Le trait, Saint-Graal de la modularité

Cours : Gustavo Petri

TD : Aldric Degorre,
Adrien Pommellet

Slides : Yann Régis-Gianas

Université Denis Diderot – Paris 7

Sous-typer avec classe

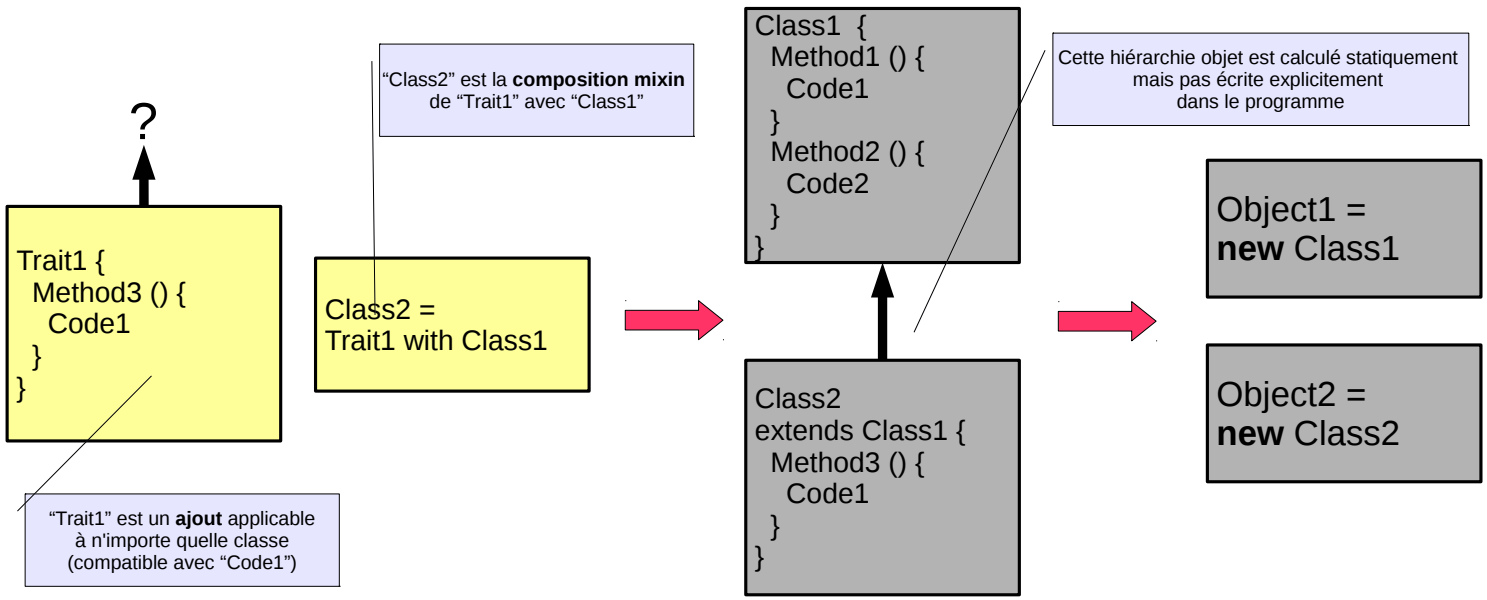
Construire des hiérarchies par composition mixin

Les traits et la composition mixins

Une alternative à l'héritage multiple

Plan de la partie *"Construire des objets"*

Construire des hiérarchies par composition mixin



Qu'est-ce qu'un trait ?

Un **trait** représente une **modification** (un ou plusieurs ajouts ou redéfinitions de méthodes) à appliquer à une classe.

Lorsque l'on écrit un trait, il faut considérer que l'on est paramétré par l'instance de sa super-classe : super est inconnu. Habituellement, la définition de l'instance que l'on spécifie quand on écrit une classe, est seulement paramétrée par l'instance d'une éventuelle sous-classe.

Les traits implémentent une double récursion ouverte !

Les traits en Scala

En Scala, la syntaxe pour définir des traits est similaire à celle des classes sauf que l'on utilise le mot-clé **trait** pour les définir.

```
trait A {
  def foo = ...
}
```

On ne peut évidemment pas instancier un trait ! En effet, il lui manque la valeur de super pour pouvoir être réaliser concrètement.

