (new B) ; b.y = 1 ; b }
06
07 def idA : A => A = (x : A) => x
08 def idB : B => B = (x : B) => x
09 def idC : C => C = (x : C) => x
```

Le type des fonctions

Soient A, B, A', B'. À quelles conditions a-t-on $A \to B \prec: A' \to B'$?

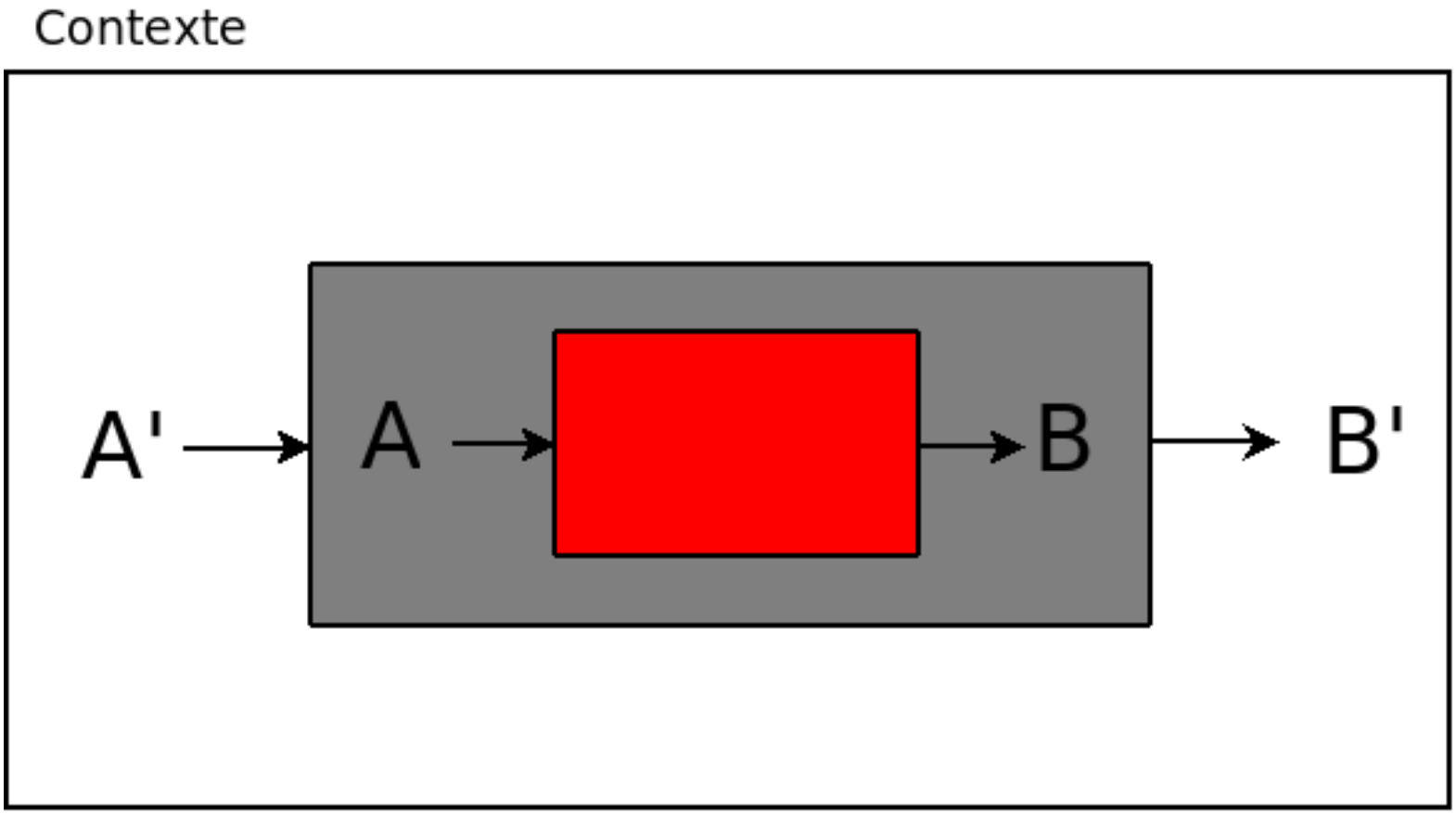
```
01 class A { var x = 0 }
02 class B extends A { var y = 1 }
03 class C extends B { var z = 2 }
04
05 def g (f : B => B) = { val b = f (new B); b.y = 1; b }
06
07 def idA : A => A = (x : A) => x
08 def idB : B => B = (x : B) => x
09 def idC : C => C = (x : C) => x
```

```
01 val b = g (idB) // OK
02 val a = g (idA) // Error
03 val c = g (idC) // Error
04
05 def fAC (a : A) : C = { val c = new C; c.x = a.x; c }
06 val c = g (fAC) // OK
```

Quel est le type de c ?

Le type des fonctions

Le type des fonctions



Contravariance

Propriété

Si $A' \prec: A$ et $B \prec: B'$ alors $A \to B \prec: A' \to B'$.

On dit que le constructeur de type des fonctions est contravariant sur son domaine. Il est covariant sur son codomaine.

Le type des références

- Soient A et B tels que $A \prec: B$.
- Le type des références sur A est muni de deux opérations :
  - $\mathrm{write} : \mathrm{ref}\, A \to A \to \mathrm{unit}$
  - $\mathrm{read} : \mathrm{ref}\, A \to A$

L'opérateur de type des références est-il covariant ou contravariant ?

Invariance

- Le type A devant être utilisé en position covariante et en position contravariante, l'opérateur de type ref est donc invariant.

Qu'en déduire sur les objets ?

Structurellement, un objet obtenu par instanciation d'une classe peut être vu comme un enregistrement (une structure en C) dont les champs correspondent aux membres de la classe.

Ainsi, d'un point de vue structurel, le type structurel de la classe `A` suivante est un sous-type du type de la classe `B` tandis que le type de la classe `C` n'est pas en relation de sous-typage avec celui de `B`. Pourquoi ?

```
class A { def m : Any => Int = (m : Any) => ... }
class B { def m : Int => Int = (m : Int) => ... }
class C { def m : Int => Any = (m : Int) => ... }
```

Qu'en déduire sur les objets ?

Règle de bon typage des redéfinitions

Quand on redéfinit un membre de type T d'une classe A dans une sous-classe B, cette nouvelle définition doit avoir un type U qui est un sous-type de T.

Corollaire

Cas particulier important : Quand le membre redéfini est une méthode, cela signifie que les types des arguments de la méthode dans B doivent être des **sur-types** des types des arguments de la méthode dans A.

Héritages valides

Pour être valide, la définition d'une sous-classe doit contenir des redéfinitions vérifiant la règle de bonne formation énoncée dans le transparent précédent ?

Qu'en est-il en JAVA, en C++, etc ?

Qu'en est-il en JAVA ?

En JAVA, la règle est plus stricte : une redéfinition ne doit pas changer le type des arguments des méthodes ! Ces types sont invariants dans toutes les redéfinitions tout au long de hiérarchie d'objets.

```
public class A {
  void m (Integer x) {}
}

public class B extends A {
   void m (Integer x) { /* Ici la redéfinition de m. */ }
}
```

Mais pourtant. . .

Vous avez sûrement déjà écrit un programme JAVA de la forme suivante :

```
public class A {
  int equal (A x) { return 0; }
}

public class B extends A {
   int equal (B x) { return 0; }
}
```

Plus précisément...

