Refactoring Object-Oriented Systems with Aspect-Oriented Concepts

Ph.D. progress report, 10th December 2004
Departamento de Informática, Universidade do Minho

Ph.D. student:
Miguel Pessoa Monteiro
mmonteiro@di.uminho.pt

Ph.D. supervisor:
João Miguel Fernandes
jmf@di.uminho.pt

ABSTRACT

This report presents the current status of the first author's Ph.D. project. The project's aims were to research a new source code style suitable for aspect-oriented programming, to be expressed through a pattern language of refactorings and code smells. The approach taken was to perform refactoring experiments on suitable code bases in order to derive useful insights. The project's main contributions are a reappraisal of traditional object-oriented smells in the light of aspect-orientation, the proposal for several novel code smells, complemented with a collection of refactorings for aspect-oriented source code capable of removing those smells from existing object-oriented systems.

1. INTRODUCTION

In the 1st of April of 2002 this first author of this report started working on a Ph.D. project under the supervision of the second author. The first author was granted a leave of 36 months, funded by *PRODEP III (Medida 5 – Acção 5.3 – Eixo 3 – Formação Avançada de Docentes do Ensino Superior)*. The leave ends at 31st March 2005. This report presents the project's current status as it is nearing its deadline.

The Ph.D. project described here tackles one of the problems found in the area of software engineering, stemming from limitations in object-oriented programming (OOP), which is the current dominant paradigm. In the end of the 1980s and beginning of the 1990s symptoms of limitations in this paradigm became increasingly prominent, which motivated various research fields, including the ones mentioned in the next section.

This report is structured as follows: in section 2 we present a brief overview of some of the problems stemming from limitations in OOP. This provides the background and motivation for this Ph.D. project. In section 3 we present refactoring and aspect-oriented programming as two trends in current software engineering that contribute to ameliorate the problems mentioned in the previous section. In section 4 we present the Ph.D. project's aims. In section 5 we describe the approach taken. In section 6 we present the project's main contributions and in section 7 we survey related work. We conclude the report by presenting the thesis claim in section 8.

2. BACKGROUND TO THE PROBLEM

Object-Oriented Programming [48][70][40][63] is the current dominant programming paradigm in software engineering, so much so that several popular software (s/w) development techniques are often discussed in terms of the concepts of object technology, even though they are independent. Examples are components [66][55][71], design patterns [30][18][59], frameworks [37] and refactoring [12][29][56]. One key aim of all these techniques is to attain **separation of concerns**[1] [58][24] as a way to make s/w reusable and its evolution simple. A concern is basically any issue in a system's design potentially deserving the attention of the programmer at a given time during the design and development. Separation of concerns is the ability to keep each and every concern in its own unit of modularity, for the sake of its own consistency and to ease the human programmer's task of reasoning with it. Over two decades of experience with OOP led to the conclusion that, although OOP enabled significant developments in s/w engineering, it still failed to achieve a full separation of concerns.

OOP is essentially a decentralised paradigm, with the various bits of functionality being placed in different objects. It copes less well with concerns affecting multiple objects at the same time, which Kiczales et al called **crosscutting concerns** [42]. This phenomenon occurs because OOP, like previous programming paradigms, supports one single decomposition criterion, in its case the class decomposition unit. Concerns that do not align well with this decomposition tend to crosscut existing units of modularity (e.g. classes and methods), resulting in several negative properties, including **code scattering** and **code tangling** [42]: code related to such concerns tends to be scattered throughout multiple modules, intertwined with code relating to other concerns. These properties increase the difficulty in understanding, adapting and reusing program source code. Unfortunately, the recent evolution of modern s/w is leading to an increasing prominence of crosscutting concerns. Examples include most, if not all, of the services provided by the so-called "middleware", including logging, synchronisation and coordination, security and authentication, persistence and storage management, transaction support, administration, performance and resource pooling.

These limitations started to be noticed at the end of the 1980s, and are the root cause of problems such as the so-called inheritance anomalies [46], which motivated an enormous quantity of research efforts and publications during the 1990s, e.g. [14][47][15][45][62][35].

