U.P.B. Sci. Bull., Series C, Vol. 72, Iss. 4, 2010 ISSN 1454-234x

CODE-SMELLS IN AOP

Şerban DRĂGĂNESCU[1], Nicolae ŢĂPUŞ[2]

În această lucrare ne propunem să prezentăm câteva "code smells" pe care le-am identificat studiind implementări clasice pentru şabloane de proiectare, care folosesc programarea orientată pe aspecte. Sunt probleme de design de natură să afecteze performanţa care nu ar fi existat dacă tehnologia orientată pe aspecte nu ar fi fost folosită. Cazurile prezentate sunt exemplificate cu şabloane de proiectare, în implementări clasice şi originale.

We intend to present in this article a few "code smells" we identified by studying classic implementations of design patterns, that are using aspect oriented programming. The identified issues are design problems that affect the performance, problems that would not have existed if aspect oriented technology would not had been used. The cases are presented with design pattern examples, in classic and original implementations.

Keywords: code smell, bad smell, AspectJ, AOP, design pattern, performance hash, syncronized

1. *Code smells*

Code smells is an expression used for shallow signs showed by source code of programs that may indicate a deeper problem. The problem itself is usually not a functional matter, but it may hinder performance or be considered bad practice, in the sense that code may be difficult to understand, extend, debug or maintain. The expression was originally coined by Kent Beck in [1] for the purpose of finding heuristics for refactoring object oriented programs. Examples of classic code smells are considered to be code that is duplicated, methods with too many parameters, methods with big or too little bodies or a class that excessively use another class. These "smells" are just signs that something *may* be wrong or could be done better. There are cases where even code duplication, our first "smell" example, is just necessary to prevent run-time decisions at the expense of compile-time decisions (code duplication) in areas where performance is critical.

[1] Eng., Faculty of Automatic Control and Computers, University POLITEHNICA of Bucharest, Romania, e-mail: draganescu@gmail.com
[2] Prof, Faculty of Automatic Control and Computers, University POLITEHNICA of Bucharest, Romania

2. Aspect oriented *code smells*

There are two directions of study in this field. One of them is centred on removing traditional, OOP, code-smells with the means offered by AOP. The other one focuses on identifying code-smells that are generated by AOP, like [5] and the current paper.

During our studies of AOP overhead we took several approaches, one[3] of them being the study of software constructs that appear as side-effects of AOP technology. While looking at design patterns implementations using AOP we have realised that some software constructs[4] are recurring in the solutions we find, constructs that would not exist if aspect oriented programming would have not been used. A good example for these constructs is the implementation of the *Observer* design pattern. We present in Illustration 1 a variant of the solution published by Hannemann and Kiczales in [2]. We will also refer to the object oriented version, but because it is well known, we will not present it here. The standard Java implementation (java.util.Observable) may be used as reference instead. Other authors (like Monteiro in [7]) have taken a critic look at the AOP design pattern implementations from [2] but none from the performance penalty perspective. The works of Monteiro in [7] mostly refer usability and functionality issues.

```
public abstract aspect ObserverProtocol {
  private WeakHashMap perSubjectObservers=new WeakHashMap();   (1)
  protected synchronized List getObservers(Object s) {         (2)
       List observers = (List)perSubjectObservers.get(s);
       if (observers == null) {
           observers = new LinkedList();
           perSubjectObservers.put(s, observers);
       }
        return observers;
  }
  public synchronized void addObserver(Object s, Object o){    (3)
       getObservers(s).add(o);
  }
  public synchronized void removeObserver(Object s,Object o){ (4)
       getObservers(s).remove(o);
  }
  protected abstract pointcut subjectChange(Object s);         (5)
```

[3] Another one is the study of intrinsic AOP overhead, which happens when everything is equal between AOP and non AOP implementations, like in [3].