```
public class OneInteger {
      public int x;
      public boolean equal (OneInteger that) { return (that.x == this.x); }
}

public class TwoIntegers extends OneInteger {
      public int y;
      public boolean equal (TwoIntegers that) {
         return (that.y == this.y && super.equal (that));
      }
}

static void test_them (OneInteger o1, OneInteger o2) {
      System.out.println (o1.equal (o2));
}

public void test () {
      TwoIntegers t1 = new TwoIntegers ();
      TwoIntegers t2 = new TwoIntegers ();
      t1.x = 0; t1.y = 1; t2.x = 0; t2.y = 2;
      test_them (t1, t2);
}
```

Qu'affiche ce programme ?

Surcharge et redéfinition

Le programme précédent affiche « true ». En effet, la méthode « equal » de la classe B **n'est pas une redéfinition** de la méthode « equal » de la classe A !

Il s'agit d'une méthode totalement différente – parce qu'elle attend des arguments de type différents – qui « par hasard » [2] a le même nom que la méthode de B.

Ainsi, même si les définitions de classe précédentes semblent parfaitement naturelles, l'exécution de ce programme ne correspondent pas **du tout** à ce que l'on attend !

2. En fait, parce que c'est plus « pratique » comme ça.

Surcharge et redéfinition

Le programme précédent affiche « true ». En effet, la méthode « equal » de la classe B **n'est pas une redéfinition** de la méthode « equal » de la classe A !

Il s'agit d'une méthode totalement différente – parce qu'elle attend des arguments de type différents – qui « par hasard » [2] a le même nom que la méthode de B.

Ainsi, même si les définitions de classe précédentes semblent parfaitement naturelles, l'exécution de ce programme ne correspondent pas **du tout** à ce que l'on attend !

Il ne faut donc **jamais** écrire ce type de programme !

2. En fait, parce que c'est plus « pratique » comme ça.

Mais, peut-être que . . .

Peut-être que la définition du sous-typage est incorrecte. . . Essayons de voir ce qui se passerait dans un hypothétique langage qui autorisait la méthode « equal » de la sous-classe B à être comprise comme une redéfinition de celle de la classe A.

```
class A =
  attribute x : int
  message equal (that : A) =
    (x = that.x)
end

class B =
  inherits A
  attribute y : int
  message equal (that : B) =
     (x = that.x ∧ y = that.y)
end
```

Mais, peut-être que . . .

Peut-être que la définition du sous-typage est incorrecte. . . Essayons de voir ce qui se passerait dans un hypothétique langage qui autorisait la méthode « equal » de la sous-classe B à être comprise comme une redéfinition de celle de la classe A.

```
class A =
  attribute x : int
  message equal (that : A) =
    (x = that.x)
end

class B =
  inherits A
  attribute y : int
  message equal (that : B) =
    (x = that.x ∧ y = that.y)
end
```

```
function will_explode (i : A) =
  let i' = new A { x = 1 } in
    i.equal (i')
```

Mais, peut-être que . . .

Peut-être que la définition du sous-typage est incorrecte. . . Essayons de voir ce qui se passerait dans un hypothétique langage qui autorisait la méthode « equal » de la sous-classe B à être comprise comme une redéfinition de celle de la classe A.

```
class A =
  attribute x : int
  message equal (that : A) =
    (x = that.x)
end

class B =
  inherits A
  attribute y : int
  message equal (that : B) =
    (x = that.x ∧ y = that.y)
end
```

```
function will_explode (i : A) =
  let i' = new integer { x = 1 } in
    i.equal (i')

function explosion () =
  let i =
    new B { x = 42, bounded = 51 }
  in
    will_explode i
```

Et en SCALA ?

Ce problème nommé « Le problème des méthodes binaires » est **intrinsèque à la programmation objet**. Il existe donc aussi en SCALA. Cependant, nous allons voir qu'il existe des solutions élégantes de ce problème en SCALA, rendues possible grâce à son riche système de type.

```
01 class A (val x : Int) {
02   def equal (that : A) = (that.x == x)
03 }
04
05 class B (x : Int, val y : Int) extends A (x) {
06   def equal (that : B) = super.equal (that) ∧ that.y == y
07 }
08
09 def check (x : A, y : A) =
10   println (x equal y)
11
12 scala> check (new B (1, 0), new B (1, 1))
13 true
```

Première (quasi) solution

```
01 class A (val x : Int) {
02   def equal (that : A) = (that.x == x)
03 }
04
05 class B (x : Int, val y : Int) extends A (x) {
06   override def equal (that : A) = that match {
07     case thatB : B => this.equal (thatB)
08     case thatA : A => super.equal (thatA)
09   }
10   def equal (that : B) = super.equal (that) ∧ that.y == y
11 }
12
13 def check (x : A, y : A) =
14   println (x equal y)
15
16 scala> check (new B (1, 0), new B (1, 1))
17 false
```

L'idée consiste à rajouter une redéfinition effective de la méthode `equal` de la classe `A` dans `B` en lui donnant la même signature. Depuis cette méthode, il est alors possible de déterminer si l'objet auquel on se compare est un `B` et alors appeler la méthode spécialisée pour `B`.

Inconvénients de cette solution

- Elle nécessite l'écriture manuelle de la version redéfinie de la méthode. Ce travail est répétitif, systématique et donc sujet à erreur. Il existe des compilateurs qui font se rajout mais pas en Scala, ni dans aucun langage non académique.
- Elle rajoute des tests dynamiques supplémentaires, ce qui peut être une source d'inefficacité.
- Plus grave : elle casse la **transitivité** de l'égalité !

Démonstration de la perte de la transitivité

```
01 val x = new B (1, 0)
02 val y = new B (1, 1)
03 val z = new A (1)
04
05 scala> x.equal (z)
06 res18: Boolean = true
07
08 scala> x.equal (y)
09 res19: Boolean = false
10
11 scala> y.equal (z)
12 res20: Boolean = true
```

Une idée ?

Seconde (quasi) solution

Une façon de résoudre le problème consiste à décider que la comparaison entre deux objets qui n'ont pas le même type échoue toujours.

```
class A (val x : Int) {
  def equal (that : A) = that match {
    case thatA : A => (that.x == x) ∧ getClass == that.getClass
    case _ => false
  }
}

class B (x : Int, val y : Int) extends A (x) {
  override def equal (that : A) = that match {
    case thatB : B => this.equal (thatB) ∧ getClass == that.getClass
    case _ => false
  }
  def equal (that : B) = super.equal (that) ∧ that.y == y
}
```

Il y a toujours un problème avec cette approche, une idée ?

Le problème des classes anonymes

```
val t = new B (1, 0) { override val y = 1 }

scala> t.equals (y)
res21: Boolean = false
```

La classe anonyme créée par SCALA ne redéfinit pas une nouvelle notion d'égalité : on aimerait qu'elle s'intègre à la relation d'égalité existante sur B.

Dernière version de cette solution au problème des méthodes binaires

```
class A (val x : Int) {
  def equal (that : A) = that match {
    case thatA : A => (that.x == x) ∧ (thatA canEqual this)
    case _ => false
  }
  def canEqual (that : A) = that.isInstanceOf[A]
}

class B (x : Int, val y : Int) extends A (x) {
  override def equal (that : A) = that match {
    case thatB : B =>
      this.x == that.x ∧ this.y == thatB.y ∧ (thatB canEqual this)
    case _ => false
  }
  override def canEqual (that : A) = that.isInstanceOf[B]
}
```

Plan de la partie *"Construire des objets"*

Classe abstraite

Classe abstraite

Une classe abstraite est une classe dont certains membres sont **indéfinis**.

Membre abstrait

Un membre est dit abstrait ou virtuel si il est indéfini dans une classe.

En Scala

En Scala, il suffit de déclarer un membre sans en donner de définition pour le rendre abstrait. Une classe abstraite doit être marquée du modificateur `abstract`.

```
abstract class Shape {
  def description : String
  def show : Unit = println (description)
}

class Square extends Shape {
  override def description = "Hello I am a square."
}

class Circle extends Shape {
  override def draw = "Hello I am a circle."
}
```

Rôle des classes abstraites

Nous verrons que les classes abstraites ont un rôle double dans la programmation orientée objet : elles servent à **factoriser** des composants en s'abstrayant de leur différence et elles servent à dénoter des **concepts**.

Dans l'exemple précédent, effectivement, la classe forme permet d'une part de factoriser le code de la méthode `show` et d'autre part, elle représente le concept de forme, qui n'a pas d'existence physique réelle. En particulier, il n'existe pas de description précise d'une forme mais il en existe pour toutes les instances de ce concept.

Exemple

Les classes abstraites permettent de décrire des relations abstraites entre des concepts. Pour comprendre cette idée, prenons l'exemple de situation suivante :

> *Imaginons l'application d'une séquence de filtres dans un logiciel de retouche d'images. Pour tenir l'utilisateur au courant de la progression du calcul, plusieurs objets s'affichent dans son interface : une barre de progression, un pourcentage dans la barre de titre, une icône sur son bureau ... Le nombre de ces indicateurs n'est a priori pas limité. Comment s'assurer qu'ils seront tous mis au courant de l'avancement du processus ?*

Écrivez un diagramme de classes UML pour décrire une architecture qui serait une solution à ce problème.

Principe de généralisation

Généralisation

Le principe d'inférence ou de généralisation consiste à établir des règles générales s'appliquant à des instances particulières.

C'est un des principes fondamentaux pour concevoir une architecture générale.

Une mauvaise réponse

Une mauvaise façon de traiter le problème précédent consisterait à intégrer le mécanisme de notification des objets au cas par cas, par exemple en appelant explicitant une méthode de chaque objet à notifier.

