Some Thoughts On Refactoring Objects to Aspects

Miguel Pessoa Monteiro[1] and
João Miguel Fernandes[2]

[1] Escola Superior de Tecnologia de Castelo Branco,
Instituto Politécnico de Castelo Branco,
Av. do Empresário 6000-767 Castelo Branco, Portugal
mpm@est.ipcb.pt
[2] Departamento de Informática, Universidade do Minho
Campus de Gualtar, 4710-057 Braga, Portugal
jmf@di.uminho.pt

Abstract. The prospect of aspect-orientation receiving widespread acceptance and adoption in the near future begs the question of how to deal with a large base of object-oriented legacy code. We propose to investigate refactoring techniques for restructuring object-oriented source code so that it can leverage the mechanisms of aspect-orientation in order to become easier to adapt, extend and evolve. Our approach is to adopt an object-oriented framework in the area of workflow as a non-trivial case study. We plan to analyse its source code, develop an aspect-oriented version through the progressive use of manual refactorings, and build a catalogue of refactoring operations based on the experience gained through the process.

1. Introduction

The work presented here refers to an ongoing Ph.D. project. An early version of the project proposal covered in this paper was presented as a poster at the *student stravaganza* that took place at the AOSD'2003 in Boston, March 2003. The rest of this paper is structured as follows. In section 2 we present the general problem. In section 3 we present refactoring and aspect-oriented programming (AOP) as two possible paths towards a solution. In section 4 we present the research project, which covers both refactoring and AOP. In section 5 we present its work plan and in section 6 we present some preliminary results. Finally in section 7 we conclude the paper.

2. The Problem

Ever since the beginning of software engineering as an independent activity one of its main goals has been to bridge the gap between the hardware and the human mind. The programming models developed in the latest decades show a path towards the way the human mind works, away from hardware's "low level". All efforts to make program-

ming easier are ultimately efforts to find a model suitable for the human mind, and translate instances of such a model to the imperative code that machines understand.
It is important to understand how the human mind works, and take into account its limitations. We clearly have limits relative to the quantity of details we can cope with at a given moment. We cannot concentrate on more than a few subjects, or concerns, at a time. We need to break, or decompose, systems and problems into smaller subsystems and sub problems, in order to concentrate on each sub problem by turns. In short, our minds need *abstraction*. This is the fundamental idea behind the works of Parnas [17] and Dijkstra [9] when they suggested *separation of concerns* as a technique for problem solving.
Unfortunately, current tools and languages fail to provide a complete and effective support for full separation of all concerns. Some of the system's concerns cannot be kept separate in their own units of modularity (e.g. methods or classes), and this results in several negative properties, such as *code scattering* and *code tangling* [13], which increase the difficulty in understanding, adapting and reusing code.
Ideally, the decomposition of a system into separate concerns would directly lead to the optimal structure of the intended system. In practice, however, this is not possible because the supporting tools do not provide the necessary composition mechanisms. Those mechanisms determine what we can separate, because we will not separate what we cannot later compose.
Traditional techniques for software development such as object-oriented programming, design patterns and frameworks all suffer from these limitations. One of the most serious consequences is the *preplanning problem* [8] – architectures that result from applying such techniques are usually more flexible for certain kinds of changes, but they also result more difficult to adapt to other, unplanned kinds – they become "brittle". This forces system developers to anticipate all changes that might be required of the system during its lifetime, a difficult task at the best of times. Ultimately it is impossible, due to the continuous change in software requirements and environments.

3. Paths to a Possible Solution

There are several different approaches that attempt to deal with the aforementioned problems, including refactoring and AOP, which are briefly surveyed in the next 2 sections.

