JAVA &
LE "PATTERN MATCHING"

4 Novembre 2015

Toulouse JUG

Didier Plaindoux (@dplaindoux)

Freelance

Computer scientist

PATTERN MATCHING

Principe reposant sur la vérification de la présence de constituants d'un motif par un programme informatique

ILLUSTRATION: EXPRESSION (SIMPLE)

Expr = ⌜Natural⌝ | Expr ⊕ Expr

⟦_⟧: Expr ⟶ Natural
⟦⌜n⌝⟧= n
⟦a ⊕ b⟧= ⟦a⟧+⟦b⟧

EXPRESSION EN OCAML

```
type expr =
  | Nat of int
  | Add of expr * expr

let rec evaluate : expr → int = function
  | Nat n      → n
  | Add (l,r) → (evaluate l) + (evaluate r)
```

EXPRESSION EN HASKELL

```
data Expr =
   | Nat Integer
   | Add Expr Expr

evaluate :: Expr → Integer

evaluate (Nat i)   = i
evaluate (Add l r) = (evaluate l) + (evaluate r)
```

EXPRESSION EN SCALA

```
sealed trait Expr
case class Nat(i:Int) extends Expr
case class Add(l1:Expr, l2:Expr) extends Expr

def evaluate(e:Expr):Int = e match {
  case Nat(n)   ⇒ n
  case Add(l,r) ⇒ evaluate(l) + evaluate(r)
}
```

ET EN JAVA ?

Pas de construction dédiée dans le langage.

EXPRESSION EN JAVA 1/2

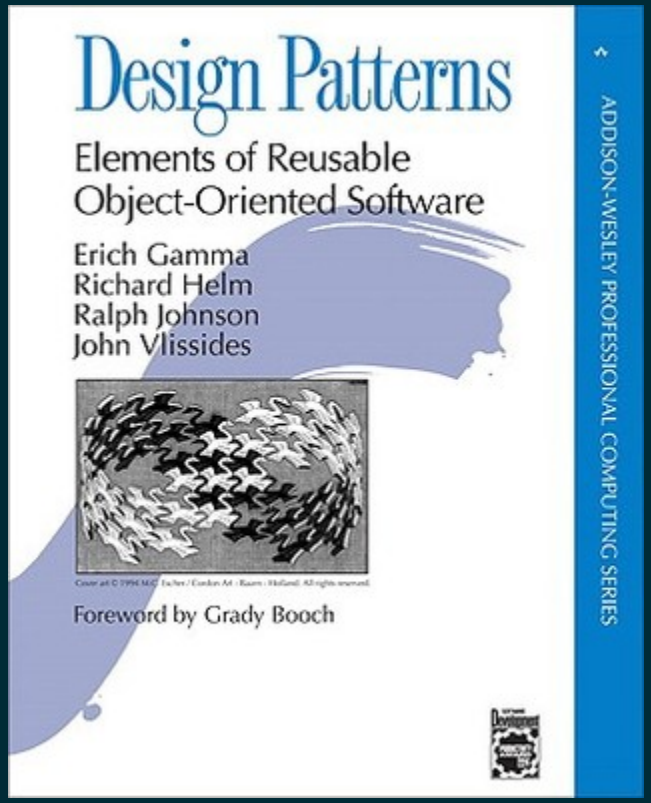
```
public interface Expr {}

public class Nat implements Expr {
  private final int val;
  public Nat(int val) { this.val = val; }
}

public class Add implements Expr {
  private final Expr left,right;
  public Add(Expr l,Expr r) { this.left = l; this.right = r; }
}
```