A problem related to the ones mentioned above is the **preplanning problem** [21], [22]: s/w architectures that were

[1] Parnas is generally credited for the introduction of the concept of separation of concerns, when proposing modular programming [58] as a better way to structure program code. The term "separation of concerns" was first coined by Dijkstra in [24].

designed with a given set of changes in mind can cope flexibly with those kinds of changes, but they usually result more difficult to adapt to other kinds. This is serious, because (1) it may not be possible to anticipate all changes required during a system's lifetime, (2) requirements tend to change over time and (3) product families require different combinations of concerns in its various member products. One traditional approach to this problem is trying to anticipate all future requirements, and provide for them in the design, but this usually leads to even more complex and inflexible structures, and sometimes it turns out that some of the anticipated requirements are not needed [16].

3. STATUS OF CURRENT RESEARCH

There are various research areas aiming to solve the aforementioned problems, among them refactoring and aspect-oriented programming (AOP). This Ph.D. project aims to strengthen the link between the two.

3.1. Refactoring

Refactoring [12][29][56] is one of the key techniques of *extreme programming* [16] and attempts to deal with the preplanning problem. A refactoring is a transformation of the source code with the purpose of obtaining code that is better organised and easier to maintain, adapt and extend, while preserving its functionality, or "externally observable behaviour". The aim of refactoring is to improve the design inherent in the source code by changing it in order to make it correspond to a better design, a procedure that reverses the traditional order of design first, code next. The programmers are assisted in their task of detecting ill-formed code by a catalogue of *code smells*, each of which specifies a symptom that is a sign of possibly inadequate code structures.

Refactorings can be performed either manually or automatically. In [29] we find a catalogue of 72 named refactorings meant to be performed in a manual but disciplined way. It is generally agreed that automatic support for refactoring [56] should be the ultimate goal, as it offers a much stronger guarantee that no bugs are introduced in the refactoring process. Refactorings are typically performed through small steps, often with tests performed in between, to prevent the eventual introduction of defects (cf. chap.1 of [29]). It is also considered prudent to perform a given restructuring with 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 given refactoring may require several others to be made first before the code is ripe for it to be applied.

3.2. Aspect-Oriented Programming

AOP is one of several research areas falling under the umbrella terms *Advanced Separation of Concerns* (ASoC) or *Aspect-Oriented Software Development* (AOSD) [3][26], which include other research lines, namely Composition Filters [4][17][13], Subject-Oriented Programming [34], Multi-dimensional Separation of Concerns [11][67][57], Adaptive Programming [8][44] and Feature-Oriented Programming [10].

Kiczales et al. proposed AOP [42] as a new programming technique capable of solving the problems related to crosscutting. They used the term *aspect* to refer to the modular implementation of a crosscutting concern. AOP's main aspect-oriented (AO) language – AspectJ [2][41] provides the same mechanisms as found in Java classes and a few novel ones. AspectJ's central concept are the *joinpoints*: interesting events in the execution of a program that aspects can intercept, and in which extra sections of code called *advice* can be executed before, after, or instead of the original event. AspectJ provides a new language construct, the *pointcut designator* (PCD), through which programmers can *quantify* over programs, i.e. specify the set of joinpoints necessary to modify, extend or delete the behaviour associated with the joinpoint. AspectJ's rich set of PCDs effectively comprises a domain-specific language for one kind of meta-programming, quantification [28]. In addition, aspects can declare their own state as well as declare additional state in existing classes, through *inter-type declarations*. When that state is declared private it is private to the aspect and the only place in the source code in which those members can be used is the aspect, ensuring its modularity. Among AO languages, AspectJ has the largest community of users, due to the greater maturity of its tool support [31][39][19][1], including several Integrated Development Environments (IDEs) [1][4][5][6][64]. Such tool support is indispensable to work with s/w systems of realistic dimensions.

