Exploring Possibilities for Symmetric Implementation of Aspect-Oriented Design Patterns in Scala

Pavol PIDANIČ[*]

Slovak University of Technology in Bratislava
Faculty of Informatics and Information Technologies
Ilkovičova 2, 842 16 Bratislava, Slovakia
`ppidanic@gmail.com`

Abstract. In aspect-oriented programming, it may be distinguished between asymmetric and symmetric implementation. Aspect-oriented design patterns are known in their AspectJ-like, asymmetric implementation. One of their categorization is by dominating element of aspect-oriented programming—pointcut patterns, advice patterns and inter-type declaration patterns. Scala is a programming language which joins object-oriented and functional paradigm. Despite Scala is not fully aspect-oriented, it contains language mechanisms which can implement aspect-oriented feature—advice in symmetric way. This paper presents three aspect-oriented design patterns. We have demonstrated that Cuckoo's Egg and Director described in Coplien's form can be implemented in Scala. We have also described Worker Object Creation pattern in similar manner and created an implementation in Scala.

1 Introduction

Scala[1] is a multiplatform, strong static typed language that runs on Java Virtual Machine. Scala joins object-oriented and functional programming paradigm together. It is possible to import existing Java classes and libraries into Scala code.

In aspect-oriented programming, it may be distinguished between asymmetric and symmetric implementation. In asymmetric approach we distinct between the base and aspects that affects this base and base should not be aware of the aspects. AspectJ is known as asymmetric and has influenced use of aspect-oriented programming. In symmetric approach we consider how to merge things, we treat all elements equally. Elements are composed without explicitly denoting as aspects and base [2]. It has its importance in modeling. Scala contains some aspect-oriented features.

In section 2 we describe aspect-oriented features in Scala. In sections 3, 4 and 5 we present three aspect-oriented design patterns and their symmetric implementation in Scala. In section 6 we

[*] Master study programme in field: Software Engineering
Supervisor: Assoc. Professor Valentino Vranić, Institute of Informatics and Software Engineering, Faculty of Informatics and Information Technologies STU in Bratislava

[1] http://www.scala-lang.org/

discuss asymmetric implementation of presented design patterns. In section 8 we describe related work and conclusion of this work is in section 9.

2 Scala and aspect-oriented programming

Despite Scala is not designed as aspect-oriented, it contains language mechanisms which exhibit symmetric aspect-oriented features [2]. These features partly reproduce all features of aspect-oriented programming as known in AspectJ.

The language mechanism is *trait*. Traits can be used as *mixin(s)*. With mixins we can not only define, but also provide behaviour for descendants. Traits can also define state and be combined together.

AspectJ-like advice *before*, *after*, *around* can be created with idiom *stackable modification* [9]. Advice is implemented with special construct `abstract override` in extending class.

Generally, one can handle all advice on method joinpoints. However, there is no quantification over joinpoints and pointcuts as in aspect-oriented programming [8].

Aspect-oriented design patterns are known almost exclusively in their asymmetric, AspectJ-like implementation. Design patterns should be described in general way independent of implementation details. One of the way is a Coplien's form [4]—set of definitions that describe patterns in general manner—as presented by Bálik and Vranić [1]. They implemented some aspect-oriented design patterns in HyperJ programming language.

One of the categorization of aspect-oriented design pattern is by dominating element of the asymmetric aspect-oriented programming—pointcut patterns, advice patterns and inter-type declaration patterns [6].

As described above, Scala is not fully aspect-oriented. Hence we focus only on inter-type declaration and advice patterns—Cuckoo's Egg, Director, Worker Object Creation.

3 Cuckoo's Egg pattern

Main use of the Cuckoo's Egg is to capture a constructor call and instead create or provide an object of an another type.

Bálik and Vranić [1] created a description in the Coplien's form as follows:

Problem: Instead of an object of the original type, under certain conditions, an object of some other type is needed.
Context: The original type may be used in various contexts. The need for the object of another type can be determined before the instantiation takes place.
Forces: An object of some other type is needed, but the type that is going to be instantiated may not be altered.
Solution: Make the other type subtype of the original type and provide its instance instead of the original type instance at the moment of instantiation if the conditions for this are fulfilled.
Resulting Context: The original type remains unchanged, while it appears to give instances of the other type under certain conditions. There may be several such types chosen for instantiation according to the conditions.
Rationale: The other type has to be a subtype of the original type.