EXPRESSION EN JAVA 2/2

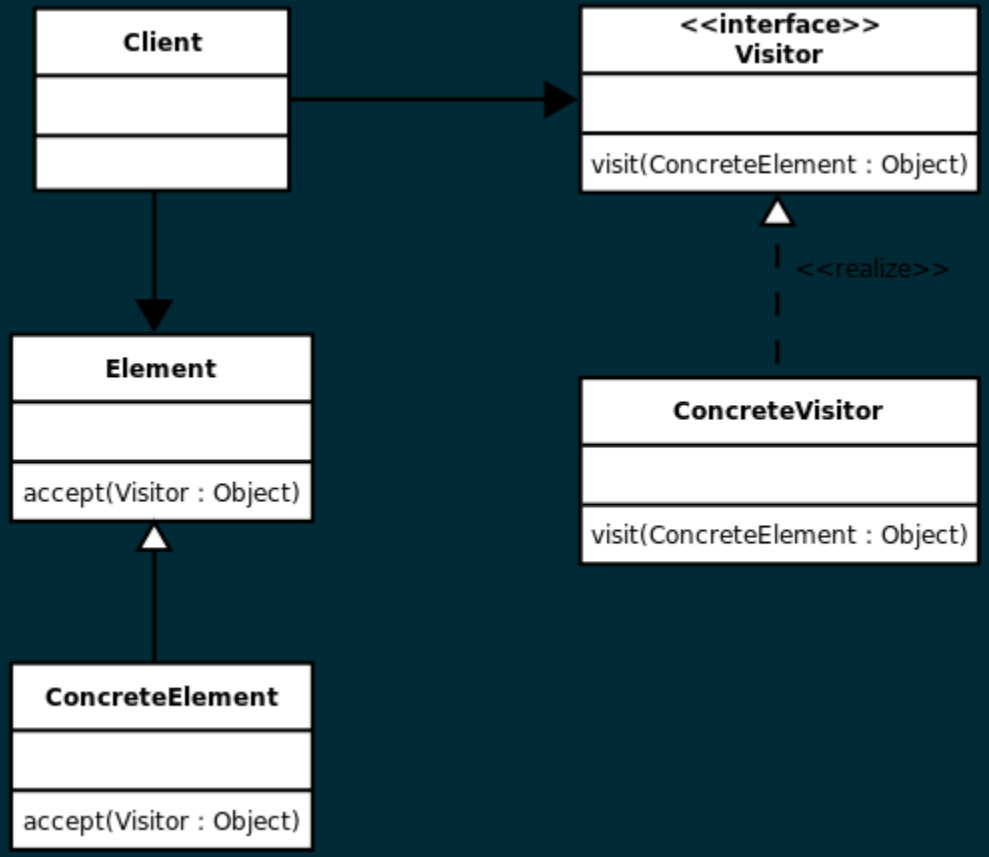
```
public class Evaluator {
  public static int eval(Expr e) {
    if (e instanceof Nat) {
      return (Nat.class.cast(e).val;
    } if (e instanceof Add) {
      final Add add = Add.class.cast(e);
      return eval(add.left) + eval(add.right);
    } else {
      throw new IllegalArgumentException();
    }
  }
}
```

BAD! QUID DU TYPAGE ?

"GANG OF FOUR"


Design Patterns
Elements of Reusable
Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

LE VISITEUR


Client
<<interface>>
Visitor
visit(ConcreteElement : Object)
<<realize>>
Element
accept(Visitor : Object)
ConcreteVisitor
visit(ConcreteElement : Object)
ConcreteElement
accept(Visitor : Object)

EXPRESSION EN JAVA & LE VISITEUR 1/3

```
public interface ExprVisitor<T> {
  T visit(Nat val);
  T visit(Add add);
}

public interface Expr {
  <T> T visit(ExprVisitor<T> visitor)
}
```

EXPRESSION EN JAVA & LE VISITEUR 2/3

```
public class Nat implements Expr {
  public final int val;
  public Nat(int val) { this.val = val; }
  public <T> T visit(ExprVisitor<T> visitor) {
    return visitor.visit(this);
  }
}

public class Add implements Expr {
  public final Expr left,return;
  public Add(Expr l,Expr r) { this.left = l; this.right = r; }
  public <T> T visit(ExprVisitor<T> visitor) {
    return visitor.visit(this);
  }
}
```

EXPRESSION EN JAVA & LE VISITEUR 3/3

```
public class Evaluator implements ExprVisitor<Integer> {
  public Integer visit(Nat expr) {
    return expr.val;
  }
  public Integer visit(Add expr) {
    return expr.right.visit(this) + expr.left.visit(this);
  }
}
```

UGLY? HUMM ...