3.3. Aspect-Oriented Refactoring

AOP's steady progress from "bleeding edge" research field to mainstream technology [60] brings forward the problem of how to deal with large number of object-oriented (OO) legacy code bases. Experience with refactoring of OO software in the last half-decade suggests that refactoring techniques have the potential to bring the concepts and mechanisms of aspect-orientation to existing OO frameworks and applications.

In this Ph.D. project we research refactoring techniques for AO code. We are not considering mechanisms for automatic support, but rather aiming to pinpoint and characterise the operations as performed manually. Pertinent issues include finding the most useful transformations, their mechanics, which preconditions must be met prior to each refactoring, and how the structure of the legacy code may influence choice of the next refactoring to apply. We chose to present the refactorings in a style similar to the one used in [29] and [38], including its detailed descriptions of mechanics and the use of code examples.

We adopted AspectJ, the most mature AOP language available, as the main tool for this study. The fact that AspectJ is a backwards-compatible extension of Java, as well as the present availability of a large and rich base of Java code that can potentially benefit from the superior composition capabilities of AspectJ, motivated us to initially focus on transformations of Java to AspectJ, in particular the extraction of concerns into aspects [50][53].

4. THE SPECIFIC PROBLEM

We believe there are several hurdles in need to be addressed so that refactoring techniques can be used in AO software in an effective and widespread way.

The first hurdle is the present lack of a fully developed idea of what comprises a good AOP style. This is an important issue, for a clear idea of style is a fundamental prerequisite for the use of refactoring. Programmers need a clear idea of to where they are heading in order to choose the next refactoring to apply. For instance, Fowler et al. [29] present 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. This knowledge is represented through a catalogue of 22 *code smells* [29], compounded by a catalogue of 72 *refactorings* through which those smells can be removed from existing code. This concept of good style became one of the key components of *Extreme Programming* [16], which regards a system's source code as primarily a communication mechanism between people rather than computers.

A second hurdle – both a cause and a consequence of the first – is the current lack of an AOP equivalent of the catalogue of OO refactorings presented in [29]. This catalogue proved very useful in bringing the concept of refactoring to a wider audience and in providing programmers with guidelines on when to refactor and how best to refactor. Our work is based on the assumption that AOP would equally benefit from its specific catalogues of smells and refactorings, helping programmers identify the situations in the source code that could be improved with aspects, and guiding them through the transformation processes.

A third hurdle – caused by the previous two – is the lack of tool support for AOP constructs in current IDEs. The catalogue presented in [29] provided a basis on which developers could rely to build automatic tool support: a similar catalogue for AOP is likely to bring similar benefits to current tool developers. However, tool developers won't be able to provide adequate support to refactoring operations unless they have a prior notion of AOP style, and a clear idea of which refactorings are worthy of their development efforts.

5. THE APPROACH

We've taken the approach of using refactoring experiments based on case studies, as a vehicle for gaining the necessary insights. The case studies we used are Java code bases with the appropriate structural characteristics. We approached those Java code bases as bad-style or "smelly" AspectJ code, and searched for the kinds of refactorings that would be effective in removing those smells. Our first case study was WorkSCo [27], a real application in the area of workflow and whose study yielded our first results [50]. Our second case study were the code examples (version 1.1) presented in [33], comprising the implementations of the 23 Gang-of-Four (GoF) design patterns [30] in both Java and AspectJ. The 23 GoF patterns illustrate a variety of design and structural issues and situations that would be hard to find in a single code base (except possibly in some large and complex ones). The implementations of the GoF patterns effectively comprise a microcosm of many possible systems. They proved to be a richer source of insights than we probably would get from traditional OO frameworks, without the need to analyze large code bases or learn domain-specific concepts.

The AspectJ implementations presented in [33] are currently one of the nearest things to examples of good AOP design, presenting a clear notion of the desirable internal structure for aspects. Our approach was to pinpoint the refactorings that would be needed to transform the Java implementations into the AspectJ implementations. We next tested and refined the refactorings thus obtained using Java implementations of the same patterns by independent authors [25] and [20], which further enriched the patterns' potential as providers of insights.

