The Search for Aspect-Oriented Refactorings Must Go On (Position paper)

Miguel P. Monteiro[1], João M. Fernandes[2]

[1] Escola Superior de Tecnologia, Instit. Politécnico de Castelo Branco,
Avenida do Empresário 6000-767 Castelo Branco, Portugal
mmonteiro@di.uminho.pt

[2] Departamento de Informática, Universidade do Minho,
Campus de Gualtar, 4710-057 Braga, Portugal
jmf@di.uminho.pt

Abstract. In this paper, we argue that current research in aspect-oriented refactoring can benefit from specific efforts to develop a new coding style appropriate for aspect-orientation. This notion can be captured through catalogues of refactorings and code smells, after which it can provide a basis for tool developers. We successfully derived useful insights from the study of toy examples and think their potential has not been fully explored yet. We will continue to include toy examples in our next research work, hoping to obtain further insights, which will then be tested and developed with larger code bases.

1. Introduction

In order to bring the advantages of Aspect-Oriented Programming (AOP) to existing object-oriented (OO) applications and frameworks, multiple research activities must be undertaken. One of these activities is Aspect-Oriented Refactoring. In this paper we use the term *refactoring* in the sense proposed in the book by Martin Fowler [4] and use Aspect-Oriented Refactoring as including the process of refactoring OO systems in order to turn them into well-formed AOP systems, as well as the process of refactoring existing AOP systems.

The rest of this paper is structured as follows. In section 2 we provide a brief background of research in aspect-oriented refactoring. In section 3 we present our view of the present situation. In section 4 report on our results so far. In section 5 we present some ideas for future work and in section 6 we conclude the paper.

2. Background

Research on Aspect-Oriented Refactoring does not comprise a single monolithic activity. Presently, we identify at least two kinds of different though related tasks in this field:

1. Expand the refactoring space – as we will argue in this paper, this really entails developing a clear idea of well-formed AOP source code.
2. Develop tools supporting the code transformations that yield well-formed AOP code.

Programmers have been performing *ad hoc* behavior-preserving transformations for decades, though they did not use this name. Only at the start of 1990s did it become the subject of formal study. The earliest works were by Opdyke and Johnson [15], which first coined the word *refactoring* and focused on OO frameworks, and by Griswold and Notkin [6], which focused on block-structured imperative programs and functional programs. Some of the early researchers based their work on the assumption that the global restructuring of software systems could not be cost-effective, unless if automated and separated from other qualitatively different maintenance activities [6]. More recently, refactoring became the subject of growing interest due to the advent of agile methodologies, most notably *Extreme Programming* (XP) [2]. Refactoring plays a key role in XP, but advocates of XP regard and use it in a subtly different light. Earlier researchers tended to regard automatic tool support as a fundamental prerequisite for refactoring to be feasible, but the advent of unit testing – another key component of XP – changed that. Testing frameworks such as JUnit [1] made manual refactoring feasible, though still a time consuming task. Naturally, advocates of XP welcome tools for automating code transformation as a very important contribution to enhancing safety and productivity. These views are expressed in Fowler's book, which also acts as a manifesto advocating the use of refactoring in software development and maintenance.

Fowler's book uses Java as the subject language and was published at a time when tool support for refactoring Java programs was not available. Fortunately, tool developers responded to Fowler's challenge, and present users of various integrate development environments can benefit from automated support for many of the refactorings described in Fowler's book.

The catalogues documented in [4] advocated a specific style for writing OO programs. Having a notion of style is important, as it provides programmers with a clear idea of where they are heading to when choosing the next refactoring to apply. In addition, the catalogues reflected past experience of what were the causes of badly written code. For instance, the first chapter of [4] presents the concept of refactoring through an example of Java code written in a procedural (i.e. bad) style, which is subject to a series of restructurings in order to make it well formed according to OO principles. Those restructurings could be made because (1) the programmer could notice the present style was inadequate, (2) he had a clear idea of what would comprise a more adequate style and (3) he knew how to transform the source code and eliminate the inadequacies.