```
object filter {
  def apply = {
    /* Do something that apply a filter to an image. */
    icon.notify
    titlebar.notify
    progressionbar.notify
  }
}
```

Une bonne réponse

Une meilleure façon de procéder pour résoudre ce problème consiste à **expliciter les concepts** mis en jeu.

Dans ce problème, il y a un **sujet d'observation** et des **observateurs**. Au moment de leur création, les observateurs doivent s'inscrire auprès du sujet pour que ce dernier décide de les notifier lorsque son état est modifié.

Cela signifie qu'il y a deux classes abstraites et qu'il faut définir leur interaction en des termes abstraits. Il suffira ensuite, par héritage, de **spécialiser** ce comportement au cas qui nous intéresse.

Premier essai

```
abstract class Subject {
  private var observers : List[Observer] = Nil
  def register (new_observers : Observer*) =
    observers ++= new_observers
  def notifyObservers =
    observers.foreach (_.onNotify (this))
}

abstract class Observer {
  def onNotify (s : Subject)
}
```

Nous avons capturé la logique de l'interaction entre les observateurs et le sujet. On pourrait cependant être plus fin en autorisant une notification *asynchrone* des observateurs. (Exercice !)

Spécialisation

```
class Filter extends Subject {
  var currentFilterName : String = _
  def apply (filterName: String) = {
    currentFilterName = filterName;
    notifyObservers
  }
}
class Icon extends Observer {
  def onNotify (s : Subject) = s match {
    case filter: Filter =>
      println ("Icon sees " + filter.currentFilterName)
  }
}
class ProgressBar extends Observer {
  def onNotify (s : Subject) = s match {
    case filter: Filter =>
      println ("Progress bar sees " + filter.currentFilterName)
  }
}
```

Critiquez cette implémentation !

Critique de ce premier essai

Cette implémentation a deux défauts apparents principaux.

Tout d'abord, il n'y a pas aucune garantie qu'un observateur soit nécessairement lié à un sujet car au moment de sa création ce n'est pas le cas. Or, notre définition suggère que ce lien doit toujours exister.

Ensuite, bien qu'un observateur sache le type exact de son sujet, il y a une vérification du type de ce dernier dans la méthode qui implémente la réaction à une notification.

Premier problème

Il y a une solution immédiate au premier défaut qui consiste à rajouter un argument au constructeur de tout observateur correspondant au sujet observé. On supprime le second défaut du même coup !

```
abstract class Subject {
  private var observers : List[Observer] = Nil
  def register (new_observers : Observer*) = observers ++= new_observers
  def notifyObservers = observers.foreach (_.onNotify)
}
abstract class Observer (s: Subject) {
  s.register (this)
  def onNotify
}
class Filter extends Subject {
  var currentFilterName : String = _
  def apply (filterName: String) = {
    currentFilterName = filterName;
    notifyObservers
  }
}
class Icon (s: Filter) extends Observer (s) {
  def onNotify = println ("Icon sees " + s.currentFilterName)
}
class ProgressBar (s: Filter) extends Observer (s) {
  def onNotify = println ("Progress bar sees " + s.currentFilterName)
}
```

Une solution peu flexible

La solution précédente peut sembler séduisante et dans certaines situations, elle est effectivement pertinente. Cependant, elle interdit à un observateur d'avoir plusieurs sujets ou même tout simplement de changer de sujet.

Enfin, plus grave encore : la durée de vie d'un sujet est maintenant liée à la vie de ces observateurs puisque ces derniers gardent un pointeur sur leur sujet. Ainsi, même si le sujet est inactif et inutile, il n'est pas libéré de la mémoire.

La première solution n'avait pas ce problème : un observateur pouvait se faire connaître auprès de plusieurs sujets et à la destruction d'un sujet, la relation avec son observateur disparaissait avec lui.

Nous revenons donc sur le premier défaut : cela n'en était pas un.

C'était notre définition initiale qui était inadéquate !

Nouvelle définition

On passe de :

> *Il y a un **sujet d'observation** et des **observateurs**. Au moment de leur création, les observateurs doivent s'inscrire auprès du sujet pour que ce dernier décide de les notifier lorsque son état est modifié.*

à

> *Il y a des **sujets d'observation** et des **observateurs potientiels** de ces sujets. Les observateurs doivent s'inscrire auprès d'un ou plusieurs sujets pour que ces derniers décident de les notifier lorsque leur état est modifié.*

Second problème

Le second problème est plus complexe : il faut synchroniser le type des observateurs avec le type du sujet qu'ils observent. Une première tentative naïve consisterait à écrire :

```
class Icon extends Observer {
  def onNotify (filter : Filter) =
      println ("Icon sees " + filter.currentFilterName)
  }
}
```

Mais cela ne fonctionne évidemment pas. Pourquoi ?

Second problème

Le second problème est plus complexe : il faut synchroniser le type des observateurs avec le type du sujet qu'ils observent. Une première tentative naïve consisterait à écrire :

```
class Icon extends Observer {
  def onNotify (filter : Filter) =
      println ("Icon sees " + filter.currentFilterName)
  }
}
```

Mais cela ne fonctionne évidemment pas. Pourquoi ? Parce que cette méthode `onNotify` n'est pas une redéfinition de la méthode `onNotify` de la classe `Observer`.

Types virtuels à la rescousse !

Une solution élégante à ce problème consiste à définir les deux classes `Object` et `Observer` à partir des types (encore inconnus) de leurs sous-classes.

```
abstract class SubjectObserver {
  /* O is the type of a concrete Observer. */
  type O <: Observer
  /* S is the type of a concrete Subject. */
  type S <: Subject
  /* The following classes are defined using these types. */
  abstract class Subject {
    /* The type of this must be S */
    self:S =>
    var observers : List[O] = Nil
    def register (new_observers : O*) =
      observers ++= new_observers
    def notifyObservers =
      observers.foreach (_.onNotify (this))
  }
  abstract class Observer { def onNotify (s : S) }
}
```

Instanciation

```
object FilterLib extends SubjectObserver {
  type S = Filter
  type O = FilterObserver

  class Filter extends Subject {
    var currentFilterName : String = _
    def apply (filterName: String) = {
      currentFilterName = filterName;
      notifyObservers
    }
  }

  abstract class FilterObserver extends Observer {
    def onNotify (s : S)
  }
}
class Icon extends FilterLib.FilterObserver {
  def onNotify (filter : FilterLib.Filter) =
      println ("Icon sees " + filter.currentFilterName)
}
class ProgressBar extends FilterLib.FilterObserver {
  def onNotify (filter : FilterLib.Filter) =
      println ("Progress bar sees " + filter.currentFilterName)
}
```

Ingrédients : Type virtuel et Type de `this`

Type virtuel

Un type virtuel est un type abstrait qui sera concrétisé uniquement dans une sous-classe concrète.

Type de `this`

Dans une classe, le type de `this` est le type qu'aura l'objet final instancié par la classe courante ou par une de ses sous-classes.

Nous reviendrons sur ces notions dans la suite du cours.

Comment organiser les classes ?

Comme vous venez de le voir, organiser des classes demandent de la réflexion et de la créativité. Pour vous aider, des schémas récurrents de conception ont été exhibés. On les appelle des patrons de conception (*Design Pattern*). La solution précédente est le patron de conception « Sujet-observateur ».

Nous allons en découvrir d'autres tout au long du cours.

Plan de la partie *"Construire des objets"*

Le sous-typage, un mécanisme de modularité

Outre son utilité pour la réutilisation du code, le sous-typage favorise aussi la modularité car c'est aussi un mécanisme d'**encapsulation**.

Pourquoi ? Le sous-typage permet de cacher le **type exact** des objets en ne conservant que leur **type utile**. On dissocie un ensemble de (sur-)types servant d'**interfaces** de communication entre les objets de l'ensemble des (sous-)types servant à décrire leur représentation interne, leur **implémentation**.

Exemple

Imaginons une bibliothèque fournissant des structures de données de dictionnaires :

