Automatic refactorings for Scala programs

Taming multi-paradigm code

Ilya Sergey       Dave Clarke       Alexander Podkhalyuzin
15 April 2010

Scala Days 2010
Outline

• Why implementing refactorings for Scala is challenging?
• What refactorings have we implemented for now?
• What kind of Scala-specific refactorings could be useful?
What are refactorings good for for

- Cleaning up code
- Changing internal code structure and design
- Improving understandability
- Providing better modularization
What are refactorings

not used for

- Adding new functionality
- Fixing bugs
- Changing overall program behaviour
Case study: extract method refactoring in Java

Input and output local values of the code fragment are captured
Catalog of useful refactorings in object-oriented frameworks

- Add Parameter
- Change Bidirectional Association to Unidirectional
- Change Reference to Value
- Change Unidirectional Association to Bidirectional
- Change Value to Reference
- Collapse Hierarchy
- Consolidate Conditional Expression
- Consolidate Duplicate Conditional Fragments
- Convert Dynamic to Static Construction
- Convert Static to Dynamic Construction
- Decompose Conditional
- Duplicate Observed Data
- Eliminate Inter-Entity Bean Communication
- Encapsulate Collection
- Encapsulate Downcast
- Encapsulate Field
- Extract Class
- Extract Interface
- Extract Method
- Extract Package
- Extract Subclass
- Extract Superclass
- Form Template Method
- Hide Delegate Method
- Hide presentation tier-specific details from the business tier
- Inline Class
- Inline Method
- Inline Temp
- Introduce A Controller
- Introduce Assertion
- Remove Middle Man
- Remove Parameter
- Remove Setting Method
- Rename Method
- Replace Array with Object
- Introduce Business Delegate
- Introduce Explaining Variable
- Introduce Foreign Method
- Introduce Local Extension
- Merge Session Beans
- Move Business Logic to Session
- Move Class
- Move Field
- Move Method
- Parameterize Method
- Preserve Whole Object
- Pull Up Constructor Body
- Pull Up Field
- Pull Up Method
- Push Down Field
- Push Down Method
- Reduce Scope of Variable
- Refactor Architecture by Tiers
- Remove Assignments to Parameters
- Remove Control Flag
- Remove Double Negative
- Remove Middle Man
- Remove Parameter
- Remove Setting Method
- Replace Assignment with Initialization
- Replace Conditional with Polymorphism
- Replace Conditional with Visitor
- Replace Constructor with Factory Method
- Replace Data Value with Object
- Replace Delegation with Inheritance
- Replace Error Code with Exception
- Replace Exception with Test
- Replace Inheritance with Delegation
- Replace Iteration with Recursion
- Replace Magic Number with Symbolic Constant
- Replace Method with Method Object
Not all the refactorings should be automatized
Catalog of **useful** refactorings in object-oriented frameworks

- **Add Parameter**
- Change Bidirectional Association to Unidirectional
- Change Reference to Value
- Change Unidirectional Association to Bidirectional
- Change Value to Reference
- Collapse Hierarchy
- Consolidate Conditional Expression
- Consolidate Duplicate Conditional Fragments
- Convert Dynamic to Static Construction
- Convert Static to Dynamic Construction
- Decompose Conditional
- Duplicate Observed Data
- Eliminate Inter-Entity Bean Communication
- Encapsulate Collection
- Encapsulate Downcast
- Encapsulate Field
- Extract Class
- Extract Interface
- Extract Method
- Extract Package
- Extract Subclass
- Extract Superclass
- Form Template Method
- Hide Delegate
- Hide Method
- Hide presentation tier-specific details from the business tier
- Inline Class
- Inline Method
- Inline Temp
- Introduce A Controller
- Introduce Assertion
- Remove Middle Man
- Remove Parameter
- Remove Setting Method
- Rename Method
- Replace Array with Object
- Introduce Business Delegate
- Introduce Explaining Variable
- Introduce Foreign Method
- Introduce Local Extension
- Merge Session Beans
- Move Business Logic to Session
- Move Class
- Move Field
- Move Method
- Parameterize Method
- Preserve Whole Object
- Pull Up Constructor Body
- Pull Up Field
- Pull Up Method
- Push Down Field
- Push Down Method
- Reduce Scope of Variable
- Refactor Architecture by Tiers
- Remove Assignments to Parameters
- Remove Control Flag
- Remove Double Negative
- Remove Middle Man
- Remove Parameter
- Remove Setting Method
- Replace Assignment with Initialization
- Replace Conditional with Polymorphism
- Replace Conditional with Visitor
- Replace Constructor with Factory Method
- Replace Data Value with Object
- Replace Delegation with Inheritance
- Replace Error Code with Exception
- Replace Exception with Test
- Replace Inheritance with Delegation
- Replace Iteration with Recursion
- Replace Magic Number with Symbolic Constant
- Replace Method with Method Object
Now comes *Scala*

- Fusion of functional and object-oriented paradigms
- First-class and higher-order functions
- Dependent types / imports
- Custom implicit conversions
- Flexible scoping rules

Classical refactorings for Java must be reconsidered
Scala-specific refactorings

• Adapt Java programming style for Scala
• Abstract over structure of types
• Take advantage of Scala’s functional programming features
Refactorings we have implemented so far

• Rename variable, method, class etc.
• Move class
• Safe delete class
• Optimize imports
• Introduce / inline variable
• Extract method
Optimize imports

• **Goal**: Remove unused import statements

• **Challenge**: Implicit conversions demand additional checks
Meaningful import of an implicit conversion function

```scala
class A
class B
object MyConversions {
  implicit def a2b(a: A): B = new B
}
def testB(b: B): Any = {/* some code */}
import MyConversions._
val a = testB(new A)
```

In fact this is `a2b(new A)`

Cannot be removed!
Demo

Optimize imports
Extract variable from an expression

• New coding style: write-introduce

• No more tedious preamble
  • final Map<String, Object> map = new HashMap<...>();

• Write an expression and introduce it as a variable: the necessary type will be inferred
Introduce variable by expression

• **Goal:** Extract an expression to a new variable or parameter of a function

• **Challenge:** Infer the type of expression
Scala’s type inference allows to omit explicit type declarations!