[4] We are not calling them “patterns” to avoid a confusion with the already established *design patterns*, even though they are similar.

```
  after(Object s): subjectChange(s) {                         (6)
      synchronized (this){
         Iterator iter = getObservers(s).iterator();
         while (iter.hasNext()) {
           updateObserver(s, iter.next());
         }
      }
  }
  protected abstract void updateObserver(Object s, Object o); (7)
}
```

Illustration 1: Implementation for Observer design pattern, [2]

The modifications of the implementation (from the original presented in [2]) are underlined and they consist mainly in methods being synchronised in order to support multi-threaded hash access.

The example in Illustration 1 is the abstract aspect. It needs to be extended by a concrete aspect that implements the abstract pointcuts and methods. In order to understand how this aspect works, we will give an example of concrete aspect in Illustration 2. Any Mouse instance can be a subject, and Cats are observers that watch mice's method "move".

```
public aspect MouseObserver extends ObserverProtocol {
  public pointcut subjectChange(Object subject) :             (8)
      execution(public void Mouse+.move(..)) && this(subject);

  public void updateObserver(Object subject, Object observer){(9)
      Mouse m = (Mouse)subject;
      Cat c = (Cat) observer;
      System.out.println("Cat " + c.name + " sees " + m.name);
  }
}
```

Illustration 2: Concrete aspect for Observer design pattern

What this implementation does is to modify all join-points that are caught by the "subjectChange" pointcut (5 and 8) and after each of them to call the advice (6) where all of the registered observers are notified. To make this even clearer, in Illustration 3 we present a usage case.

```
Mouse jerry = new Mouse("Jerry");
Mouse roquefort = new Mouse("Roquefort");
Cat tom = new Cat("Tom");
MouseObserver.aspectOf().addObserver(jerry, tom);
MouseObserver.aspectOf().addObserver(roquefort, tom);
roquefort.move(2, 2);
jerry.move(3, 2);
```

Illustration 3: Usage of the Observer design pattern

For each existing subject in the virtual machine, one concrete aspect will be created, but the abstract aspect would be shared for all the subjects. Therefore the code inside the abstract aspect (Illustration 1) would be shared, including the hash defined in (1), which will contain subject-observer pairs for the whole virtual machine. To make this clear, we will spell out how the 3 usage cases are working: observer notification, observer adding and observer removing. Notifying an observer consists of:

-retrieving the subject (of the current joinpoint) from the hash shown in (1)
-if the subject exists then an iterator of all observers is created
-all observers from the iterator are notified.

Adding an observer consists of:

-looking up the associated subject in the hash and add it if it does not exist
-adding the observer in the subject's associated list

Removing an observer consists of:

-looking up the associated subject in the hash
-removing the observer from the subject's associated list.

Join point resolution

The first AOP code smell we identified is the usage of a hashmap construct, (1), for the storage of types or instances. This is the first step in all three usage cases. We realised that this needs to happen because pointcuts are defined in relation with types (names of fields, methods and classes) while the constructs might apply to subtypes or instances. Therefore hashes would store the members of the relation: subtypes of a type or instances of a type. We named *join point resolution* the process of associating the join points from a pointcut with an instance or subtype.

Hashes are not needed in a normal object-oriented implementation, as the list of observers is found directly as a member of the subject.

Hashes are a common construct in software, and whole development paradigms rely on their usage, like J2EE web applications. Their main overhead consists of the lookup or modification computation time, which is quite efficient and therefore easy to tolerate.

Table 1

Join point resolution

Overhead type	Hash lookup of the Subject
OOP situation	Notification code is placed inside each Subject and called from there
Other problems	Using a hash also implies the need for synchronised access

Synchronisations

Using hashes, however, leads to a second code smell: their operations need to be synchronised, which is another performance overhead! Occasionally, in applications hashes can be used without the synchronisation, because the operations are known to be serial. However, if the operations need to be generic, like in a library, they need to be protected for concurrent access and read accesses (2) and write accesses (3 and 4) are protected by a synchronised zone in Illustration 1.