3. Present Situation

When tool developers started developing automatic support for refactoring, notions of good style were already well established, and Fowler's catalogue of refactorings was available. Besides refactorings, the book presented a catalogue of *code smells* – descriptions of symptoms in source code which can be used as signs that its structure is not optimal – and provided hints of which refactorings could be used to remove each of the smells. The present situation regarding AOP is different: at present, we still do not have a fully developed idea of good AOP style, and we do not know what the causes for badly written aspects can be. Because of this, we also do not have an AOP equivalent of Fowler's catalogues of refactorings and code smells. Besides bringing the concept of refactoring to a wider audience and in providing programmers with guidelines on when to refactor and how best to refactor, it also provided a basis on which developers could rely to build automatic tool support. We believe that catalogues of aspect-specific refactorings and smells can bring similar benefits to AOP. Automating those refactorings should follow, as the crosscutting nature of the concerns involved makes AOP refactorings more time consuming and their potential for damage when mistakes are made more serious and widespread.

In many cases, the two aforementioned in Aspect-Oriented Refactoring can and should be carried out by independent research groups. We believe their efforts will be mutually beneficial. However, in some cases task 1 is a prerequisite for task 2. One example is the work on programming laws for AspectJ by Cole and Borba [3] which is based on existing refactorings proposed by independent authors [8]. Ideally, researchers and developers of tool support would always be able to rely on already available results, but that requires that specific efforts focusing on task 1 do not stop.

4. Earlier Results

We focused our work exclusively on task 1. Our aim was to find new refactorings for AspectJ [13, 14], contributing to expand the current set of refactorings for AOP [10, 8]. We approached Java programs as if they were AspectJ programs written in a bad style (i.e. Java code as "smelly" AspectJ code). We selected some appropriate code bases as case studies and investigated what code transformations would turn them into examples of good AOP style. Generally, we considered good style to be whatever coding practices yield code easier to maintain and evolve. We reasoned that whenever there are multiple ways to achieve some result or effect, the way that causes the least problems to present and future programmers should be considered the one in the best style.

We developed a collection of refactorings – presented in [13] and [14] using a presentation style and level of detail similar to [4] and [9] – and proposed few novel code smells [14], which these refactorings aim to remove. Though we developed the refactorings to be performed manually, we believe they can help developers of tool support in identifying useful material on which they can focus their work.

The experiments we undertook made us aware of the influence of the decentralized nature of the OO paradigm in shaping many designs, and how different these designs can sometimes be from the optimal AOP design for the same problem. This is partly due to the ability of AOP to modularize concerns that affect multiple objects, but is also due to the efforts of OO designers to compensate for the crosscutting effects. We can already picture that one of the symptoms of "designing for crosscutting" in existing OO systems includes the use of various patterns (e.g. Chain of Responsibility [5] and Observer). More specifically, we mean the traditional OO implementations of the patterns. As was shown in [7], some OO patterns are completely subsumed by the AOP implementations (i.e. they "disappear"). For instance, one could argue that the AspectJ implementation of Observer is not a pattern any more, just an aspect like any other.

These efforts can result in more complex designs than otherwise would be the case, but this complexity is no longer needed after the concern is extracted to an aspect (our work has not, until now, covered cases involving multiple aspects and associated issues). This fact should be taken into account in reengineering processes involving the extraction of aspects from an existing OO code base: further refactorings must be performed on the extracted aspect (something that is easier to do after modularization) and possibly on the remaining code base as well.

We first became aware of these issues when we were testing refactorings for extracting the various elements of a Java implementation of Observer design pattern [5]. Several of the extraction refactorings were based on the mechanism of inter-type declarations (a.k.a. introductions), like those proposed by other authors [8]. When we compared the aspect resulting from the extraction with the reusable AspectJ implementation of the pattern by Hannemann and Kiczales [7], we noticed they were completely different. The internal structure of the extracted aspect did not look right, particularly when compared to the one from [7]. It had duplicated introductions of the same members to multiple target types and did not provide the same flexibility as the solution presented in [7]. The extracted aspect was not reusable and did not enable an object to participate in multiple, separate, observing relationships. Its structure was not taking advantage of modularization, as was the case the one from [7]. Further refactoring of the aspect's internal structure was clearly needed, which led us to develop an entire subgroup of refactorings[1] [14]. In [12] we present a complete refactoring process, which documents the differences in detail.

5. Discussion and Future Work

When selecting code bases to perform our experiments, we favored real applications, as we believed these would provide us with richer insights. For his reason, we selected a Java framework for workflow applications as our first case study. However, a collection of "toy examples" gave us rich insights – the design pattern imple-

[1] The refactorings presented in [14] deal with improving the internal structure of the extracted aspect. Improving the structure of the remaining code base is left out to future work.

mentations in Java and AspectJ developed by Hannemann and Kiczales [7]. To a certain extent, this reflects the relative immaturity of the field.