Composition *mixin* avec un trait

Pour instancier un trait, il faut utiliser le mécanisme de **composition mixin**. Il s'agit de donner la classe mère sur laquelle le trait – la modification – doit être appliqué. Ainsi :

```
class A { def n = ... }

trait C { def m = ... }

class B extends A with C {}
```

est (à peu près) équivalent à :

```
class A { def n = ... }
class B extends A {
  def m = ...
}
```

Un trait représente donc un **changement incrémental réutilisable**.

Création de classe à la volée

On peut appliquer la composition mixin au moment de l'instanciation ; ce qui évite de créer explicitement des classes (et de leur donner un nom) :

```
class A { def n = ... }

trait C { def m = ... }

val x = new A with C
```

Composition *mixin* avec plusieurs traits

On peut effectuer une composition mixin avec plusieurs traits à la fois :

```
class A { def n = ... }
trait C { def m = ... }
trait D { def k = ... }
class B extends A with C with D {}
```

qui est équivalent (à peu près) à :

```
class A { def n = ... }
class B extends A {
   def m = ...
   def k = ...
}
```

Redéfinition et composition *mixin*

Que se passe-t-il dans le programme suivant ?

```
class A { def n = println ("A") }
trait C { override def n = println ("C") }
trait D { override def n = println ("D") }
val x = (new A with C with D).n
```

Redéfinition et composition *mixin*

Que se passe-t-il dans le programme suivant ?

```
class A { def n = println ("A") }
trait C { override def n = println ("C") }
trait D { override def n = println ("D") }
val x = (new A with C with D).n
```

SCALA rejette le programme, on ne sait pas quelle est la méthode n de quelle classe on est en train de redéfinir.

Redéfinition et composition *mixin*

trait C extends A signifie que la classe mère de C est une sous-classe de A :

```
class A { def n = println ("A") }
trait C extends A { override def n = println ("C") }
trait D extends A { override def n = println ("D") }
val x = (new A with C with D).n
```

Qu'affiche ce programme ?

Redéfinition et composition *mixin*

trait C extends A signifie que la classe mère de C est une sous-classe de A :

```
class A { def n = println ("A") }
trait C extends A { override def n = println ("C") }
trait D extends A { override def n = println ("D") }
val x = (new A with C with D).n
```

Qu'affiche ce programme ? "D" car l'ordre suivant lequel la composition est effectuée compte !

Redéfinition et composition *mixin*

Que se passe-t-il si on fait un appel à une méthode abstraite dans un trait ?

```
class A { abstract def n : Unit }
trait C extends A { override def n = { super.n ; println ("C") } }
trait D extends A { override def n = { super.n ; println ("D") } }
val x = (new A with C with D).n
```

Redéfinition et composition *mixin*

Que se passe-t-il si on fait un appel à une méthode abstraite dans un trait ?

```
class A { abstract def n : Unit }
trait C extends A { override def n = { super.n ; println ("C") } }
trait D extends A { override def n = { super.n ; println ("D") } }
val x = (new A with C with D).n
```

SCALA rejette ce programme : comment s'assurer que la méthode super est bien définie ?

Redéfinition et composition *mixin*

On peut indiquer que l'on **promet** que le trait sera composé avec une sous-classe de A qui implémente n.

```
class A { abstract def n : Unit }
trait C extends A {
   abstract override def n = { super.n; println ("C") }
}
trait D extends A {
   abstract override def n = { super.n; println ("D") }
}
class E extends A { override def n = println ("E") }
val x = (new E with C with D).n
```

Le mécanisme d'héritage entre traits

Un trait peut aussi hériter d'un autre trait :

```
trait A { def n = ... }
trait B extends A { def m = ... }
```

Il s'agit juste ici d'étendre des modifications par d'autres modifications.

Le Graal de la modularité

Exercice :
En quoi les traits apportent-ils une solution au problème d'extensibilité fonctionnelle évoqué plus tôt sur les visiteurs ?

Plan de la partie *"Construire des objets"*

Retour sur l'exemple

Comment utiliser les traits de SCALA pour simuler l'héritage multiple ?

```
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 {}
```

Retour sur l'exemple

```
trait Doubling extends IntQueue {
  abstract override def put (x: Int) { super.put (2 * x) }
}

trait Incrementing extends IntQueue {
  abstract override def put (x: Int) { super.put (x + 1) }
}

class MyQueue extends BasicIntQueue with Doubling with Incrementing

object test {
  def main (args: Array[String]) {
    val q = new MyQueue
    q.put (1)
    println (q.get)
  }
}
```

Qu'affiche ce programme ?