APPROCHE FONCTIONNELLE

"Structural Pattern Matching en Java" par Rúnar Óli.

APPROCHE FONCTIONNELLE & JAVA 1/3

```
public class Either<A,B> {
  private A left; private B right;
  private Either(A a, B b) { left = a; right = b; }
  public static <A,B> Either<A,B> left(A a) {
    return new Either<>(a,null);
  }
  public static <A,B> Either<A,B> right(B b) {
    return new Either<>(null,b);
  }

  public <R> R fold(Function<A,R> ifLeft, Function<B,R> ifRight) {
    return Optional.ofNullable(left).map(_ → ifLeft.apply(left))
                   .orElse(() → ifRight.apply(right));
  } // ⇒ Catamorphisme
}
```

"Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire" par Erik Meijer

APPROCHE FONCTIONNELLE & JAVA 2/3

```
public interface Expr { Either<Nat,Add> toEither(); }

public class Nat implements Expr {
  public final int val;
  public Nat(int val) { this.val = val; }
  public Either<Nat,Add> toEither() { return Either.left(this); }
}

public class Add implements Expr {
  public final Expr left,right;
  public Add(Expr l,Expr r) { this.left = l; this.right = r; }
  public Either<Nat,Add> toEither() { return Either.right(this); }
}
```

APPROCHE FONCTIONNELLE & JAVA 3/3

```
public class Evaluator {
  public static int evalNat(Nat v) {
    return v.val;
  }

  public static int evalAdd(Add a) {
    return eval(a.left.toEither()) + eval(a.right.toEither());
  }

  public static int eval(Either<Nat,Add> e) {
    return e.fold(Evaluator::evalNat, Evaluator::evalAdd);
  }
}
```

GOOD? OUI MAIS EITHER<...,EITHER<...,...>>

DÉFINITION D'UN DSL INTERNE

"Internal DSLs are a particular form of API in a host general purpose language" Martin Fowler

UNE PREMIÈRE PROPOSITION

"Towards Pattern Matching in Java" par François Sarradin

```
static PatternMatching pm = new PatternMatching(
  inCaseOf(Var.class, x → x.val),
  inCaseOf(Add.class, x → pm(x.left)+pm(x.right))
);

static int pm(Expr expr) { return pm.matchFor(expr); }
```

Typage dynamique et casts ... mais cela reste une étude

LE PROJET DERIVE4J

"Java 8 annotation processor for deriving pattern-matching"
par J.B. Giraudeau

Généralisation de l'approche fonctionnelle
Répose sur la JSR-269 pour tout ce qui est "Boiler Plate"

Extension au Lens pour l'évolution des objets

EXPRESSION AVEC DERIVE4J

```
@Data
public abstract class Expr {
  interface Cases<R> {
    R Nat(int value);
    R Add(Expr left, Expr right);
  }
}

public class Evaluator {
  static int eval(Expr expr) { return eval.apply(expr); }

  static Function<Expr,Integer> eval =
    Exprs.cases().Nat((v)   -> v)
                 .Add((l,r) -> eval(l) + eval(r));
}
```

LES "EXTRACTOR OBJECTS" DANS SCALA 1/2

En Scala il est possible de décorréler
"pattern" et "case classes"

LES "EXTRACTOR OBJECTS" DANS SCALA 2/2

```
object Zero {
  def unapply(n:Int):Option[Int] = if (n==0) then Some(0) else None
}

object Succ {
  def unapply(n:Int):Option[Int] = if (n>0) then Some(n-1) else None
}

object Peano extends App {
  23 match { case Zero(_) => true case Succ(_) => false }
}
```

LA LIBRAIRIE SUITCASE

Un DSL au service du Pattern Matching

PRINCIPES ET CONSTRUCTIONS

```
Matcher<I,O> matcher = Matcher.create();

// Reconnaissance pure i.e. WithoutCapture
matcher.caseOf( (? ≤ I) → () ).then( () → O );
matcher.caseOf( (? ≤ I) → () ).when( () → bool ).then( () → O );

// Reconnaissance par décomposition structurelle i.e WithCapture<C>
matcher.caseOf( (? ≤ I) → C ).then( C → O );
matcher.caseOf( (? ≤ I) → C ).when( C → bool ).then( C → O );
```

Le Matcher est un objet donc ... extensible etc.