```
01 class ListDictionary[Key, Data] {
02   private var entries : List[(Key, Data)] = Nil
03   def assoc (key : Key) : Option[Data] =
04     entries.find ({ case (dkey, _) => key == dkey }) map ({ case x => x._2 })
05   def inject (key : Key, data : Data) =
06     entries = (key, data) :: entries
07 }
08
09 class ArrayDictionary[Key, Data] {
10   private var entries : ResizableArray[(Key, Data)] = new ArrayBuffer[(Key, Data)]
11   def get (key : Key) : Option[Data] =
12     entries.find ({ case (dkey, _) => key == dkey }) map ({ case x => x._2 })
13   def push (key : Key, data : Data) =
14     entries :+ (key, data)
15 }
```

Critiquez la conception de cette bibliothèque !

Première critique : Non uniformité du nommage

La première critique évidente vis-à-vis de la modularité s'appuie sur la non uniformité du nommage. En effet, imaginons un client qui utiliserait de façon intensive la classe `ListDictionary` de cette bibilothèque. Son code source (faisant plusieurs millions de ligne de code) contiendrait des milliers d'instanciation et d'appels de méthode de la forme :

- `new ListDictionary[K, D]`
- e.assoc (key)
- e.inject (key, data)

Si ce client décide de remplacer ses utilisations de cette classe par la classe `ArrayDictionary` alors il devra modifier toutes ces expressions ! Ouch !

Correction du problème de nommage

```
01 class ListDictionary[Key, Data] {
02   private var entries : List[(Key, Data)] = Nil
03   def assoc (key : Key) : Option[Data] =
04     entries.find ({ case (dkey, _) => key == dkey }) map ({ case x => x._2 })
05   def inject (key : Key, data : Data) =
06     entries = (key, data) :: entries
07 }
08
09 class ArrayDictionary[Key, Data] {
10   private var entries : ResizableArray[(Key, Data)] = new ArrayBuffer[(Key, Data)]
11   def assoc (key : Key) : Option[Data] =
12     entries.find ({ case (dkey, _) => key == dkey }) map ({ case x => x._2 })
13   def inject (key : Key, data : Data) =
14     entries :+ (key, data)
15 }
```

Il reste encore une certaine duplication de code. . .

Classe abstraite pour factoriser du code

```
01 abstract class SearchableEntries[Key, Data] {
02   type Entries <: {
03     def find (p : ((Key, Data)) => Boolean) : Option[(Key, Data)]
04   }
05   protected var entries : Entries
06   def assoc (key : Key) : Option[Data] =
07     entries.find ({ case (dkey, _) => key == dkey }) map ({ case x => x._2 })
08 }
09
10 class ListDictionary[Key, Data] extends SearchableEntries[Key, Data] {
11   type Entries = List[(Key, Data)]
12   override protected var entries : Entries = Nil
13   def inject (key : Key, data : Data) =
14     entries = (key, data) :: entries
15 }
16
17 class ArrayDictionary[Key, Data] extends SearchableEntries[Key, Data] {
18   type Entries = ResizableArray[(Key, Data)]
19   override protected var entries : Entries = new ArrayBuffer[(Key, Data)]
20   def inject (key : Key, data : Data) =
21     entries :+ (key, data)
22 }
```

Le type virtuel `Entries` dénote la représentation final de l'ensemble des entrées du dictionnaries. Ce n'est toujours pas totalement satisfaisant...

Nécessité d'encapsulation

Un client de cette bibliothèque respecte maintenant un protocole uniforme d'interaction avec un dictionnaire. Cependant, il reste deux problèmes.

D'abord, il doit encore choisir explicitement une classe concrète à instancier et s'il veut passer d'une classe à une autre, il faut qu'il change le nom de cette classe en tout point de son code.

Ensuite, comme la bibliothèque divulgue l'implémentation interne des deux structures de données, il se peut qu'il s'appuie de façon abusive sur la logique interne de ces types d'objets. Dans notre cas, un client utilisant la classe `ListDictionary` pourrait appuyer ses algorithmes le fait que si deux entrées sont ajoutées pour la même clé, la dernière prévaut.

Qu'arrive-t-il si il décide d'utiliser la classe `ArrayDictionary` ou si dans une nouvelle version de la bibliothèque, cette propriété n'est plus respectée ?

Une classe abstraite pour interface d'utilisation

```
abstract class Dictionary[Key, Data] {
  /* Add a new entry that associates [key] to [data]. If there */
  /* already is an entry with that key, the behavior is unspecified. */
  def inject (key : Key, data : Data) : Unit
  def assoc (key : Key) : Option[Data]
}

abstract class SearchableDictionary[Key, Data] extends Dictionary[Key, Data] {
  type Entries <: {
    def find (p : ((Key, Data)) => Boolean) : Option[(Key, Data)]
  }
  protected var entries : Entries
  def assoc (key : Key) : Option[Data] =
    entries.find ({ case (dkey, _) => key == dkey }) map ({ case x => x._2 })
}

class ListDictionary[Key, Data] extends SearchableEntries[Key, Data] {
...
}

class ArrayDictionary[Key, Data] extends SearchableEntries[Key, Data] {
...
}
```

L'interface est un contrat

Même si les classes concrètes sont publiées par la bibliothèque, il est souvent plus modulaire et plus robuste pour le client de spécifier ces propres interfaces et d'écrire ses propres fonctions en s'appuyant seulement sur les interfaces. En effet, une interface décrit ce que **promet** le concepteur de la bibliothèque à ces clients. D'une version à l'autre d'une bibliothèque, les modifications des APIs sont en général minimes.

(Nous reviendrons sur cette propriété un peu plus loin.)

Existe-t-il un moyen de renforcer cette discipline ?

Le patron de conception Usine

```
abstract class Dictionary[Key, Data] {
  /* Add a new entry that associates [key] to [data]. If there */
  /* already is an entry with that key, the behavior is unspecified. */
  def inject (key : Key, data : Data) : Unit
  def assoc (key : Key) : Option[Data]
}

object Dictionary {
  sealed abstract class Criteria
  case object FastestInject extends Criteria
  case object FastestAssoc extends Criteria
  // The dictionary must not weight too much in memory
  case object SmallestFootPrint extends Criteria

  /* Instantiate a dictionary w.r.t. a list of ordered criteria. */
  def apply [Key, Data](criteria : List[Criteria]) : Dictionary[Key, Data] = {
    type Default = ListDictionary[Key, Data]
    criteria match {
      case Nil => new Default
      case FastestAssoc :: _ => new ArrayDictionary[Key, Data]
      case FastestInject :: _ => new ListDictionary[Key, Data]
      case SmallestFootPrint :: xs => apply (xs)
  }
}
  /* Here the implementation of the subclasses, which are */
  /* private to the package. */
```

Extensibilité et modularité via l'utilisation des usines

Pour construire un dictionnaire, un client doit désormais nécessairement utiliser l'objet compagnon `Dictionary` en fournissant les critères de performances attendues du dictionnaire à construire.

Par conséquent, le client ne fait plus référence directement aux noms des sous-classes. La documentation de bibliothèque ne mentionne alors plus du tout ces sous-classes. Il n'y a donc aucune bonne raison pour que le client s'appuie sur leur logique interne.

De plus, la bibliothèque peut proposer de nombreux protocoles distincts de construction de dictionnaires (en fonction du nombre maximal d'entrées par exemple).

Enfin, si la bibliothèque rajoute une nouvelle implémentation de dictionnaire, par exemple un dictionnaire avec une recherche plus rapide (mais une empreinte mémoire plus importante) s'appuyant sur un arbre de recherche, la migration ne nécessite aucune adaptation pour le client. Le concepteur de la bibliothèque ne devra modifier que l'usine pour prendre en compte cette nouvelle sous-classe.

Plan de la partie *"Construire des objets"*

Présentation du problème

Utiliser l'héritage comme un procédé de réutilisation de code peut s'avérer assez subtil. Nous avons déjà vu avec le problème des méthodes binaires que certaines redéfinitions qui pourraient sembler licites ne le sont en fait pas pour des raisons de sûreté de l'exécution. Heureusement, dans ce cas, le système de type du langage de programmation nous empêche de se méprendre sur la forme des redéfinitions autorisées dans les sous-classes.

Malheureusement, il existe des problèmes de **conception** des classes de base pour lesquels un système de types standard n'est d'aucune aide. C'est le cas du problème dit de la classe de base fragile.

Exemple

Tiré de *A study of the fragile base class problem.* Mikhajlov, Sekerinski

Voici une classe qui implémente une structure de données contenant des entiers :

