

Using Design Patterns as Indicators of Refactoring Opportunities (to Aspects)

-- Position paper submitted to the LATE workshop at AOSD 2006 --

Miguel P. Monteiro
Escola Superior de Tecnologia, Instit. Politécnico de Castelo Branco,
Avenida do Empresário 6000-767 Castelo Branco, Portugal
mmonteiro@di.uminho.pt

1. Introduction

In this paper, we focus on the problem of identifying refactoring opportunities¹ in object-oriented (OO) legacy systems, in the light of aspect-oriented programming (AOP). Our position is based on the premise that design patterns comprise an important technique used by developers of OO systems to cope with crosscutting. We argue that such efforts would benefit of a more systematic knowledge of the uses that developers make of design patterns in such circumstances. Such knowledge promises to yield useful catalogues of refactoring opportunities, i.e., descriptions of situations in OO code that can be improved using AOP's superior compositional capabilities.

This paper is intended to be approached as a true "position paper": we describe our current position and proposed approach towards meeting the stated goal. We limit ourselves to a general explanation of the approach and do not present concrete results.

This paper is structured as follows. Section 2 presents our position. To better support the position, we present a few illustrative examples in section 3. Section 4 outlines the approach we propose to meet our goals. Section 5 concludes the paper.

2. Patterns as indications of refactoring opportunities

In the last decade, design patterns became increasingly popular as a way to express design solutions to recurring problems [5]. The 1990s witnessed a veritable industry of "pattern hunting" and as a result we now have a rich repository of object-oriented (OO) patterns. Patterns are currently regarded as an essential component of the skills of any programmer involved in developing, maintaining and evolving object-oriented systems.

However, it is also possible to view patterns in a more negative light. Patterns are problem-solution pairs [12], meaning that whenever we use a pattern, there must be

¹ For the purposes of this paper, the concept of refactoring opportunity is akin to that of *code smell* as proposed in [4].

problem that the pattern is supposed to solve, or at least circumvent. In a significant number of cases, the problem stems from limitations in the OO language used. When a feature or composition capability is not directly available in a language, the solution often lies in implementing some pattern that achieves the intended effect, usually at some cost in flexibility and added complexity. Often, patterns are used as work-arounds for limitations that theoretically need not exist. In such a light, the existence of such a large variety of patterns seems to suggest that existing OO languages are rather limited.

Let us give some examples: (1) *Abstract Factory* (pages 87-96 of [5]) proposes a way to emulate co-variance, (2) *Factory Method* (pages 107-116 of [5]) describes a way to emulate polymorphic construction of objects (directly supported in languages such as Objective C), (3) *Prototype* (pages 117-126 of [5]) is a way to emulate the prototype-cloning effect found in prototype-based languages such as Self [11] (4) *Decorator* (pages 175-184 of [5]) describes a way to emulate mixins [2] and (5) *Visitor* (pages 331-344 of [5]) proposes a way to emulate multiple dispatch. Many other examples could be given, though an extensive list lies out of the scope of this paper.

Despite its limitations, OO is a rich paradigm that sometimes enables multiple variants to achieving a given effect. Many design problems can be addressed by a plethora of different solutions, each one providing its unique set of specific advantages and trade-offs. This richness makes it likely that different programmers working in different contexts may select different solutions to deal with the same problem.

It has been noted that some of the motivations for implementing patterns has its roots in crosscutting, such as those that can be effectively tackled using aspect-oriented languages. Some patterns are known to “disappear” when implemented using AOP, while other patterns witness a significant simplification in their implementations [6]. As with other design problems, we observe a rich variety of different OO design solutions in relation to crosscutting. As a consequence, symptoms of the presence of crosscutting concerns in OO legacy systems can take many different forms and patterns. In the next section, we mention a few testimonies that can be found in the literature. We believe that patterns may offer a rich set of clues of when to refactor well-formed OO systems to aspects.

Why do we focus on patterns? In the context of this paper, we’re referring to well-formed OO code, developed by experienced and knowledgeable programmers. Such programmers are more likely to use patterns well (in an OO sense) than novice programmers and be aware that “duplication is evil” [4]. Therefore, they take great effort to remove such duplication, by keeping their code clean through refactoring. It seems reasonable to assume that many such programmers resort to patterns to deal with such issues, both when designing [5] and when refactoring [8].