PATTERN MATCHING PAR TYPAGE DYNAMIQUE

```
Matcher<Expr,Boolean> isNat = Matcher.create();

isNat.caseOf(typeOf(Nat.class)) // Nat ⟶ ()
     .then(() -> true);         // () ⟶ Boolean

isNat.caseOf(Add.class)         // Add ⟶ ()
     .then(false);              // () ⟶ Boolean
```

PATTERN MATCHING PAR CAPTURE

```
Matcher<Expr,Boolean> isNat = Matcher.create();
isNat.caseOf(var())                  // Expr ⟶ Expr
     .then(e ⟶ e instanceof Nat); // Expr ⟶ Boolean
```

```
Matcher<Expr,Boolean> isNatZero = Matcher.create();

isNatZero.caseOf(var(Nat.class))   // Nat ⟶ Nat
         .then(e ⟶ e.val == 0);   // Nat ⟶ Boolean

isNatZero.caseOf(__)               // Add ⟶ ()
         .then(false);             // () ⟶ Boolean
```

EXPRESSION EN JAVA & SUITCASE V1

```
Matcher<Expr,Integer> eval = Matcher.create();

eval.caseOf(var(Nat.class))
    .then(e -> e.val);

eval.caseOf(var(Add.class))
    .then(e -> eval.match(e.left) + eval.match(e.right));
```

BAD AGAIN?

LES CLASSES CASE

```
public interface Case<I,C> {
  Optional<C> unapply(I t);
}
```

Deux sous-types WithCapture<?> et WithoutCapture

NAT(_) CLASSE CASE

```
static TypeCase1<Expr, Nat, Integer> Nat =
  new TypeCase1<>(Nat.class, (e -> e.val));

static WithoutCapture<Expr> Nat(WithoutCapture<Integer> aCase) {
  return Nat.of(aCase);
}

static <C> WithCapture<Expr, C> Nat(WithCapture<Integer, C> aCase) {
  return Nat.of(aCase);
}
```

Idem pour le **Add(_,_)**

EXPRESSION EN JAVA & SUITCASE V2

```
Matcher<Expr,Integer> eval = Matcher.create();

eval.caseOf(Nat(var()))        // Nat ⟶ Integer
    .then(e ⟶ e);

eval.caseOf(Add(var(),var())) // Add ⟶ Pair<Expr,Expr>
    .then(p ⟶ eval.match(p._1) + eval.match(p._2));
```

Reconnaissance par décomposition structurelle

⇒ en profondeur

NOT SO ... BAD!

RETOUR SUR LE TYPAGE 1/2

Le type entrant et le type sortant sont vérifiés naturellement
Le type des données capturées est synthétisé