We used this description to a demonstration that the pattern can be implemented in Scala programming language.

Instead of direct instantiation of an object one could use the object-oriented design pattern Factory Method [7]. Hence we were able to implement the Cuckoo's Egg pattern in Scala as follows where we substitute instance of `Bird` with instance of `Cuckoo`:

```
trait Egg {
  def hatch: Unit
}

trait Nest {
  var eggs: List[Egg] = Nil

  def newEgg: Unit = {
    var egg = layEgg
    eggs = egg :: eggs
    egg.hatch
  }

  def layEgg: Egg
}

class BirdEgg extends Egg {
  def hatch: Unit = {
    println ("bird")
  }
}

class BirdNest extends Nest {
  def layEgg: Egg = {
    new BirdEgg
  }
}

class CuckooEgg extends Egg {
  def hatch: Unit = {
    println ("cuckoo")
  }
}

trait Cuckoo extends Nest {
  abstract override def layEgg = {
    new CuckooEgg
  }
}
```

4 Director pattern

The Director pattern defines additional roles that are enforced onto the existing types without changing an existing implementation.

A general description of the Director pattern in the Coplien's form by Bálik and Vranić [1]:

Problem: Additional roles have to be defined in application.
Context: A type hierarchy that defines the roles.
Forces: The application has to be extended with additional roles, but the original class hierarchy in the source code has to remain free from these roles.
Solution: Introduce the additional roles as types and enforce their implementation by the corresponding types externally.
Resulting context: The type hierarchy preserved in the source code, but extended with new roles in

execution.
Rationale: Director provides two main benefits: the application behavior can be easily changed by replacing a particular concern and the core functionality is less complicated.

We used the same approach as with the Cuckoo's Egg pattern in section 3 and demonstrated a Scala implementation of the Director pattern. The Observer object-oriented design pattern presented by Skeel [10] exhibits the Director pattern features at the same time.

Trait `SuperSensor` extends a behaviour of the class `Sensor` with a new functionality of notifying attached observers, when a value on sensor changes:

```
trait Subject[T] {
  self : T =>
    private var observers: List[T => Unit] = Nil

    def subscribe(obs: T => Unit):Unit =
        observers = obs :: observers

    def unsubscribe(obs: T => Unit):Unit =
        observers = observers filter (_ != obs)

    def publish = for(obs <- observers ) obs(this)
}

class Sensor(val label: String){
  var value:Double = _

  def changeValue(v: Double) =
    value = v
}

class Display(val label: String) {
  def notify(s: Sensor) =
    println(label + "␣" + s.label + "␣" + s.value )
}

trait SuperSensor extends Sensor with Subject[Sensor] {
  override def changeValue(v: Double) = {
    super.changeValue(v)
    publish
  }
}
```

5 Worker Object Creation pattern

The Worker Object Creation pattern delays executing (mostly time-consuming) methods with a worker object without interfering main thread of an application.

We created a description in the Coplien's form as follows:

Problem: Delaying execution of some method out of main application thread.
Context: Order and time of an execution of some methods is not important.
Forces: Delayed execution of a method runs later.
Solution: Create an worker object and encapsulate a method and delay its execution later.
Resulting context: Methods calls are preserved, but executed later.
Rationale: Particular method calls must be out of main application thread. Worker objects can be

passed to an other context.

```
class Later {
  def doLater: Unit =
    println("method␣execution")
}

import java.awt.EventQueue

trait WorkerObject extends Later {
  abstract override def doLater = {
      def proceed = super.doLater _

      val worker =  new Runnable() {
                   def run() {
                     Thread.sleep(5000);
                     println("late␣execution")
                     proceed()
                   }
           }
      EventQueue.invokeLater(worker)
  }
}
```

6 Asymmetric implementation of design patterns

As mentioned above in asymmetric approach we distinct between base and aspects that affects base functionality and base should not be aware of the aspects. In AspectJ poincuts are used.

In Scala there is no general quantification over joinpoints, specially pointcuts definition. Spiewak and Zhao [11] introduced an experimental framework that supports pointcuts declaration. This framework uses its own Domain Specific Languages and Proxy Methods. Scala AOP framework uses non-transparent mechanism for method interception and every captured class method requires a call to a singleton delegate. Delegate function accepts a functional which represents original method body. Poincuts can capture types and names of methods and they have access to a context of instance variables and match `execution` a `within` in AspectJ programming language. Before and after-like advice functionality is supported.