On the other hand, the compositional capabilities of OO – currently the dominant programming paradigm – do not seem to be sufficient to cope with all demands of modern software, namely crosscutting. There seems to be a conflict between the stated aims of the test-driven and refactoring communities and what can be achieved with the programming paradigm that most people from those communities (currently) use. Most testimonies from these communities suggest that all forms of duplicated code can be eliminated from OO code. And yet, this claim is likely to raise eyebrows

from among the AOP community, given the close link between crosscutting and duplication. Note that crosscutting is not generally mentioned in [5, 8] as a reason to use patterns.

In our view, what explains this apparent conflict is the use of elaborate design structure, namely patterns, to mask the symptoms of duplication. Developers that “mercilessly refactor” a given OO code base until all manifestations of duplication are removed, are really trading one problem with another: they merely remove the *semblance* of duplication, replacing it with increased structural complexity and inflexibility. There is a risk that the refactored structure proves to be almost as hard to evolve and reason with as the original one. By contrast, AOP promises to provide developers with more acceptable trade-offs. That is the claim suggested by Isberg in [7], which analyses the structure of the JUnit framework [1]. Isberg discusses trade-offs of the current design decisions for JUnit and proposes reimplementations of some parts of the framework using pointcuts and advice, pointing out that most of what he proposes to reimplement is, to current thinking, well-modularised.

3. A few illustrative examples

In this section, we describe a few examples of the use of OO design patterns that are used to cope with crosscutting. We suggest hypothetical refactorings that yield better AOP alternatives. Throughout the descriptions, we assume the reader has a general knowledge of the Gang-of-Four patterns [5] and refrain from providing descriptions of the patterns.

Decorator

In [3], Feathers describes various techniques to deal with cases in which additional logic must be added to the core logic of a system. Feathers mentions as a typical case a situation in which the new logic to be added happens to execute at the same time as the one in a method, giving rise to temptation to place it in the same method. However, the new logic is otherwise unrelated and there is a chance that in future someone will want to use one without the other. Feathers uses logging as an example of such an additional logic. People familiar to AOP recognise the favourite example of crosscutting (though in recent times it has been challenged by the *Observer* pattern). To address this problem, Feathers proposes a few techniques that include *Wrap Method* (pages 67-70 of [3]) and *Wrap Class* (pages 71-76 of [3]). There are a few variants to implementing *Wrap Method*, but it basically entails wrapping the method with the original logic with a new method that simply performs the additional logic (before or after, depending on the specific problem) and forwards it to the old method. *Wrap Class* is, by Feathers’ own admission, really an instance of *Decorator*.

The two above techniques suggest two new AOP refactorings: *Replace Wrap Method with Pointcut and Advice* and *Replace Decorator with Aspect*. We envision *Replace Wrap Method with Pointcut and Advice* as creating a pointcut that captures the points in the execution of the program where the wrap method is called and add-

ing an advice acting on those joinpoints that provides the logic formerly provided by the wrap method. Next, the wrap method can be removed. *Replace Decorator with Aspect* is about creating an aspect that captures joinpoints where the behaviour that the decorator decorates is called, and placing in the aspect an advice acting on those joinpoints that provides the logic of the decorator. The decorator class can probably be removed afterwards. In the simplest cases, using an AOP implementation of *Decorator* proposed in [6] may be sufficient.

Template Method

Template Method (pages 325-330 of [5]) looks very promising as a signal of refactoring opportunities. It comprises one of the design backbones of many OO frameworks and frequently features in APIs. Classes **java.applet.Applet** and **java.lang.Thread** from Java's API include widely-known examples of *Template Method*.

It is possible to view *Template Method* as a crude technique to emulate pointcuts and advice. The template method performs a role that bears some similarities to pointcuts in that it serves to control the moments when some desired logic executes. The concrete classes that override and concretise the hooks exposed by the template method perform a role similar to that of advice: in both cases, the blocks of code execute reactively, or implicitly. This suggests a *Replace Template Method with Pointcut and Advice* refactoring.

Singleton

The *Singleton* pattern (pages 127-134 of [5]) is one of the patterns that attracted most criticisms. There are many testimonies of the excessive use of singletons, some of which can be found in the refactoring mailing list at Yahoo². Overuse of singletons is troublesome, as it scatters multiple dependency points to the singleton throughout the system. Singletons also create specific problems when creating unit tests for classes that depend on them [10, 3]. The pattern is considered prone to misuse, often by programmers that have yet to fully absorb the fundamental principles of OO and that lean on singletons to write "procedural-style OO code". Such programmers tend to create too many singletons that are really procedural-style global variables.