3.1. Refactoring

Refactoring [10][16][3] is a technique that attempts to deal with the apparent brittleness of software and the preplanning problem. A *refactoring* is a restructuring of the source code whose purpose is to obtain code that is better organized and easier to maintain, adapt and extend, while preserving its functionality. The idea of refactoring is to improve the design inherent in the code by changing the code in order to make it

correspond to a better design, a procedure that reverses the traditional order of design first, code next.
Refactorings can be performed either manually or automatically. In [10] we find a catalogue of 72 named refactoring operations meant to be performed in a manual but disciplined way. This catalogue in effect comprises a pattern language for manual refactoring. Opdyke's Ph.D. thesis [16] is generally considered the first major written work on the subject of automatic refactoring of object-oriented (OO) programs.
Refactorings are typically performed through small steps, often with tests performed in between, to make sure that no errors are introduced (cf. first chapter of [10]). It is considered prudent to perform any restructuring through a sequence of small refactorings rather than a few large ones, as large restructurings increase the likelihood of introducing errors. Larger refactorings are usually decomposed into several small ones. Sometimes a specific refactoring may require several others to be applied prior to it, before the code is ripe. For instance, when referring to the *Extract Method* refactoring (cf. p.111 of [10]) Fowler et al. mention awkward situations that may require the previous use of 2 other refactorings before it can be safely applied.

3.2. Aspect-Oriented Programming

As a solution to crosscutting and other related problems Kiczales et al. propose a new programming technique they dubbed aspect-oriented (AO) [13], using the term *aspect* to refer to the modular implementation of a crosscutting concern.
The central concept of AOP and its main AO language – AspectJ [1] – are the *joinpoints* – points in the dynamic execution of a program that aspects can intercept, and in which extra sections of code called *advice* can be executed before, after, or even instead of the original code.
AspectJ provides a new kind of language construct, the *pointcut designators* (PCDs), through which sets of crosscutting joinpoints can be defined. AspectJ's rich set of PCDs effectively comprises a domain-specific language for meta-programming. AspectJ provides another new mechanism – *inter-type declarations*, also known as *introductions*. This tends to be overlooked next to PCDs and advice but plays a fundamental role in the encapsulation of crosscutting concerns. Its purpose is to declare additional fields and methods into classes and to change their inheritance hierarchies (with some limitations). Access modifiers of inter-type members are scoped with regard to the aspect that declares them, not the target type. Therefore when class members declared by the aspect are classified as *private*, they are private to the aspect, and the only place in the source code in which those members can be used is the aspect, ensuring its completely modular structure.
The experience gained in the last few years gives cause to believe that AOP indeed fulfils its promise of improving on the OO model by enabling a stronger separation of concerns. There is the prospect in the near future of a widespread adoption of the concepts from aspect-orientation, which begs the question of how to deal with a large legacy of OO code in an AO world. Experience with refactoring (of OO software) in the last half-decade suggests that refactoring techniques may provide an adequate answer in a significant number of cases.

Among AO languages, AspectJ has the greater following [18]. This is probably due to the fact that AspectJ enjoys a more widespread support of tools [11][12][7][2], including several Integrated Development Environments (IDEs). Such tool support is indispensable to work with software systems of realistic dimensions. The fact that AspectJ is a backward-compatible extension to Java makes it logical to use applications written in Java for research on refactoring OO software to the new paradigm.

4. Project Proposal

We propose to investigate refactoring techniques for converting OO legacy code into AO code. Fowler et al. [10] present the concept of refactoring using a case study written in bad-style Java, which is then subject to a series of restructurings in order to make it well-formed. The example was written in a procedural style, which is indeed inadequate for an OO language like Java. This approach suggests that a catalogue of object-to-aspect refactorings could be built, presenting a set of code transformations from both the procedural and OO styles to a new, AO style.

This in turn leads us to a question that has yet to receive a complete and totally satisfactory answer: what is a "good aspect-oriented style"? The refactorings presented in [10] were based on the assumption that there is a clear idea of what comprises well-formed OO code. In fact, Fowler et al. propose a programming style that stems from the ideas of *Extreme Programming* [6], which regard a system's source code as primarily a communication mechanism between people, not computers. Therefore this style places great value to code adequate for the capabilities of human programmers.