Retour sur l'exemple

```
trait Doubling extends IntQueue {
  abstract override def put (x: Int) { super.put (2 * x) }
}

trait Incrementing extends IntQueue {
  abstract override def put (x: Int) { super.put (x + 1) }
}

class MyQueue extends BasicIntQueue with Doubling with Incrementing

object test {
  def main (args: Array[String]) {
    val q = new MyQueue
    q.put (1)
    println (q.get)
  }
}
```

Qu'affiche ce programme ? 4 !

Une pile de modifications

La composition mixin permet d'appliquer une pile de modification à une classe. Par construction, il n'y a qu'un unique chemin dans la hiérarchie de classe entre la classe obtenue et la classe initiale, ce qui élimine le problème du diamant par construction. Ce mécanisme d'héritage s'appelle la linéarisation.

Linéarisation

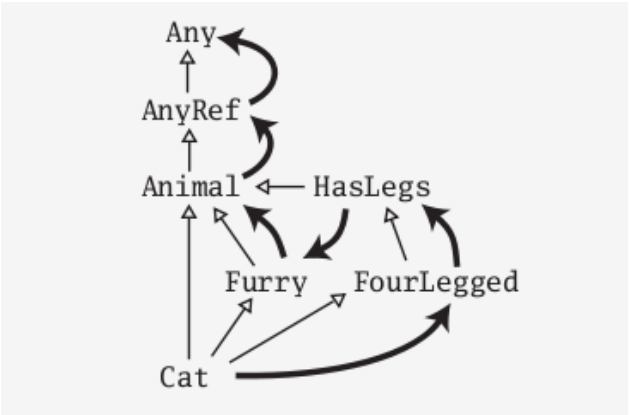
Exemple tiré de *Programming in Scala* de Martin Odersky

```
class Animal
trait Furry extends Animal
trait HasLegs extends Animal
trait FourLegged extends HasLegs
class Cat extends Animal with Furry with FourLegged
```

Quelles hiérarchies de classe sont construites par ces compositions mixins ?

Linéarisation

Exemple tiré de *Programming in Scala* de Martin Odersky



- Animal → AnyRef → Any
- Furry → Animal → AnyRef → Any
- Cat → FourLegged → HasLegs → Furry → Animal → AnyRef → Any

Sous-typage nominal pour les traits ?

Par défaut, SCALA offre un sous-typage **nominal** qui s'appuie sur les relations de sous-typage entre instances de classe définies par les hiérarchies. Pour étendre le sous-typage aux classes construites avec des traits, il faut pouvoir écrire des types pour les représenter.

Par exemple, si on a un trait A :

```
trait A { def x : Int }
```

On peut parler des types des instances de classe construite à l'aide de A ainsi :

```
def f (a : A) { println (a.x) }
```

Et si on se donne un autre trait de la forme trait B { def x : Int }, est-ce que f est compatible avec les classes composées avec B ?

Sous-typage structurel sur les traits

Si on veut élargir le domaine de la fonction f à tous les objets qui possèdent une méthode x alors on peut utiliser un type structurel :

```
def f (a : { def x : Int }) { println (a.x) }
```

que l'on pourrait aussi écrire :

```
type has_x = { def x : Int }
def f (a : has_x) { println (a.x) }
```

Ces syntaxes sont des sucres syntaxiques pour :

```
def f (a : AnyRef { def x : Int }) { println (a.x) }
```

Des domaines de fonctions très précis

Les types de la forme A { ... } permettent de donner des spécifications aux domaines des fonctions en mélangeant les notions de sous-typage nominal et structurel. Par exemple, on peut parler de toutes les sous-classes de A qui possèdent une méthode x de la façon suivante :

```
class A {
  def f (x : Int) = x + 1
}
class B extends A { def x : Int = 42 }
class C extends A { def y : Int = 42 }
object test {
  def g (a : A { def x : Int }) = a.f (a.x)

  g (new B)
  // g (new C) // Rejected
  g (new A { def x = 42 })
}
```

Traits paramétrés par des types

Tout comme les classes, les traits peuvent être paramétrés par des types.