Such bad uses of *Singleton* can be addressed in a number of ways and a few of them are suggested in [8, 3]. However, in cases, turning the singleton into an aspect may be the appropriate solution. Aspects can have global access to the remaining elements of the system, but can also compose behaviour in a controlled way. Aspects can compose the behaviour equivalent to that provided by the singleton in an *implicit* way, thus avoiding the kind of dependencies that result from scattered calls to singleton logic.

This suggests a *Replace Singleton with Aspect* refactoring. Such a refactoring entails moving to an aspect the singleton logic that is called from multiple places and

² <http://groups.yahoo.com/group/refactoring/>

ensuring that the aspect is able to capture all those points. The logic provided by the singleton is next moved to advices within the aspect that act on those joinpoints.

4. Proposed approach

In order to build a catalogue of refactoring opportunities such as those mentioned above, we propose an approach similar to that taken in [9], which is based on the use of suitable case studies with the desirable characteristics. Such case studies should be OO code bases (e.g., frameworks) rich in patterns and crosscutting concerns. Any refactorings derived from such a study must address various issues, including the one that follow:

- What are the detailed mechanics of the refactorings?
- What are the preconditions for the refactorings? Are there any special situations that prevent its use or do not make it advisable to apply it? For instance, it is likely that some instances of *Singleton* remain desirable: the description of a refactoring opportunity should make clear which uses motivate the use of the refactoring.
- Do the refactorings require the use of other, preparatory, refactorings?
- What impact do the refactorings have on the remaining code base?
- Are there any drawbacks in resorting to aspect-based solutions?

5. Conclusion

In this paper, we argue that in order to identify refactoring opportunities to aspects, we need to go beyond the more superficial manifestations of crosscutting, such as duplicated (and scattered) code. The efforts of well-meaning and experienced programmers and designers may mask such superficial manifestations behind less obvious ones such as elaborate structures and use of design patterns. We argue that patterns comprise a primary candidate to build a deeper base of knowledge for identifying opportunities to evolve legacy systems using AOP. To illustrate, we describe uses of a few patterns whose motivation stems from either the presence of crosscutting effects or limitations of OO relative to AOP. We suggest a few refactorings that address the same problems more effectively.

Acknowledgements

Miguel P. Monteiro is partially supported by project PPC-VM (POSI/CHS/47158/2002) and by FCT under project SOFTAS (POSI/EIA/60189/2004).

References

1. JUnit home page. <http://www.junit.org/>
2. Bracha G., Cook W., *Mixin-Based Inheritance*, ECOOP/OOPSLA 1990, ACM press, pp. 303-311, Ottawa, Canada, October 1990.
3. Feathers, M., *Working Effectively with Legacy Code*, Prentice Hall 2005.
4. Fowler, M. (with contributions by K. Beck, W. Opdyke and D. Roberts), *Refactoring – Improving the Design of Existing Code*, Addison Wesley 2000.
5. Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
6. Hannemann, J., Kiczales, G., *Design Pattern Implementation in Java and AspectJ*, OOPSLA 2002, November 2002.
7. Isberg, W., *Design with pointcuts to avoid pattern density*, online article at Developerworks (AOP@Work series), June 2005.
<http://www-128.ibm.com/developerworks/java/library/j-aopwork7/index.html>
8. Kerievsky, J., *Refactoring to Patterns*, Addison-Wesley, 2004.
9. Monteiro, M. P., *Refactorings to Evolve Object-Oriented Systems with Aspect-Oriented Concepts*. Ph.D. thesis, Universidade do Minho, Portugal, March 2005.
10. Rainsberger, J., *Use your Singletons Wisely*, online article at Developerworks, July 2001. <http://www-128.ibm.com/developerworks/webservices/library/co-single.html>
11. Ungar D., Smith, R., *Self: the power of simplicity*, OOPSLA'87, Orlando, USA, October 1987.
12. Venners, *Patterns and Practice – A Conversation with Erich Gamma, Part IV*, Artima developer, June 2005. http://www.artima.com/lejava/articles/patterns_practice.html