The notion of good AOP style has not yet been fully developed, however. Therefore we expect that the task ahead will relate as much with determining what is well-formed AOP code as with building the catalogue of refactorings that will enable us to obtain it from existing OO code.

We do not intend to cover techniques for automatic support for refactoring operations, rather we aim to pinpoint and survey the operations themselves, as performed manually. Pertinent issues include the right order of refactoring steps, which steps are the most adequate for typical situations, which preconditions must be met for each of the steps, and how the structure of the legacy code may influence the refactorings. Also, this project is not expected to cover a separate problem generally known as *aspect mining*. That relates to the task of identifying and locating the code fragments related to a given concern. For the purposes of this project we intend to rely on available documentation and the information provided by the original creators and developers do code bases used as case studies.

4.1. WorkSCo as a Case Study

We intend to tackle this task through the study of several case studies. At least one of these must be non-trivial in both size and complexity, and must be independent of the author (neither created by him nor under his specifications).

For the non-trivial case study we selected WorkSCo, an OO framework for workflow management systems [5] initially designed and developed by Manolescu [14], using Smalltalk, and reimplemented with Java [19] at the ESW [4] group at INESC-ID, Lisbon, Portugal. Workflow applications are a good example of modern software architectures that stress to the limit the capabilities for separation of concerns of current software engineering technologies. The requirements of workflow applications cover a multitude of crosscutting concerns, as well as a very flexible support for evolution.
The design of WorkSCo relies on a considerable number of design patterns, most of which are intended to achieve particular separations of concerns. The framework as implemented at ESW/INESC-ID introduces a few structural deviations from Manolescu's original design in order to achieve more ambitious separations.
WorkSCo can be regarded an example of the maximum results of separation of concerns that can be obtained using conventional approaches such as object-oriented programming and design patterns. That makes it an interesting case study both as a starting point and as a benchmark for research work in this field of AOP.
We propose to develop an AO version of WorkSCo and use the insights and experience gained in performing that task as the basis for the initial development of a catalogue of refactoring operations.

4.2. The Fragile Base Code Problem

Extension of the present knowledge of refactoring techniques to include AOP presents a number of issues, which can be broadly divided into two groups: (1) study of how to make the existing OO refactoring catalogue take aspects into account, and (2) identify and characterize the new refactorings specific to the AOP mechanisms.
Both are deserving of attention. In our view the strongest motivation for the first group of tasks lies in the fact that aspects are extremely fragile relative to the structure of the base code, so much so that we can talk of a new edition of the *Fragile Base Class* problem [15]. Even the most fundamental refactoring, *Rename Method* (p.273 of [10]) can potentially break aspect code, since AspectJ's pointcut language is based mainly on lexical elements.
The Fragile Base Code Problem is not our main focus, however. Rather our work will concentrate on the refactoring operations that contribute to adding an aspect dimension to existing OO systems, by transferring the crosscutting elements of the implementation to aspects. We envision operations such as *Move Field from Class to Inter-Type Introduction*, *Move Method from Class to Inter-Type Introduction*, *Extract Fragment into Advice*, and *Extract Initialisation to Constructor Advice*[1].

[1] We are currently working on a paper in which we present the detailed mechanics of these refactorings.

5. Work Plan

We plan to undertake the research in the following incremental phases:
(1) Analysis of the source code of WorkSCo's core classes, in order to identify and catalogue the various concerns present in the code. We will also rely on the available documentation [14][19].
(2) Build an AO prototype of WorkSCo's core using AspectJ. This task will be based on a snapshot version of WorkSCo's code and will be carried out through a series of refactorings over that code base, rather than by developing a new version from the beginning.
(3) Build a catalogue of object-to-aspect refactoring operations as work on (2) progresses. It is hoped that the experience gained in (2) can be leveraged to provide the basis for a pattern language of object-to-aspect refactorings. Work on the refactoring of WorkSCo is expected to both provide a basis and a validation case.
(4) At a final stage assess the feasibility of automating a subset of the refactorings.