As consequence that the framework lacks around-like advice we are not able to implement two of presented design pattern because both Cuckoo's Egg [7] and Worker Object Creation [5] require replacing original method execution in their original AspectJ-like implementation.

7 Discussion

An asymmetric implementation of design patterns has been developed first in AspectJ by Laddad [5] and Miles [7].

We used the same description without any modification and sustained with the implementation in Scala language. Although our description of the Worker Object Creation has not been sustained, the provided implementation of two design patterns advances the Coplien's form as the general description of the design patterns independent of the symmetric or the asymmetric view of the aspect-oriented programming.

8 Related work

In the section 3 we described the implementation of the Cuckoo's Egg pattern that uses the Factory Method pattern and in the section 4 Scala implementation of the object-oriented Observer pattern exhibits Director aspect-oriented design pattern. Bača and Vranić [3] show that there is a correlation between object-oriented and aspect-oriented design patterns. The Director pattern can substitute many patterns, indicating Prototype pattern. The Worker Object Creation substitutes Proxy pattern and the Cuckoo's Egg replaces Abstract Factory, Singleton and Flyweight.

9 Conclusion and future work

Scala is a modern programming language which joins object-oriented and functional paradigm that become more popular in business. The same holds for the aspect-oriented programming with main language AspectJ. Aspect-oriented design patterns are known almost exclusively in their asymmetric, AspectJ-like implementation. Design patterns should be described in the general way.

This paper presented three aspect-oriented design patterns—Cuckoo's Egg, Director and Worker Object Creation. The Cuckoo's Egg and the Director are described in the Coplien's form as presented by Bálik and Vranić [1]. We used the same description without any modification and sustained with the implementation in Scala language. We also described and provided the implementation of the Worker Object Creation pattern in the similar manner.

A future work could focus on a description of another aspect-oriented design patterns and an implementation in Scala also with the framework. An extension of the framework is another interesting study for the future work.

Acknowledgement: The work reported here was supported by the Scientific Grant Agency of Slovak Republic (VEGA) under the grant No. VG 1/1221/12.

References

[1] Bálik, J., Vranić, V.: Sustaining Composability of Aspect-Oriented Design Patterns in Their Symmetric Implementation. In: *2nd International Workshop on Empirical Evaluation of Software Composition Techniques*. ESCOT 2011, at ECOOP 2011, 2011.

[2] Bálik, J., Vranić, V.: Symmetric Aspect-orientation: Some Practical Consequences. In: *Proceedings of the 2012 Workshop on Next Generation Modularity Approaches for Requirements and Architecture*. NEMARA '12, New York, NY, USA, ACM, 2012, pp. 7–12.

[3] Bača, P., Vranić, V.: Replacing Object-Oriented Design Patterns with Intrinsic Aspect-Oriented Design Patterns. In: *Engineering of Computer Based Systems (ECBS-EERC), 2011 2nd Eastern European Regional Conference on the*, 2011, pp. 19–26.

[4] Coplien, J.: Design pattern definition, `http://www.hillside.net/component/content/article/50-patterns/222-design-pattern-definition`.

[5] Laddad, L.: AspectJ in Action: Practical Aspect-Oriented Programming, 2003, Manning Publications Co., Greenwich, CT, USA.

[6] Menkya, R.: Aspect-Oriented Design Patterns, 2007, Bachelor Thesis, Faculty of informatics and information technologies, Slovak university of technology in Bratislava.

[7] Miles, R.: AspectJ Cookbook, 2004, O'Reilly Media, Inc.

[8] Odersky, M.: The Scala Experiment – Can We Provide Better Language Support for Component Systems? In Chin, W.N., ed.: *Programming Languages and Systems*. Volume 3302 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2004, pp. 364–365.

[9] Odersky, M., Spoon, L., Venners, B.: Programming in Scala: A Comprehensive Step-by-step Guide, 2008, Artima Incorporation.

[10] Skeel Løkke, F.: Scala & Design Patterns (Exploring Language Expressivity). Masters thesis, University of Aarhus, Department of Computer Science, 2009, `http://www.scala-lang.org/old/sites/default/files/FrederikThesis.pdf`.

[11] Spiewak, D., Zhao, T.: Method Proxy-Based AOP in Scala. In: *Journal of Object Technology, vol. 8, no. 7*, 2009, pp. 149–169.