6. CONTRIBUTIONS

The main results of this Ph.D. project are the following:

- A collection of 28 refactorings for the AspectJ programming language.
- A review of the traditional OO code smells in light of AOP.
- The proposal of several novel code smells, including one that is specific to aspects.
- Several considerations associated with the above refactorings and code smells, most of which are presented in [54].

The collection of 28 refactorings is structured in the following groups:

- 10 refactorings for the extraction of crosscutting concerns from Java code bases to aspects.
- 6 refactorings for improving the internals of aspects, including aspects resulting from extraction processes performed according to refactorings of the previous group.
- 11 refactorings to deal with the extraction of common code between multiple aspects and the associated transfer of aspect-specific constructs between superaspects and subaspects.
- One refactoring dealing with the separation of concerns in the signature of constructors that are part of published interfaces.

6.1. Publications

These contributions are presented/documented in the following publications:

- [53] states the aims of this Ph.D. project and presents very early results.
- [50] presents our first case study and documents 5 refactorings that stemmed from that study.
- [51] presents a short analysis of the code examples that comprised our second case study in the light of ease of use and reusability.
- [49] is a technical report documenting all 28 refactorings, in a style and format similar to the ones used in [29] and [38]. We chose to document this part of our work through a technical report when we concluded that it would be extremely tricky to document refactorings in conference proceedings, due to space constraints and to the fact that small sets of refactorings are not likely to be considered a substantial enough contribution to be accepted by the review boards of prestigious international conferences.
- [54] presents most of the refactorings, reviews the traditional OO code smells in light of AOP, proposes several novel aspect-oriented code smells, including one that is specific to aspects. It presents considerations of various orders, including the effects of crosscutting in the design of existing OO systems. It also surveys related work and proposes several new directions of research in this field.

- [52] presents a refactoring process of a Java code base into an AspectJ equivalent, using 17 of the refactorings documented in [49]. It is complemented with an eclipse project, available online, presenting dozens of complete snapshots of the code being refactored.

7. RELATED WORK

Deursen et al [23] give a brief overview of the state of art in the area of aspect mining and refactoring. Though their main concern seems to be tools for the automatic detection of aspects, they also mention several open questions about refactoring to aspects, including "how can existing code smells be used to identify candidate refactorings?" and "how can the introduction of aspects be described in terms of a catalogue of new refactorings?". In our work we contribute to answering these two questions.

Iwamoto and Zhao announce in [36] their intention to build a catalogue of AO refactorings. They present a catalogue of 24 refactorings, but the information provided about them is limited to their names, with no further information. The refactorings we document in [49] include a characterisation of the situations where the refactoring applies, mention of preconditions, descriptions of the mechanics, and code examples.

Several authors [65][36][32][69][72] call into attention what we call the *fragile base code problem*, caused by the fact that almost all refactorings can potentially break existing aspects, particularly pointcuts. We do not directly tackle this problem in our work but we hope that adoption of an appropriate style for programming and evolving aspect constructs – particularly pointcuts – can ameliorate it until a new generation of tools that take into account the presence of aspects is available.

Hanenberg et al [32] propose a set of enabling conditions to preserve the observable behaviour. By the author's admission, these conditions must be automatically verified by an aspect-aware tool, as the manual verification is an exhausting task, even in small systems. Hanenberg et al announce a tool – implemented as a plug-in for eclipse [9] – providing a subset of the functionality they deem desirable for a few refactorings. These authors also document a small set of refactorings for AspectJ, including a basic refactoring for extracting code snippets from existing objects to aspects. These refactorings only scratch the surface of the entire refactoring space, and our collection of refactorings [49] goes significantly deeper, providing more detail and tackling other issues, namely the tidying up of the internal structure of aspects resulting from extraction processes.

In [43] Laddad presents a collection of novel refactorings and prescribes several guidelines to ensure AO refactorings for concern extraction are applied in a safe way. This material has a significant utility value, particularly to developers of J2EE applications. However, these are not presented in a style and format similar to the ones used in [29][38] and [49], and we believe that as a consequence some of the potential insights are lost.