6. Preliminary Results

Here we present a few early findings from our work, relative to issues of style, and a problem that can make itself felt in some cases that involve the composition of several aspects.

6.1. Aspect Dependent versus Aspect Friendly Base Code

The experience gained through our earlier efforts suggests that it is sometimes necessary to refactor the base code in order to expose the necessary joinpoints to AspectJ. That shouldn't mean, however, that the base code becomes dependent on the aspect code. We make the distinction between such a type of base code, which we call *aspect dependent*, and *aspect friendly* base code, which is base code amenable to the proper quantification, without betraying dependencies to any specific aspect. While the former type of code is clearly undesirable, we believe that there are no negative consequences in refactoring the base code in order to make it more aspect friendly. Actually, our initial efforts suggested that the more well-formed is the base code, the less need is felt for any such refactorings. We also feel that when such refactorings are required, they tend to be the kind of refactoring that would be recommended in [10] anyway. Examples of such refactorings are *Extract Method* (p.110 of [10]) and *Replace Method with Method Object* (p.135 of [10]).

6.2. The Role of Privileged Aspects

Although AspectJ is able to bypass Java's visibility rules through the use of *privileged aspects*, we believe that such a practice should be avoided, for the sake of preserving

encapsulation. However, our preliminary results suggest that a temporary use of privileged aspects during certain refactorings can be of great help when non-public elements of the implementation are moved across scope units such as classes and packages. Privileged aspects enable the source code to stay compilable, and therefore testable, while such movements are carried out.

6.3. Packages versus Directories

We also felt that it is good AOP style to keep aspects and classes in separate directories, even if scope and visibility rules require them to be kept in the same package.
Many people interpret Java's scope and visibility rules as enforcing a 1:1 relationship between directories and packages, to the point that *package* and *directory* became synonyms. However, that is not strictly the case, and it is possible to associate more than one directory to a single package. One way is by registering several entries in the CLASSPATH environment variable, and keeping several directories with the same name, one under each entry. Classes inside these like-named directories must all declare the package's common name and its instances behave as if they all were enclosed within the same package (they have access to each other's package-protected members).
Some developers find this technique useful to separately manage files of different natures, such as JUnit tests and generated source files. We feel that aspects comprise another case of source files that are best managed in separate directories, even when visibility rules demand they are placed within the same package.

6.4. The Constructor Explosion Problem

There are cases when a given functionality has its implementation code tightly connected to other implementation elements of the base code and scattered over several points, but is not needed in all configurations. Such functionality would ideally be made pluggable and unpluggable, and so its code makes a good candidate for extraction into an aspect.
In many cases the extraction of all the code relative to this kind of functionality requires the movement of fields, which must become inter-type declarations. In some cases those fields are initialised by constructor(s) of the original host class, which receive their initial values through parameters.
In order for all the code to be encapsulated within an aspect, it is necessary to transfer the initialisation code to the aspect as well. However, when the constructor's signature receive parameters specific to the concern to extract, this presents a problem.
These cases can be dealt with by preserving the original constructor's signature in the aspect, for the sake of the client code that depended on it, and by implementing another constructor signature (assuming one did not already exist), devoid of any parameters related to the fields to be moved. The new constructor should be placed in the host class, and the old version of the constructor should be refactored so that it would not include any initialisation code not directly related to the extracted concern.