Table 2

Synchronisations

Overhead type	Synchronised area usage
OOP situation	Subjects need not to be retrieved but Observers are stored in lists that need protection for concurrent access and copy-on-write may be used as a solution.

Bottlenecks

The above mentioned synchronisation is done for all the subjects in the virtual machine, as the aspect is a singleton aspect, and this is a potential bottleneck in an application where all subjects and observers reside in the same hash in the same synchronised area. Bottlenecks are the third code smell we generally identified, and the Observer design pattern implementation helped illustrate. Normally, an object oriented implementation of this pattern would keep lists of observers inside each subject, therefore the bottleneck would not exists.

Table 3

Bottlenecks

Overhead type	Threads wait for each other on bottlenecks like single hash and single synchronised area.
OOP situation	The Observers are distributed to each Subject, access to lists, hashes or synchronised areas is done per Subject.

Generous Decorations

One more subtle issue that may arise is what we have called “generous decoration”. This code-smell consists in inserting a lot of advices that only fire when some decision is taken at runtime. The pointcut defined in (5 and 8) decorates all the calls of the “move” method, but if the respective mouse participates or not in a Observer pattern is decided inside (6) when the iterator turns out either full or empty. The runtime decision is not a huge overhead, but implementers should take it into consideration, especially where the usage ratio (number of cases where the advice fires divided to the total number of join-points in the pointcut) is small. The chosen example does not underline well this smell. A better example would make the case of the Decorator design pattern, in the AOP implementation. In the AOP Decorator, any decoration would affect all instances of one type.

Table 4

Generous Decoration

Overhead type	Code is added to innocent bystander areas
OOP situation	Notification code is also added in all Subject instances.

3. Avoiding the smells

We are trying to show ways of avoiding some of the smells we have described earlier. This chapter is not a recipe against the code smells as we consider implementation to be a matter of taste and specific problems may have specific solutions. However, in order to prove that there are also solutions that are still aspect oriented and do not present the mentioned code-smells, we present another Observer design pattern implementation, in Illustration 4 and a concrete Observer in Illustration 5.

```
public abstract aspect ObserverProtocol
                           perthis(subjectCreation(Object)){ (10)
   protected Object subject;                                  (11)
   private LinkedList observers = new LinkedList();           (12)

   public void addObserver(Object o) {                        (13)
       synchronized(subject){
              observers. add(o);
       }
   }
   public synchronized void removeObserver(Object o) {         (14)
       synchronized(subject){
              observers.remove(o);
       }
   }
   protected abstract pointcut subjectCreation(Object subject);
                                                              (15)
   after(Object subject): subjectCreation(subject) {          (16)
       this.subject = subject;
   }
   protected abstract pointcut subjectChange(Object subject);(17)
   after(Object subject): subjectChange(subject) {             (18)
       synchronized(subject){                                   (19)
              Iterator iter = observers.iterator();              (20)
           while (iter.hasNext()) {
               updateObserver(iter.next());
           }
       }
   }
   protected abstract void updateObserver(Object observer);
}
```

Illustration 4: Smell-free Observer implementation, abstract part

We see at (20) that this aspect is declared “perthis”, which means that there will be one instance of the aspect for each parameter to the “perthis” clause. The parameter is a pointcut, which leads us to the down side of this approach: there is a need for a second pointcut (15 and 23) that is used (16) for the initialisation of the concrete aspect. Because the aspect is now “perthis”, there is no need to have a hash to select the current instance and the linked list is stored directly in the aspect. Also the synchronised zones that access the list are synchronised on each subject, lowering the contention and removing the bottleneck we had earlier (Illustration 1), where all subjects in the virtual machine resided in a single hash.

```
public aspect MouseObserver extends ObserverProtocol {  (21)
       public pointcut subjectChange(Object subject) :   (22)
            execution(public void Mouse+.move(. .)) &&
            this(subject);

       public pointcut subjectCreation(Object subject) : (23)
              initialization(Mouse+.new(. .)) && this(subject);

       public void updateObserver(Object observer) {     (24)
              Cat cat = (Cat)observer;
              Mouse m = (Mouse)subject;
              cat.update(m.name);
       }
}
```

Illustration 5: Smell-free Observer implementation, concrete part

To summarise, we present Table 5, where the columns represent, respectively, results for the Observer OOP implementation (eg java.util.Observable), the AOP classic implementation (Illustration 1) and the AOP smell-free proposal (Illustration 4).