Tonella and Ceccato [68] base their work on the assumption that interfaces are often (though not always) related to concerns other than the one pertaining to the system's main decomposition. The authors provide very specific guidelines for when an interface implementation is a symptom of a latent aspect and present an aspect mining and concern extraction tool which uses these criteria, and report on experimental results. These extractions are also covered by the refactorings we document in [49]. The authors also point out issues that can arise in a typical extraction of an interface implementation into an aspect. The refactorings from [49] prescribe procedures to deal with all these issues.

To our knowledge, no work besides ours deals with the potentially bad internal structure of aspects resulting from extraction processes. We also do not have knowledge of any other work covering the issue of AO code smells, the work by Tonella and Ceccato [68] being the only exception, though these authors do not give this name to their work.

8. THESIS CLAIM

AOP requires its own specific programming style, which catalogues of AOP-specific code smells and refactorings can help to capture. It is beneficial to present those catalogues in a familiar style, analogous to the one used in [29] and [38].

9. REFERENCES

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

[2] AspectJ homepage: http://www.eclipse.org/aspectj/.

[3] Aspect-Oriented Software Development homepage: http://aosd.net/.

[4] AspectJ for emacs: http://aspectj4emacs.sourceforge.net/

[5] AspectJ for JBuilder: http://aspectj4jbuildr.sourceforge.net/

[6] AspectJ for NetBreans: http://aspectj4netbean.sourceforge.net/

[7] Composition Filters homepage: http://trese.cs.utwente.nl/composition_filters/

[8] Demeter project homepage: http://www.ccs.neu.edu/research/demeter/.

[9] Eclipse Home Page. http://www.eclipse.org

[10] Product-Line Architecture Research Group homepage: http://www.cs.utexas.edu/users/schwartz/.

[11] Multi-Dimensional Separation of Concerns project homepage: http://www.research.ibm.com/hyperspace/.

[12] Refactoring homepage: http://www.refactoring.com/.

[13] Aksit, M., Bergmans, L., Vural, S., "An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach", ACM, proceedings of the ECOOP'92, June/July, pp. 372-396.

[14] Aksit, M., Bosch, J., Van Der Sterren, W., Bergmans, L., "Real-Time Specification Inheritance Anomalies and Real-Time Filters", pp. 386–405 of ECOOP '94, July 1994.

[15] Baquero C., Oliveira R., Moura F., "Integration of Concurrency Control in a Language with Subtyping and Subclassing", pp. 173-183 of the proceedings of the 1st USENIX Conference on Object-OrientedTechnology and Systems (COOTS), 1995.

[16] Beck, K., "Extreme Programming Explained – Embrace Change", Addison Wesley 2000. ISBN 0-201-61641-6.

[17] Bergmans, L., Aksit, M., "Composing Crosscutting Concerns using Composition Filters", pp. 51-57 of Communications of the ACM, October 2001.

[18] Buschmann, F., Meunier, R., Rohnert, H. , Sommerlad, P., Stal, M., "Pattern-Oriented Software Architecture: A System of Patterns", John Wiley & Sons, 1996. ISBN 0471958697.

[19] Clement, A., Colyer, A. Kersten, M., "Aspect-Oriented Programming with AJDT", workshop on Analysis of Aspect-Oriented Software (ECOOP'03), 2003.

[20] Cooper, J., "Java Design Patterns: A Tutorial", Addison-Wesley 2000. ISBN: 0201485397. Also availabe at

http://www.patterndepot. com/put/8/DesignJavaPDF.ZIP or http://www. patterndepot.com/put/8/DesignJava.PDF.

[21] Czarnecki, K., "Generative Programming: Principles and Techniques of Software Engineering Based on Automated Cofiguration and Fragment-Based Component Models", Ph.D. thesis, Technische Universität Ilmenau, Germany, 1998.

[22] Czarnecki, K., Eisenecker, U. W., "Generative Programming - Methods, Tools, and Applications", Addison Wesley, June 2000. ISBN 0201309777.