In most cases the inter-type constructor should call the new, simpler version. This way old client code would still see the old interface (now declared in the aspect) and would not break. This solution also opens the way for simpler, cleaner client code whenever it does not need the extracted functionality.
However, this brings us a question: what happens when we are in the presence of not one, but several aspects, each with its own extra set of parameters to add to the class's constructor signature?
Each of the aspects may export a partial constructor signature related to its respective concern, but clients that require the complete constructor signature that results from the combined functionality won't find it anywhere. AspectJ does not provide a mechanism that directly augments the signature of constructors.
A possible solution would be the creation of a *manager aspect* that deals with issues related to the composition of several concerns. Such an aspect can declare and export the missing compound signature. However, for each different combination of more than one of these aspects, a different constructor signature will have to be implemented, resulting in a *constructor explosion* problem.
In our view, this problem stems not from AspectJ, but from Java's brittle constructor mechanism. However, the problem with Java's constructors becomes more exposed when using AspectJ, due to its greater capability to keep each concern separate.
Java's constructor mechanism is unsatisfactory in several ways. For instance, it does not allow for two constructors with identical parameter lists, for the constructor's names are preset for each class. This causes problems in some situations, for example when we're dealing with point objects and we would like to have constructors for both polar and cartesian representations. Under Java's rules we can't, because both signatures require the same number of numeric parameters. We think that ideally Java would provide a solution to the constructor explosion problem in a future version, if it adopted a more flexible approach to constructors.

7. Conclusion

In this paper we presented AOP and refactoring as two approaches that give a contribution in the quest of bringing the structure of software systems closer to the human mind. We raised the issue of how to deal with a large base of OO legacy code if AOP becomes the dominant paradigm, and proposed an approach using refactoring.
We identified some of the current issues and obstacles and presented the goals of an ongoing Ph.D. project. These include the construction of a catalogue of refactoring operations that can help programmers to convert OO systems into AO ones. Finally, we presented some early findings, related to issues of source code style, and reported on a problem we identified, which we termed the *Constructor Explosion Problem*.

8. References

[1] AspectJ home page http://www.eclipse.org/aspectj/

[2] AJDT project. http://www.eclipse.org/ajdt/.

[3] Refactoring Home Page, http://www.refactoring.com/

[4] The Software Engineering Group home page. http://www.esw.inesc.pt/

[5] The Workflow Management Coalition, http://www.wfmc.org/

[6] K. Beck, “Extreme Programming Explained”, Addison Wesley, 2000.

[7] A. Clement, A. Colyer, M. Kersten, “Aspect-Oriented Programming with AJDT”, workshop on Analysis of Aspect-Oriented Software, ECOOP'2003.

[8] K. Czarnecki; U. W. Eisenecker. “Generative Programming - Methods, Tools, and Applications”, Addison Wesley, 2000.

[9] E. Dijsktra, “A Discipline of Programming”, Prentice Hall, 1976.

[10] M. Fowler (with contributions by K. Beck, W. Opdyke and D. Roberts), “Refactoring – Improving the Design of Existing Code”, Addison Wesley 2000.

[11] W. Griswold, Y. Kato, J. Yuan, "Aspect Browser: Tool Support for Managing Dispersed Aspects", Technical Report CS99-0640, Department of Computer Science and Engineering, University of California, San Diego, December 1999.

[12] M. Kersten, "AO Tools: State of the (AspectJ) Art and Open Problems", workshop on Tools for Aspect-Oriented Software Development, OOPSLA'2002, November 2002.

[13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, J. Irwin, “Aspect-Oriented Programming”, ECOOP’97, June 1997.

[14] D. Manolescu, "A Micro Workflow Architecture Supporting Compositional Object-Oriented Software Development", Ph.D. thesis, University of Illinois at Urbana-Champaign, 2001.

[15] L. Mikhajlov, E. Sekerinski, "A Study of The Fragile Base Class Problem", ECOOP'98, July 1998.

[16] W. F. Opdyke, "Refactoring Object-Oriented Frameworks", Ph.D. thesis, University of Illinois, 1992.

[17] D. L. Parnas. “On the criteria to be used in decomposing systems into modules”, pp. 1053-1059 of Communications of the ACM, December 1972.

[18] D. F. Savarese, "Java's Continuing Evolution", November 2002 of Java Pro magazine. Also available at http://www.fawcette.com/javapro/2002_11/magazine/features/dsavarese/default.asp

[19] A. Silva, S. Fernandes and P. Vieira. Micro-Workflow Architecture Extension to Support Evolution, COMBINE Document 20-c, Version 0.2, August 2002.