La syntaxe SCALA est de la forme :

```
trait A[T] {
  // ...
}
```

On peut imposer des **contraintes de sous-typage** sur ces paramètres :

```
trait B
trait A[T <: B] {}
trait C[T <: B { def x : Int }] {}
```

Exemple : le trait Ordered[T]

```
trait Ordered[A] {
  def compare(that: A): Int

  def < (that: A): Boolean = (this compare that) < 0
  def > (that: A): Boolean = (this compare that) > 0
  def ≤ (that: A): Boolean = (this compare that) ≤ 0
  def ≥ (that: A): Boolean = (this compare that) ≥ 0
  def compareTo (that: A): Int = compare(that)
}
```

En quoi ce trait est-il utile ?

Exemple de trait paramétré avec borne récursive

Voici un drôle de trait :

```
trait A[S <: A[S]] {
  // ...
}
```

À quoi peut-il servir ?

Traits paramétrés par un paramètre modélisant le type de *Self*

```
trait A[S <: A[S]] {
  def self : S
  def x : Int = 42
}

trait B[S <: B[S]] extends A[S] {
  def self : S
  def y : Int = 21
}

class C extends B[C] {
  def self = this
}
```

Dans quelle situation la connaissance du type le plus précis de l'objet peut elle être intéressante ?

Chaînage de méthode

Voici deux classes écrites en objet « classique » :

```
class A {
  var x = 0
  def incr = { x += 1; this }
}
class B extends A {
  def decr = { x -= 1; this }
}
object test {
  val x = new B
  x.incr.decr
  // Error : error : value decr is not a member of A
}
```

Le type le plus précis de l'instance x a été oublié lors de l'appel à la méthode incr. Le compilateur rejette ce programme pourtant licite !

Solution au problème de chaînage de méthodes

```
trait A[S <: A[S]] {
  var x : Int = 0
  def self : S
  def incr : S = { x += 1 ; self }
}
trait B[S <: B[S]] extends A[S] {
  def decr : S = { x -= 1 ; self }
}
class C extends B[C] {
  def self = this
}
object test {
  val x = new C
  x.incr.decr
}
```

Extension de l'idée

On peut faire des hypothèses supplémentaires sur S, c'est-à-dire sur la façon dont la classe A sera étendue :

```
// All subclasses of A must eventually define a method y.
trait A[S <: A[S] { def y : Int } ] {
  def self : S
  def x : Int = self.y
}

trait B[S <: B[S]] extends A[S] {
  def self : S
  def y : Int = 21
}

class C extends B[C] {
  def self = this
}
```

Traits paramétrés et type *Self*

L'utilisation d'un paramètre de type représentant le type le plus précis de this est très pratique pour exprimer des contraintes sur les futurs héritages et en tirer parti dans la classe mère.

Pour en faciliter l'usage, une syntaxe spéciale a été introduite en Scala.

L'exemple précédent s'écrit de la façon suivante :

```
trait A { self : A { def y : Int } =>
  def x : Int = y
}

class B extends A { def y : Int = 42 }
```

Définition modulaire d'une structure d'arbre

Un arbre est composé d'une étiquette et d'un ensemble de sous-arbre.

On suppose de plus que chaque étiquette a une arité, c'est-à-dire un entier naturel. Pour être bien formé, un arbre doit respecter l'arité de ses étiquettes : l'ensemble de sous-arbre associé à une étiquette doit avoir pour cardinal l'arité de l'étiquette.

À l'aide de traits, comment obtenir une définition modulaire d'arbre qui permet de faire varier indépendamment la représentation de l'étiquette et la représentation de l'ensemble des sous-arbres ?

Une définition modulaire des arbres

```
trait Tree {
  type Label
  def arity : Int
  def label : Label

  type Children
  def children : Children
  def childrenCount : Int
  def childrenGet (x: Int) : Tree

  def labelsOfPath (path : List[Int]) : List[Label] =
    path match {
      case Nil =>
        List (label)
      case x :: xs => {
        label :: childrenGet (x).labelsOfPath (xs).asInstanceOf[List[Label]]
      }
    }
  def toString : String
}
```

Notez que les types Label et Children sont abstraits.