```
01 class Bag {
02    private var elements : List[Int] = _
03    def add (x: Int) = elements = x :: elements
04    def addAll (xs : List[Int]) = xs foreach add
05    def cardinal = elements.size
06 }
```

Exemple

Tiré de *A study of the fragile base class problem.* Mikhajlov, Sekerinski

On veut proposer une version optimisée de la structure précédente calculant le cardinal incrémentalement :

```
01 class CountingBag extends Bag {
02    var counter = 0
03    override def add (x: Int) = { counter += 1 ; super.add (x) }
04    override def cardinal = counter
05 }
```

Quelles sont les hypothèses faites ici sur la classe mère ?

Le problème de la classe de base fragile

On optimise la classe mère en modifiant sa méthode `addAll` :

```
01 class Bag {
02    private var elements : List[Int] = _
03    def add (x: Int) = elements = x :: elements
04    def addAll (xs : List[Int]) = elements ++= xs
05    def cardinal = elements.size
06 }
```

Est-ce que la classe fille est toujours correctement implémentée ?

Principe de substitutivité étendu aux raffinements

Une nouvelle contrainte sur les héritages interdits ou autorisés apparaît ici : une sous-classe B peut hériter sans danger d'une classe A, si son type induit est un sous-type de cette classe A et **des raffinements futurs de cette classe** A.

Cette idée est assez difficile à formaliser et à mettre en pratique. En observant des exemples pathologiques qui ne respectent pas ce principe, nous allons établir une discipline de conception permettant de résoudre ce problème.

Exemple 1 : Récursion mutuelle non anticipée

```
01 class I {
02   var i : Int = 0
03   def m = i += 1
04   def n = i += 1
05 }
06
07 class SI extends I {
08   override def n = m
09 }
```

Imaginez une nouvelle version de `I` qui mettrait en cause l'implémentation de `SI`.

Exemple 1 : Récursion mutuelle non anticipée

```
01 class I {
02   var i : Int = 0
03   def m = n
04   def n = i += 1
05 }
06
07 class SI extends I {
08   override def n = m
09 }
```

Ouch ! L'héritage peut créer des cycles inattendus !

Fausses hypothèses sur la classe mère

```
01 class SquareRoots {
02    def root2 (x : Float) = sqrt (x)
03    def root4 (x : Float) = pow (x, 0.25)
04 }
05
06 class OtherSquareRoots extends {
07    def root2 (x : Float) = - sqrt (x)
08 }
```

Imaginez une nouvelle version de `SquareRoots` qui mettrait en cause l'implémentation de `OtherSquareRoots`.

Fausses hypothèses sur la classe mère modifiée

```
01 class SquareRoots {
02    def root2 (x : Float) = sqrt (x)
03    def root4 (x : Float) = sqrt (sqrt (x))
04 }
05
06 class OtherSquareRoots extends {
07    def root2 (x : Float) = - sqrt (x)
08 }
```

Ouch ! Même si on a le code de `root2` devant ses yeux lorsque l'on programme `root4`, il ne faut pas faire d'hypothèses trop fortes sur la façon dont la méthode `root2` est implémentée.

Fausses hypothèses sur la classe fille

```
01 class A {
02    def l (k : Int) = assert (k ≥ 5)
03    def m (k : Int) = l (k)
04    def n (k : Int) = {}
05 }
06
07 class B extends A {
08    override def l (k : Int) = {}
09    override def n (k : Int) = m (k)
10 }
```

Imaginez une nouvelle version de `A` qui mettrait en cause l'implémentation de `B`.

Fausses hypothèses sur la classe fille

```
01 class A {
02    def l (k : Int) = assert (k ≥ 5)
03    def m (k : Int) = assert (k ≥ 5)
04    def n (k : Int) = {}
05 }
06
07 class B extends A {
08    override def l (k : Int) = {}
09    override def n (k : Int) = m (k)
10 }
```

Ouch ! L'implémentation d'une classe mère ne doit pas non plus faire d'hypothèses trop fortes sur l'implémentation de la classe fille.

Accès direct à l'état de la classe mère

```
01 class C {
02   var x : Int = 0
03   def m = x += 1
04   def n = x += 2
05 }
06
07 class D extends C {
08   override def n = x += 2
09 }
```

Imaginez une nouvelle version de `C` qui mettrait en cause l'implémentation de `D`.

Accès direct à l'état de la classe mère

```
01 class C {
02   var x : Int = 0
03   var y : Int = 0
04   def m = { y += 1 ; x = y }
05   def n = { y += 2 ; x = y }
06 }
07
08 class D extends C {
09   override def n = x += 2
10 }
```

Ouch ! La classe fille casse l'invariant de la classe mère maintenant le fait que `x = y`.

Fausse hypothèse sur les invariants d'une classe de base

```
01 abstract class E {
02   private var x : Int = 0
03   def getX = x
04   def incrX = { x += 1 ; onIncrX }
05   def onIncrX : Unit
06 }
07
08 class F extends E {
09   var y : Int = 0
10   override def incrX = { y += 1 ; super.incrX }
11   def onIncrX = assert (getX == y)
12 }
```

Imaginez une nouvelle version de `E` qui mettrait en cause l'implémentation de `F`.

Fausse hypothèse sur les invariants d'une classe de base

```
01 abstract class E {
02   private var x : Int = 0
03   def getX = x
04   def incrX = { onIncrX; x += 1 }
05   def onIncrX : Unit
06 }
07
08 class F extends E {
09   var y : Int = 0
10   override def incrX = { y += 1; super.incrX }
11   def onIncrX = assert (getX == y)
12 }
```

La classe fille a supposé à tort que les appels de la méthode `onIncrX` s'effectuait après que la méthode `incrX` ait été évaluée.

Qu'en retenir ?

Nous venons de voir beaucoup de cas distincts de hiérarchies de classe (trop) sensibles aux évolutions futures. Pour prévenir ce genre de problème, il faut **découpler** au maximum l'implémentation des sous-classes de celle des classes mères, donc utiliser avec beaucoup de précaution l'héritage d'implémentation.

- Il ne faut pas qu'une sous-classe et qu'une révision puisse introduire des cycles de récursion mutuelle.
- Quand on développe une méthode d'une classe mère, il ne faut pas faire d'hypothèses trop fortes sur l'implémentation des autres méthodes de cette classe.
- Quand on développe une méthode de sous-classe, on doit faire comme si les méthodes de classe de base ne pouvaient pas être redéfinies.
- Une sous-classe ne doit jamais accéder directement à l'état interne d'une classe de base (*i.e.* il faut bannir le mot-clé `protected`).

Comment écrire des classes de base stables ?

Il y a une autre façon de résoudre le problème de la classe fragile, beaucoup plus extrême : ne jamais modifier la logique interne d'une classe dont dépendent d'autres classes. On peut se permettre de corriger des erreurs sans changer fondamentalement la logique interne ou bien encore rajouter des fonctionnalités, mais c'est tout !

Comment maximiser les chances d'être dans cette situation ?
C'est l'art de la conception... Le mot d'ordre est de voir une classe de base comme une interface aussi bien pour les classes des autres hiérarchies que pour ses futures sous-classes. Il faut alors suivre le mot d'ordre suivant valable pour toute conception d'interface :

Publier une interface simple et réduite.

Comment m'assurer que l'interface est simple et réduite ?

Un moyen de vérifier la simplicité de votre interface est de **documenter** votre classe mère en direction de deux types de client :

- Celui qui l'**utilise** directement en lui expliquant le lien entre les entrées et les sorties des méthodes ainsi que leurs effets sur l'état de l'objet, décrit de façon abstraite.
- Celui qui **hérite** en décrivant la façon dont les méthodes peuvent être redéfinies et en décrivant les états de l'objet qui doivent être supposés avant l'évaluation de chacunes d'elles.

Cette documentation devra servir de référence à respecter pour les futures évolutions de votre classe.

Le patron de conception Facade

Un cas assez courant de simplification vertueuse d'interface survient lorsque l'on utilise une bibliothèque externe dans un développement.

Prenons l'exemple d'une bibliothèque externe servant à interagir avec un système de gestion de base de données. Ce système peut être très évolué et proposer un très grand nombre de fonctionnalités réalisées par l'interaction d'un grand nombre d'objets.

Une classe facade est une **couche d'abstraction** de cet ensemble de fonctionnalités en un sous-ensemble restreint correspondant à ce dont à besoin le reste du développement. Elle correspond à un **point d'entrée** dans ce système externe.

Exemple de façade

Voici une façade très simple pour interagir avec une base de donnée :