Is it always good?

Sometimes a type can provide essential information about an expression
Some unannotated code

```scala
trait Rule[-In, +Out, +A, +X] extends (In => Result[Out, A, X]) {
  val factory : Rules
  import factory._

  def as(name : String) = ruleWithName(name, this)

  def flatMap[Out2, B, X2 : X](fa2ruleb : A => Out => Result[Out2, B, X2]) = mapResult {
    case Success(out, a) => fa2ruleb(a)(out)
    case Failure => Failure
    case err @ Error(_) => err
  }

  def map[B](fa2b : A => B) flatMap { a => out => Success(out, fa2b(a)) }

  def _>[Out2, B, X2 : X](fa2ruleb : A => Out => Result[Out2, B, X2]) : flatMap(fa2ruleb)

  def _>[Out2, B, X2 : X](fa2resultb : A => Result[Out2, B, X2]) : flatMap { a => any => fa2resultb(a) }

  def _>[Out2, B, X2 : X](pf : PartialFunction[A, Rule[Out, Out2, B, X2]]) : filter(pf.isDefinedAt _) flatMap pf

  def _>[Out2, B, X2 : X](next : => Rule[Out, Out2, B, X2]) : for (a <- this; b <- next) yield new -(a, b)

  def _>[Out2, B, X2 : X](next : => Rule[Out, Out2, B, X2]) : for (a <- this; b <- next) yield a

  def _>[Out2, B, X2 : X](next : => Rule[Out, Out2, B, X2]) : for (a <- this; b <- next) yield b

  def _>[Out2, B, X2 : X](next : => Rule[Out, Out2, Seq[B], X2]) : for (a <- this; b <- next) yield a :: b.toList

  /** Apply the result of this rule to the function returned by the next rule */
  def _>[Out2, B, X2 : X](next : => Rule[Out, Out2, A => B, X2]) : for (a <- this; fa2b <- next) yield fa2b(a)

  /** Apply the result of this rule to the function returned by the previous rule */
  def _>[InPrev, B, X2 : X](prev : => Rule[InPrev, In, A => B, X2]) : for (fa2b <- prev; a <- this) yield fa2b(a)

  def _>[Out2, B, X2 : X](next : => Rule[Out, Out2, B, X2]) : for (a <- this; b <- next orError) yield new -(a, b)

  def _>[Out2, B, X2 : X](next : => Rule[Out, Out2, B, X2]) : for (a <- this; b <- next orError) yield a

  def _>[Out2, B, X2 : X](next : => Rule[Out, Out2, B, X2]) : for (a <- this; b <- next orError) yield b

  def _>[In2 < In](exclude : => Rule[In2, Any, Any, Any]) : !exclude -- this

  def _>[B1, B2, B : A <= B1 - B2, C](f : (B1, B2) => C) : map { a =>
    (a : B1 - B2) match { case b1 - b2 => f(b1, b2) }
  }
}
```
Demo

Introduce variable
Inline variable refactoring

- **Goal**: Inline all occurrences of a variable
- **Challenge**: Computing reaching definitions of local variables to restrict the scope of refactoring
Does definition reach a usage?

```scala
var a = 1
val cl = {
  println("Returning a closure");
  {(x: Int) => a = x}
}
foo(cl)
println(a)
```

The variable `a` cannot be inlined because it may be reassigned via closure `cl`
Finally, extract method in Scala

• **Goal**: Extract a piece of code into a separate function

• **Challenges**:  
  • Scoping  
  • Computing input and output variables of the code fragment
Demo

Extract method
Closure with state are evil!

```
def foo = {
  var a = 42
  
  val cl = (i: Int) => {
    a = a + i
  }
  doSomething(cl)
  print(a)
}
```

Closure `cl` changes the local environment of the method `foo`
Solution: introduce environment instance

```scala
class MyMethodEnv(var a: Int)
def myMethod1(env: MyMethodEnv) =
  (i: Int) => {
    env.a = env.a + i
  }
def foo = {
  val env = new MyMethodEnv(a = 42)
  val cl = myMethod1(env)
  doSomething(cl)
  println(env.a)
}
```
Abstracting context

- Closures change state
  - Introduce environment object
- Dependent types, inner classes or type aliases
  - Introduce generics, presumably, of higher kind
- Local implicit conversions
  - Cannot be abstracted (at least, not elegantly)
A correct refactoring sometimes makes code very messy...

How should the boundary between useful and correct refactoring be defined?
Refactoring implementation is a tradeoff

Generality

Elegance  Correctness
Future directions?

• Scripting language - generality
• Language of constraints - correctness
• Software metrics - elegance
Thanks for your attention

Questions?