Une implémentation pour les étiquettes

Voici une implémentation particulière des étiquettes par des entiers. On choisit un nouveau numéro pour chaque nouvelle instance. L'arité de tous les nœuds est fixée à 2.

```
var count = 0
trait AutomaticIntBinaryLabel { self: Tree =>
  type Label = Int
  val label = { count += 1; count }
  def arity = 2
  override def toString =
    label.toString +
    (if (childrenCount == 0) ""
     else "(" + children.toString + ")")
}
```

Pour pouvoir afficher l'arbre, il suffit de savoir afficher une étiquette. En fixant le type de *self* à Tree, on peut utiliser les méthodes travaillant sur les sous-arbres comme childrenCount par exemple.

Une implémentation d'une forêt d'arbres

Voici une implémentation de forêt à l'aide de liste.

```
trait ListChildren { self: Tree =>
  assert (arity == childrenCount)
  type Children = List[Tree]
  def childrenCount = children.length
  def childrenGet (x: Int) = children (x)
  def children : Children
}
```

Notez comme cette définition fait référence à la méthode arity définie dans la classe précédente. Encore une fois, cette référence est permise puisque l'on promet que la classe finale obtenue après composition *mixin* sera de type Tree, comme l'indique l'annotation de type de *self*.

Classe finale par composition mixin

Les deux définitions précédentes sont **mutuellement récursives**. On peut néanmoins les composer de façon à obtenir le point fixe de leurs définitions :

```
class TreeImpl (val children : List[TreeImpl]) extends Tree
with AutomaticIntBinaryLabel
with ListChildren
```

Grâce à la composition mixin, on peut définir des fragments de classes avec une granularité très fine car ils peuvent faire références les uns aux autres d'une façon très libre. Ce mécanisme s'appuie sur les annotations de type *self* qui déclarent *a priori* la forme de la classe finale.

Le trait, Saint-Graal de la modularité

Soit une classe C contenant des attributs, des méthodes, des types...

On peut partitionner C *d'une façon arbitraire* sous la forme de *traits* $T_1 \dots T_n$ dont le type de *self* est C et obtenir C par composition mixin de ces traits.

Cela signifie que l'on peut à tout moment décider *d'abstraire une partie du code d'une classe* pour le faire varier *a posteriori*.

Le trait, Saint-Graal de la modularité

Le trait, une spécificité de Scala ?

- Ruby a aussi une composition mixin mais pas de système de type. (Perl 6 aussi.)
- OCaml permet d'écrire des modules mutuellement récursifs de première classe dont l'expressivité est similaire aux traits de Scala et dont le bon usage est vérifié par le typeur. Par contre, ils sont plus lourds à utiliser syntaxiquement.

Retour sur le visiteur modulaire

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** ?

Existence d'un visiteur sur une hiérarchie de classe

```
trait ShapesBase {

  abstract class Shape {
    def accept (v: ShapeVisitor)
  }

  type ShapeVisitor <: AbstractShapeVisitor
  trait AbstractShapeVisitor {
    self: ShapeVisitor =>
    def visit (s : Shape) = s.accept (this)
  }
}
```

On introduit **un type virtuel** qui va être celui d'un visiteur travaillant sur une hiérarchie dont la base est Shape. Le trait AbstractShapeVisitor représente l'interface que doit implémenter un visiteur concret travaillant sur cette hiérarchie (qui n'a pas de sous-classe pour l'instant).

Première extension de données

```
trait ShapesLibExtension1 extends ShapesBase {
  class Circle extends Shape {
    def accept (v: ShapeVisitor) = v.visit (this)
  }

  type ShapeVisitor <: AbstractShapeVisitor
  trait AbstractShapeVisitor extends super.AbstractShapeVisitor {
    self: ShapeVisitor =>
    def visit (s : Circle) : Unit
  }
}
```

On raffine ici le type des visiteurs sur cette nouvelle hiérarchie : un visiteur doit maintenant traiter un cas supplémentaire, celui des cercles.