Repose sur un typage fort

RETOUR SUR LE TYPAGE 2/2

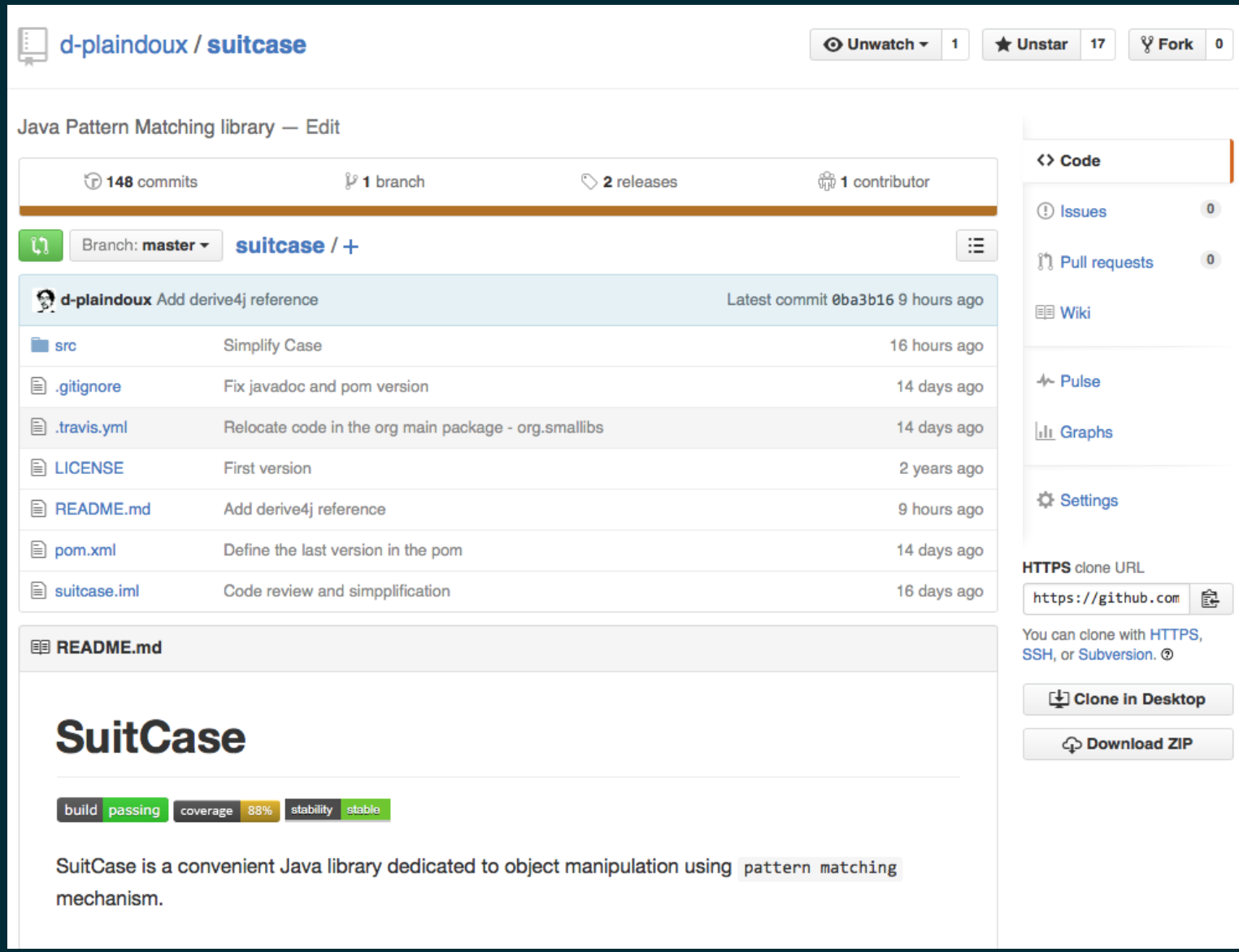
```
Matcher<Expr,Integer> eval = Matcher.create();

eval.caseOf(Nat(var())).then((String e) → 0);
```

"Incompatible parameter types in lambda expression"
Erreur notifiée dans les IDEs lors de l'écriture

CONCLUSION & PERSPECTIVE

Librairie simple, ouverte et expressive

Refonte complète en cours avec la version 0.3
Recours à la JSR-269 pour tout ce qui est "Boiler Plate" ?


d-plaindoux / suitcase
Unwatch 1
Unstar 17
Fork 0
Java Pattern Matching library — Edit
148 commits
1 branch
2 releases
1 contributor
Branch: master
suitcase / +
d-plaindoux Add derive4j reference
Latest commit 0ba3b16 9 hours ago
src Simplify Case 16 hours ago
.gitignore Fix javadoc and pom version 14 days ago
.travis.yml Relocate code in the org main package - org.smallibs 14 days ago
LICENSE First version 2 years ago
README.md Add derive4j reference 9 hours ago
pom.xml Define the last version in the pom 14 days ago
suitcase.iml Code review and simpplification 16 days ago
README.md
SuitCase
build passing
coverage 88%
stability stable
SuitCase is a convenient Java library dedicated to object manipulation using pattern matching mechanism.
Code
Issues 0
Pull requests 0
Wiki
Pulse
Graphs
Settings
HTTPS clone URL
https://github.com
You can clone with HTTPS, SSH, or Subversion.
Clone in Desktop
Download ZIP

QUESTIONS ?