```
01 abstract class DB {
02    type Table = List[String]
03    type Row = List[String]
04    type TableName = String
05    def connect : Unit
06    def create (table : Table) : String
07    def insert (table : TableName, row : Row) : Unit
08    def select (table : TableName, field : String, value : String) : List[Row]
09 }
```

La migration d'un système de base de données à un autre est facilité par ce genre d'abstraction. (Attention cependant à ne pas confondre "abstraction simple" et "abstraction simpliste"...)

Adaptateur

Plutôt que de revenir sur une interface déjà fixée pour la modifier de façon importante, il est parfois possible et plus pertinent de fournir des **adaptateurs** permettant de passer d'une interface à une autre.

Exemple

```
01 abstract class OptimisticDictionary[Key, Data] {
02   def inject (key : Key, data : Data) : Unit
03   def assoc (key : Key) : Data
04 }
05
06 class ToOptimisticAdapter[Key, Data, D <: Dictionary[Key, Data]]
07 (dict : D)
08 extends OptimisticDictionary[Key, Data] {
09   def inject (key : Key, data : Data) : Unit = dict.inject (key, data)
10   def assoc (key : Key) : Data =
11     dict.assoc (key) match {
12       case None => error ("Unknown key.")
13       case Some (v) => v
14     }
15 }
16
17 class FromOptimisticAdapter[Key, Data, D <: OptimisticDictionary[Key, Data]]
18 (dict : D)
19 extends Dictionary[Key, Data] {
20   def inject (key : Key, data : Data) : Unit = dict.inject (key, data)
21   def assoc (key : Key) : Option[Data] =
22     try Some (dict.assoc (key))
23     catch {
24       case exn => None
25     }
26 }
```

Coercions implicites en Scala

```
01 object adapters {
02
03   implicit def
04   toOptimisticAdapter [Key, Data, D <: Dictionary[Key, Data]]
05   (dict : D)
06   : OptimisticDictionary[Key, Data] = new OptimisticDictionary[Key, Data] {
07     def inject (key : Key, data : Data) : Unit = dict.inject (key, data)
08     def assoc (key : Key) : Data =
09       dict.assoc (key) match {
10         case None => error ("Unknown key.")
11         case Some (v) => v
12       }
13   }
14
15   implicit def
16   fromOptimisticAdapter[Key, Data, D <: OptimisticDictionary[Key, Data]]
17   (dict : D)
18   : Dictionary[Key, Data] = new Dictionary[Key, Data] {
19     def inject (key : Key, data : Data) : Unit = dict.inject (key, data)
20     def assoc (key : Key) : Option[Data] =
21       try Some (dict.assoc (key))
22     catch {
23       case exn => None
24     }
25   }
26 }
```

Coercions implicites en SCALA

```
01 object test {
02    import adapters._
03    def main (args: Array[String]) : Unit = {
04       val d = Dictionary[Int, Int] (List (Dictionary.FastestInject))
05       d.inject (1, 2)
06       d.inject (2, 1)
07       val od : OptimisticDictionary[Int, Int] = d
08       od.inject (3, 0)
09    }
10 }
```

L'appel à la fonction **implicite** de conversion du type `Dictionary` vers le type `OptimisticDictionary` est automatiquement inséré par le compilateur. Nous reviendrons sur ce mécanisme.

Plan de la partie *"Construire des objets"*

Problème de l'extensibilité fonctionnelle

Étant donnée une hiérarchie de classes comme celle-ci :

```
01 abstract class Shape { def description : String}
02 class Square extends Shape { def description = "A square" }
03 class Circle extends Shape { def description = "A circle" }
```

Il est très simple de l'étendre en rajoutant un **nouveau type de donnée**.
Il suffit d'écrire une nouvelle sous-classe :

```
01 class Triangle extends Shape { def description = "A triangle" }
```

Mais comment rajouter une nouvelle méthode **sans modifier** le code existant ?

Première tentative

La façon naïve d'opérer consiste à rajouter un nouvel étage à la hiérarchie :

```
01 class XCircle extends Circle { def area : Int = ... }
02 class XSquare extends Square { def area : Int = ... }
```

Très bien, mais qu'en est-il de la classe abstraite `Shape` ? Comment lui rajouter la méthode `area` ?

Deuxième tentative

Si on essaie d'étendre la hiérarchie en partant de la classe `Shape`...

```
01 class XShape extends Shape { def area : Int }
02 class XSquare extends XShape { ? }
03 class XCircle extends XShape { ? }
```

...on ne sait pas trop de quelle classe hériter dans les classes filles.

Héritage multiple ?

Une (fausse) solution consisterait à utiliser l'héritage multiple (qui n'existe pas en SCALA d'ailleurs) :

```
01 class XShape extends Shape { def area : Int }
02 class XSquare extends XShape, Square { def area = ... }
03 class XCircle extends XShape, Circle { def area = ... }
```

En C++, qui propose l'héritage multiple, cette solution fonctionnerait . . . temporairement puisqu'à la prochaine extension du même genre un héritage en diamant surviendrait (et tous les problèmes que vous connaissez sur ce sujet).

Le patron de conception VISITOR

Si le nombre d'extensions de fonctionnalités d'une hiérarchie de classes n'est pas prévisible à l'avance, on peut définir une classe de **visiteur** capable de discriminer entre les différentes sous-classes :

```
01 abstract class ShapeVisitor {
02   def visit (s : Square)
03   def visit (c : Circle)
04   def visit (s : Shape) = s.accept (this)
05 }
06
07 abstract class Shape {
08   def description : String
09   def accept (v: ShapeVisitor)
10
11 }
12
13 class Square extends Shape {
14   def description = "A square"
15   def accept (v: ShapeVisitor) = v visit this
16 }
17
18 class Circle extends Shape {
19   def description = "A circle"
20   def accept (v: ShapeVisitor) = v visit this
21 }
```

Extension des données

Lorsque l'on rajoute une nouvelle sous-classe, il faut étendre le visiteur pour qu'il la prenne en compte :

```
01 abstract class NewShapeVisitor extends ShapeVisitor {
02   def visit (t : Triangle)
03 }
04
05 class Triangle extends Shape {
06   def description = "A triangle"
07   def accept (v: ShapeVisitor) =
08     v match {
09       case v: NewShapeVisitor => v visit this
10     }
11 }
```

Qu'arrive-t-il si on utilise un visiteur de type `ShapeVisitor` et non pas `NewShapeVisitor` pour visiter une instance de la classe `Triangle` ?

Extension des fonctionnalités

```
01 class AreaVisitor extends ShapeVisitor {
02     def visit (s: Square) = ...
03     def visit (s: Circle) = ...
04 }
```

Pour utiliser cette nouvelle fonctionnalité, on peut écrire une fonction de la forme :