Première extension de fonctionnalité

```
trait ShapesLibExtension1_2 extends ShapesLibExtension1 {
  trait DescriptionVisitor extends super.AbstractShapeVisitor {
    self: ShapeVisitor =>
    def visit (s: Circle) = println ("Here is a circle!")
  }
}
```

Pour implémenter un visiteur qui travaille sur cette hiérarchie, il suffit de créer une sous-classe concrète de la classe AbstractShapeVisitor. Pour pouvoir l'instancier, il faut fixer le type virtuel :

```
object ShapesLibVersion1_2 extends ShapesLibExtension1_2 {
  type ShapeVisitor = AbstractShapeVisitor
  class DescriptionVisitor extends super.DescriptionVisitor
}
```

Seconde extension des données

```
trait ShapesLibExtension2 extends ShapesBase {
  class Square extends Shape {
    def accept (v: ShapeVisitor) = v.visit (this)
  }

  type ShapeVisitor <: AbstractShapeVisitor
  trait AbstractShapeVisitor extends super.AbstractShapeVisitor {
    self: ShapeVisitor =>
    def visit (s : Square) : Unit
  }
}
```

Remarquez que cette extension-ci est **indépendante** de la première.

Seconde extension de fonctionnalités

```
trait ShapesLibExtension2_2 extends ShapesLibExtension2 {
  trait DescriptionVisitor extends super.AbstractShapeVisitor {
    self: ShapeVisitor =>
    def visit (s: Square) = println ("Here is a square !")
  }
}

object ShapesLibVersion2_2 extends ShapesLibExtension2_2 {
  type ShapeVisitor = AbstractShapeVisitor
  class DescriptionVisitor extends super.DescriptionVisitor
}
```

Par le même schéma que précédemment, on instancie un visiteur qui travaille sur une hiérarchie qui contient une classe Square

Question centrale

Comment produire une hiérarchie possédant ces deux types de données et un visiteur correspondant à la composition des deux visiteurs spécifiques à ces deux types ?

Une première solution

En utilisant la composition *mixin*, on fait l'**union** les deux hiérarchies.

```
trait ShapesLibExtension12
  extends ShapesLibExtension1
  with ShapesLibExtension2 {

  type ShapeVisitor <: AbstractShapeVisitor

  trait AbstractShapeVisitor
  extends super[ShapesLibExtension1].AbstractShapeVisitor
  with super[ShapesLibExtension2].AbstractShapeVisitor {
    self: ShapeVisitor =>
  }

  trait DescriptionVisitor
  extends super[ShapesLibExtension2].AbstractShapeVisitor
  with super[ShapesLibExtension1].AbstractShapeVisitor {
      self: ShapeVisitor =>
      def visit (s: Square) = println ("Here is a square!")
      def visit (s: Circle) = println ("Here is a circle!")
  }
}
```

Première solution : concrétisation

```
object ShapesLibVersion12 extends ShapesLibExtension12 {
  type ShapeVisitor = AbstractShapeVisitor
  class DescriptionVisitor extends ShapeVisitor
  with super.DescriptionVisitor {
    self: ShapeVisitor =>
  }
}

object testShapesLibVersion12 {
  def main (args: Array[String]) {
    import ShapesLibVersion12._
    (new DescriptionVisitor).visit (new Square)
    (new DescriptionVisitor).visit (new Circle)
  }
}
```

Seconde solution

On peut faire mieux en composant aussi les visiteurs :

```
trait ShapesLibExtension12_bis
  extends ShapesLibExtension1_2
  with ShapesLibExtension2_2 {

  type ShapeVisitor <: AbstractShapeVisitor

  trait AbstractShapeVisitor
  extends super[ShapesLibExtension1_2].AbstractShapeVisitor
  with super[ShapesLibExtension2_2].AbstractShapeVisitor {
    self: ShapeVisitor =>
  }

  trait DescriptionVisitor
  extends super[ShapesLibExtension1_2].DescriptionVisitor
  with super[ShapesLibExtension2_2].DescriptionVisitor {
      self: ShapeVisitor =>
  }
}
```

Pourquoi est-ce que cela compose bien le comportement des visiteurs ?

Seconde solution : concrétisation

```
object ShapesLibVersion12_bis
extends ShapesLibExtension12_bis {
   type ShapeVisitor = AbstractShapeVisitor
   class DescriptionVisitor extends ShapeVisitor
   with super.DescriptionVisitor {
      self: ShapeVisitor =>
   }
}


object testShapesLibVersion12_bis {
   def main (args: Array[String]) {
      import ShapesLibVersion12_bis._
      (new DescriptionVisitor).visit (new Square)
      (new DescriptionVisitor).visit (new Circle)
   }
}
```