These are exciting times, when we can learn new things from such simple examples. The potential of toy examples has not been fully explored yet – even the examples from [7] still have the potential to yield new insights. Naturally, this does not preclude using "real" cases, but we now think it is makes sense to start exploring smaller examples to obtain the earlier results. Only afterwards do we plan explore larger examples, with the benefit of the hints and insights derived from the toy examples. The earliest results will also be tested and matured in the larger cases. For instance, we believe the refactorings derived from the study of Observer and other patterns can be generalized to many other cases in which aspects act as controllers linking independent sets of classes.

We believe many useful insights can be derived by studying patterns whose OO solutions suffer from crosscutting, and analyze common OO ways of dealing with them. The analysis can be done (1) by looking the description of the patterns themselves, (2) surveying current OO implementations, and (3) by analyzing the motivation to refactor *to* or *towards* them [9]. We also plan to broaden the pattern space, namely with patterns related to concurrency and synchronization [11]. We hope to derive from this analysis a set of indicators to help detecting latent aspects (i.e. AOP code smells).

6. Conclusion

In this paper, we argued that current research in aspect-oriented refactoring could benefit from studies aiming specifically to develop a new AOP specific coding style. Fowler [4] showed that notions of good style can be captured through catalogues of refactorings and code smells.

We're undertaking a research effort to develop catalogues of code smells an refactorings for the AspectJ language. We hope our results will provide a useful basis to tool developers.

The relative immaturity of this research field meant that useful insights could be derived even from toy examples. We think toy examples will continue to have some potential to yield interesting results, which can form a basis for experiments on larger and more complex case studies. For this reason, we will continue to include toy examples in our next research work, with the aim of obtaining new insights. The results thus obtained will then be tested and developed on larger systems.

Acknowledgements

Miguel P. Monteiro is partially supported by *PRODEP III (Medida 5 – Acção 5.3 – Eixo 3 – Formação Avançada de Docentes do Ensino Superior)*. Miguel P. Monteiro and João M. Fernandes are partially supported by project *PPC-VM (POSI/CHS/47158/2002)*.

References

1. JUnit home page. http://www.junit.org/
2. Beck, K., *Extreme Programming Explained: Embrace Change*, Addison-Wesley 2000.
3. Cole, L., Borba, P., *Deriving Refactorings for AspectJ*, to be presented in AOSD'2005, Chicago, USA, March 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. Griswold, W., Notkin, D. *Automated assistance for program restructuring*. ACM Transactions on Software Engineering and Methodology, July 1993.
7. Hannemann, J., Kiczales, G., *Design Pattern Implementation in Java and AspectJ*, OOPSLA 2002, November 2002.
8. Hanenberg, S., Oberschulte, C., Unland, R., *Refactoring of Aspect-Oriented Software*, Net.ObjectDays 2003, Erfurt, Germany, September 2003.
9. Kerievsky, J., *Refactoring to Patterns*, Addison-Wesley, 2004.
10. Laddad, R., *Aspect-Oriented Refactoring*, parts 1 and 2, The Server Side, 2003. http://www.theserverside.com/
11. Lea, D., *Concurrent Programming in Java: Design Principles and Patterns*, second edition. Addison-Wesley 1999.
12. Monteiro, M. P., *Refactoring a Java Code Base to AspectJ – An Ilustrative Example*, Technical Report UM-DI-GECSD-200403, Universidade do Minho, December 2004. Available at http://www.di.uminho.pt/~jmf/PUBLI/papers/2004-TR-03.pdf
13. Monteiro, M. P., Fernandes, J. M., *Object-to-Aspect Refactorings for Feature Extraction*, industry paper presented at AOSD'2004, UK, Lancaster, March 2004. Available at http://aosd.net/ 2004/archive/Monteiro.pdf
14. Monteiro, M. P., Fernandes, J. M., *Towards a Catalogue of Aspect-Oriented Refactorings*, to be presented AOSD'2005, Chicago, USA, March 2005.
15. Opdyke, W., Johnson, R. *Refactoring: An aid in designing application frameworks and evolving object-oriented systems*. In Proceedings of SOOPPA '90: Symposium on Object-Oriented Programming Emphasizing Practical Applications, September 1990.