Table 5

Code-smell presence

smell \ implementation	OOP	AOP1	AOP2
pointcut resolution	No	Yes	No
synchronisation	Yes*	Yes	Yes*
generous decoration	Yes**	Yes	Yes
bottleneck	No	Yes	No

*Synchronisation is present in all 3 implementations, but its usage is different. In OOP and AOP2 it is related to keeping track of multiple observers in one subject (in a list) while in AOP1 it is also about keeping track of subjects (in a hash), so it is actually needed at two levels but since the two resources (the hash and the list) are used in the same place, only one synchronised zone is used. We needed to mention that, because the synchronisation need for the hash is removed in OOP and AOP2 implementations but the chosen example is not really helpful in showing the removal of the code-smell.

**Generous decoration happens in all the implementations. In OOP the Mouse type would extend Subject and contain all the necessary code, even if the code is never called when there are no registered observers. However, a type that just inherits Subject and does not call the notification it does not bear the overhead in any way. In the AOP cases, if the type is caught in the pointcut, the notification is fired, although the observer list is void. A better example of this situation might be given by the AOP decorator design pattern: if a type gets decorated then a flag is needed to determine if the decoration should be executed or not, otherwise all instances of the type are forcedly decorated.

4. Conclusion

We have identified four AOP code-smells and we have illustrated them with implementations of the Observer pattern. Hashes, synchronisations, generous decorations and bottlenecks are constructs that appear sometimes unnecessarily in AOP implementations. Our AOP Observer design pattern implementation along with the normal OOP implementation prove that those kind of constructs can be avoided.

R E F E R E N C E S

[1] *M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts,* Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999

[2] *J. Hannemann, G. Kiczales,* Design Pattern Implementation in Java and AspectJ, ACM SIGPLAN Notices, **vol. 37**, issue 11 , pages: 161–173, 2004

[3] *Ş. Drăgănescu, N. Ţăpuş,* COSTUL DE PERFORMANŢĂ AL PROGRAMĂRII ORIENTATE PE ASPECTE (Performance cost of AOP), Revista Română de Informatică şi Automatică, **vol. 19**, nr. 4, Bucureşti, 2009

[4] *V.C. Garcia, E.K. Piveta, D. Lucredio, A. Alvaro, E. Santana de Almeida, A. Francisco do Prado, L.C. Zancanella,* Manipulating Crosscutting Concerns, SugarLoafPLoP 2004

[5] *E.K. Piveta, M. Hecht, M.S. Pimenta, R.T. Price, Bad Smells* em Sistemas Orientados a Aspectos (*Bad smells* in Aspect Oriented Systems), SBES 2005

[6] *M. Iwamoto, J, Zhao,*Refactoring Aspect-Oriented Programs, The 4th AOSD Modeling With UML Workshop, 2003

[7] *M. Pessoa Monteiro, J.M. Fernandes,* Pitfalls of AspectJ Implementations of Some of the Gang-of-Four Design Patterns, Desarrollo de Software Orientado a Aspectos, 2004

[8] *M. Pessoa Monteiro, J.M. Fernandes,* An illustrative example of refactoring object-oriented source code with aspect-oriented mechanisms, Software—Practice & Experience, **vol 38**, issue 4, pages: 361-396, 2008

[9] *E.K. Piveta; M. Hecht, M.S.Pimenta, R.T. Price,* - Detecting Bad Smells in AspectJ, Journal of Universal Computer Science, **vol. 12**, no. 7 2006.