```
01 def area (s: Shape) = {
02     (new AreaVisitor) visit s
03 }
```

Remarquez que cette fonction travaille en monde clos : si une nouvelle sous-classe est rajoutée, elle ne sera pas prise en compte.

Problème ouvert : Comment travailler en monde ouvert et toujours permettre de nouvelles extensions de données et de fonctionnalités ?

Remplacement du visiteur par une analyse par motifs

La dernière remarque nous incite à jeter à la poubelle le visiteur. Autant, le remplacer par une analyse de motif de la forme :

```
01 def area (s: Shape) =
02   s match {
03     case square: Square => ...
04     case circle : Circle => ...
05   }
06 }
```

Cette fonction est équivalente et ne nécessite pas l'écriture de la classe `ShapeVisitor` et des méthodes associées.

Nous verrons cependant que les `trait`s de SCALA redonneront de la pertinence au patron de visiteur...

Un problème en suspens. . .

Problème ouvert :
Comment travailler en monde ouvert et toujours permettre de nouvelles extensions de données et de fonctionnalités ?

En d'autres termes, comment **composer les extensions** ?

Plan de la partie *"Construire des objets"*

Situation-problème

Réfléchissons sur la conception d'un logiciel de dessin :

> *Dans le module noyau responsable de la gestion des formes que l'on peut utiliser pour produire une image, on souhaite proposer plusieurs types de formes (un carré, un triangle, un cercle...) et aussi plusieurs façon de représenter cette forme (une équation, une énumération de points, une image binaire...). Une forme doit proposer une méthode pour s'afficher à une position donnée dans une image binaire. Les cas particuliers de formes doivent apparaître dans l'interface de ce module.*

Comment représenter un cercle représenter par une équation, un carré représenté par une équation, un cercle représenté par une énumération de points, ... bref, **toute combinaison** d'un cas particulier de forme et de représentation de l'ensemble de ses points ?

Une solution (très mauvaise)

L'idée sans doute la plus maladroite serait de créer une classe par combinaison !

```
abstract class Shape { def draw (p : Position, i : Bitmap) : Unit }
class EquationCircle (val radius: Int) extends Shape {
  def draw (p : Position, i : Bitmap) =
    // Set all point p' in the bitmap
    // such that | p' - p | = radius.
}
class EquationSquare (val dimension : Int) extends Shape {
  def draw (p : Position, i : Bitmap) =
    // Connect the four corner points p'
    // such that | p' - p | = sqrt(2)/2 * dimension.
}
class EnumerationCircle (val radius: Int) extends Shape {
  val points : List[Position] =
    // Compute an enumeration of points that represents
    // a circle.
  def draw (p : Position, i : Bitmap) =
    // For each point p' in points, set p + p'.
}
```

Deux solutions standards

Une première solution consiste à utiliser l'**héritage multiple** si le langage de programmation le propose. Nous allons voir que ce mécanisme pose de nombreux problèmes.

Une autre solution consiste à une utilise une relation de **composition** entre deux classes qui sont en relation parce que la première **délègue** un certain nombre d'opérations à la seconde qui les implémentent concrètement. Ce patron de conception s'appelle le **pont**.

Le pont

```
abstract class Points { def draw (p : Position, i : Bitmap) : Unit }

abstract class Shape {
  val points : Points
  def draw (p : Position, i : Bitmap) : Unit = points.draw (p, i)
}

class Enumeration (l : List[Position]) extends Points {
  def draw (p : Position, i : Bitmap) : Unit = ...
}

class Equation (e : Bitmap => List[Position]) extends Points {
  def draw (p : Position, i : Bitmap) : Unit = ...
}

abstract class PointSetKind
case object EnumerationKind extends PointSetKind
case object EquationKind extends PointSetKind

class Circle (val radius : Int, val kind: PointSetKind) extends Shape {
  val points : Points = kind match {
    case EnumerationKind => new Enumeration (...)
    case EquationKind => new Equation ((b : Bitmap) => ...)
  }
}
```

Rôle et généralisation

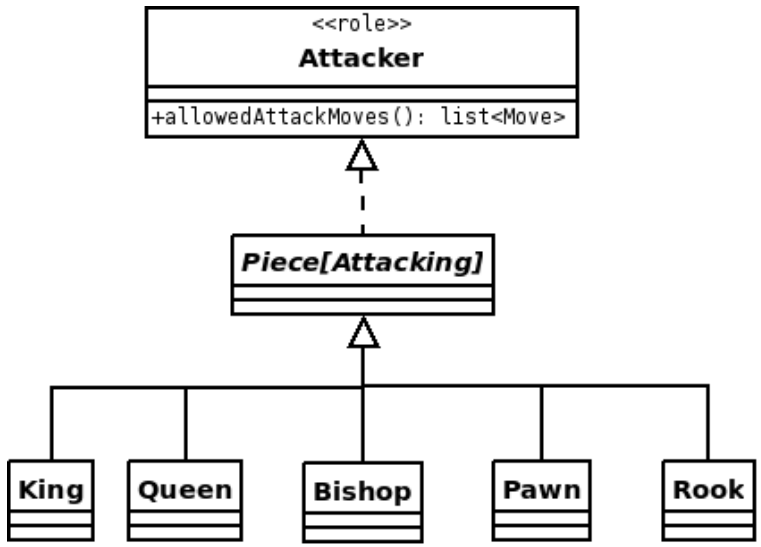
Un objet peut vérifier un prédicat de la forme “Est(X)” pendant un temps **limité**. On dit qu'il a alors un **rôle**. Un rôle sert de classification **dynamique** tandis que les relations de généralisations entre classe fille et classe mère servent de classification statique.

Exemple

Dans un jeu d'échec, lors qu'une pièce tente la prise d'une autre pièce, elle a le rôle d'un attaquant. Le gestionnaire de règle doit vérifier que le coup est valide, en fonction du type de la pièce. Il attend donc d'un attaquant qu'il décrive les règles qui lui sont spécifiques.

Vision logique d'un rôle

En UML, on note les rôles de la façon suivante :

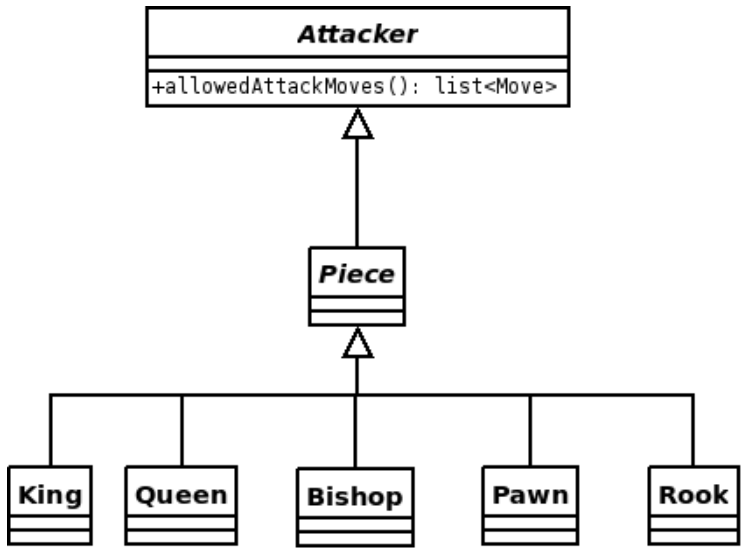


La notation "Piece[Attacking]" signifie que ce diagramme décrire cette classe lorsqu'une de ses instances jouent le rôle d'attaquant.

La flêche pointillée signifie une relation de **dépendance** entre une instance de la classe "Piece" quand elle joue le rôle d'attaquant et la classe qui décrit ce qu'est un attaquant.

Comment **implémenter** cette vue logique de l'architecture ?

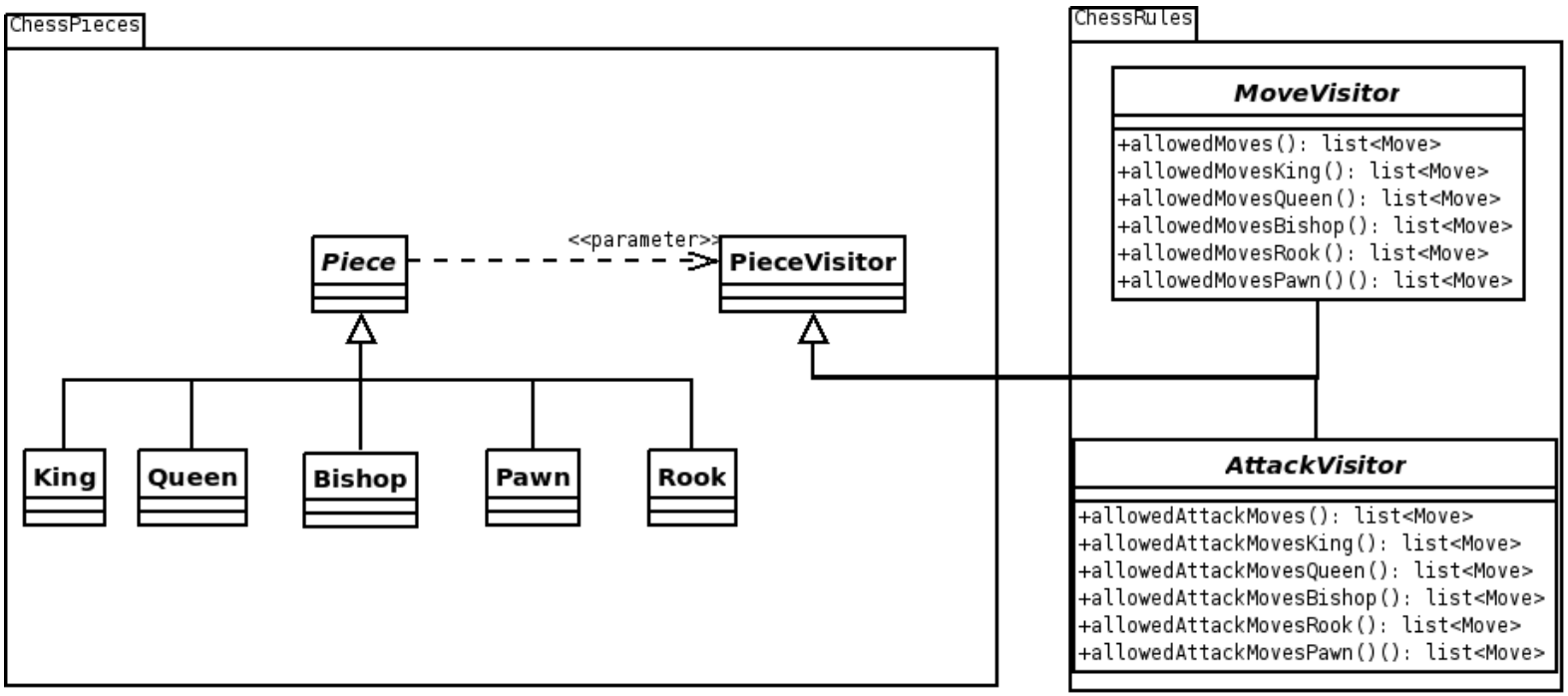
Première (mauvaise) proposition



On a remplacé la relation de dépendance par une relation de généralisation et on traduit cette dernière par un héritage. Dès lors, toutes les sous-classes doivent implémenter la méthode `allowedAttackMoves` qui se confondera certainement avec la méthode `allowedMoves` implémentée par ailleurs.

À l'aide d'une autre architecture, peut-on obtenir une garantie, grâce au typage, que l'on ne peut appeler cette méthode `allowedAttackMoves` qu'en présence d'une pièce ayant le rôle d'un attaquant ?

Seconde proposition



Une première idée consiste à fragmenter les méthodes d'une pièce en extrayant un sous-ensemble de méthode qui peut varier dynamiquement. On peut utiliser deux visiteurs différents pour implémenter les deux types de coups autorisés en fonction du rôle de la pièce.

Critiquez cette architecture !

Troisième proposition : le patron “décorateur”

Une autre façon de voir ce problème est de considérer une pièce attaquante comme une **pièce temporairement décorée** par une fonctionnalité (le calcul des coups d'attaque autorisés).

```
abstract class Piece {
  def allowed_moves : List[Move]
}

class Queen extends Piece { def allowed_moves = Nil }
class Kind extends Piece { def allowed_moves = Nil }

class Attacking[A <: Piece] (piece: A) extends Piece {
  def allowed_attack_moves : List[Move] =
    piece match {
      case as_queen: Queen => Nil
      case as_kind: Kind => Nil
    }
  // Delegate the other methods to the piece
  def allowed_moves = piece.allowed_moves
}
class ChessBoard {
  def display_attack_moves (s : Attacking[Piece]) { ... }
}
```

Garantie statique

L'intérêt de la classe `Attacking` est qu'elle décrit statiquement le rôle de l'objet. Si une fonction attend une pièce attaquante, elle peut maintenant le préciser dans son prototype. En d'autres termes, le programmeur est obligé d'enpaqueter explicitement une pièce dans ce décorateur pour pouvoir l'utiliser comme une pièce attaquante.

Cette solution est assez lourde à utiliser. Pourquoi ?

Garantie statique

L'intérêt de la classe `Attacking` est qu'elle décrit statiquement le rôle de l'objet. Si une fonction attend une pièce attaquante, elle peut maintenant le préciser dans son prototype. En d'autres termes, le programmeur est obligé d'enpaqueter explicitement une pièce dans ce décorateur pour pouvoir l'utiliser comme une pièce attaquante.

Cette solution est assez lourde à utiliser. Pourquoi ? Parce que l'on doit écrire explicitement les délégations des méthodes de la classe `Piece` à l'instance contenue dans l'attribut `piece`. De plus, les méthodes spécifiques aux sous-classes ne sont plus accessibles. . .

Solution qui ne fonctionne pas

Malheureusement, en SCALA, on ne peut pas écrire :

```
class Attacking[A <: Piece] (piece: A) extends A {
  ...
}
```

qui permettrait d'hériter automatiquement des méthodes de classe A :

```
scala> class Attacking[A <: Piece] (piece: A) extends A {}
<console>:6: error: class type required but A found
        class Attacking[A <: Piece] (piece: A) extends A {}
```

D'ailleurs, cela ne règlerait pas tout à fait le problème puisqu'on aimerait aussi dire que `this` est en fait `piece`...

Heureusement, les traits vont offrir une solution élégante à ce problème.

L'héritage multiple

Revenons maintenant sur l'héritage multiple. C'est un mécanisme qui permet d'hériter de plusieurs classes en même temps. En théorie, ce serait une solution idéale à notre problème... mais en pratique, c'est un mécanisme qui souffre de nombreux défauts.

Le problème de l'ambiguité

Voici un exemple tiré de *Programming in Scala* de Martin Odersky :

```
abstract class IntQueue {
  def get(): Int
  def put(x: Int)
}
class BasicIntQueue extends IntQueue {
  private val buf = new ArrayBuffer[Int]
  def get() = buf.remove(0)
  def put(x: Int) { buf += x }
}
class Doubling extends IntQueue {
  override def put(x: Int) { super.put(2 * x) }
}
class Incrementing extends IntQueue {
  override def put(x: Int) { super.put(x + 1) }
}

// The following code is not accepted in Scala
class MyQueue extends Doubling and IntQueue {}
```

Quelle serait la méthode `put` de la classe `MyQueue` ?

Le problème du diamant

La tentative de suppression de l'ambiguïté suivante :

```
class MyQueue extends Doubling and IntQueue {
   def put(x: Int) {
     Incrementing.super.put(x) // (Not real Scala)
     Doubling.super.put(x)
  }
}
```

fait apparaître un autre problème : la méthode `put` de la classe mère `BasicIntQueue` est appelée deux fois !

Les problèmes d'implémentation

L'héritage multiple pose enfin des problèmes d'implémentation dans les compilateurs. (Souvenez de l'héritage `public virtual` de C++.) En effet, déterminer statiquement la disposition des champs des différents attributs de l'objet est difficile car la relation d'héritage ne définit plus un ordre total. (La solution par C++ nécessite une modification silencieuse du pointeur qui tient l'objet lorsque l'on veut voir une sous-classe du point de vue d'une de ses classes mères.)