[23] Deursen, A., Marin, M., Moonen, L., "Aspect Mining and Refactoring", workshop on REFactoring: Achievements, Challenges, Effects (REFACE03), Waterloo, Canada, November 2003.

[24] Dijsktra, E., "A Discipline of Programming", Prentice Hall, 1976.

[25] Eckel, B., "Thinking in Patterns", book in progress, revision 0.9, 20 May 2003. Available at http://64.78.49.204/TIPatterns-0.9.zip

[26] Elrad, T. (moderator) with panelists Aksit, M., Kiczales, G., Lieberherr, K., Ossher, H. "Discussing Aspects of AOP", pp. 33-38 of Communications of the ACM, October 2001.

[27] Fernandes, S. M., Cachopo J., Silva, A. R., "Supporting Evolution in Workflow Definition Languages", proceedings of SOFSEM 2004, Czech Republic, Merin, January 2004.

[28] Filman, R. E., Friedman, D. P., "Aspect-Oriented Programming is Quantification and Obliviousness", workshop on Advanced Separation of Concerns (OOPSLA 2000), October 2000, Minneapolis.

[29] Fowler, M. (with contributions by K. Beck, W. Opdyke and D. Roberts), "Refactoring – Improving the Design of Existing Code", Addison Wesley 2000. ISBN 0201485672.

[30] Gamma, E., Helm, R., Johnson, R., Vlissides, J., "Design Patterns, Elements of Reusable Object-Oriented Software", Addison Wesley, 1995. ISBN 0201633612.

[31] Griswold, W., Kato, Y., Yuan, J., "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.

[32] Hanenberg, S. , Oberschulte, C., Unland, R., "Refactoring of Aspect-Oriented Software", Net.ObjectDays 2003, Erfurt, Germany, September 2003.

[33] Hannemann, J., Kiczales, G., "Design Pattern Implementation in Java and AspectJ" proceedings of OOPSLA'02, USA, Seatle, 2002.

[34] Harrison, W., Ossher, H., "Subject-Oriented Programming (A Critique of Pure Objects)," proceedings of the OOPSLA'93, 1993.

[35] Holmes, D., "Synchronisation Rings – Composable Synchronisation for Object-Oriented Systems", Ph.D. thesis, Macquarie University, Sydney. 20th October 1999.

[36] Iwamoto, M., Zhao, J., "Refactoring Aspect-Oriented Programs", 4th AOSD Modeling With UML Workshop, UML'2003, San Francisco, USA, October 2003.

[37] Johnson, R., "Frameworks = (Components + Patterns)", Communications of the ACM, 40(10), 39-42, 1997.

[38] Kerievsky, J., "Refactoring to Patterns", Addison-Wesley, 2004. ISBN 0321213351.

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

[40] Khoshafian, S., Abnous, R., "Object Orientation, second edition", John Wiley & Sons, 1995. ISBN 0471078344.

[41] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W. G., "An Overview of AspectJ", ACM, proceedings of the ECOOP 2001, Budapest, Hungary, 2001.

[42] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J., "Aspect-Oriented Programming", ACM, proceedings of the ECOOP 1997, Finland. Springer-Verlag Lecture Notes in Computer Science, vol. 1241, June 1997.

[43] Laddad, R., "Aspect-Oriented Refactoring", parts 1 and 2, The Server Side, 2003. http://www.theserverside.com/

[44] Lieberherr, K., Orleans, D., Ovlinger, J. "Aspect-Oriented Programming with Adaptive Methods", pp. 39-41 of Communications of the ACM, October 2001.

[45] Lopes, C. V., "D: A Language Framework for Distributed Computing", Ph.D. thesis, College of Computer Science, Northeastern University, Boston, 1997.

[46] Matsuoka, S., Yonezawa, A., "Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages" in "Research Directions in Concurrent Object-Oriented Programming" (Agha G., Wegner P., et al., Eds.), pp. 107-150, MIT press, 1993.

[47] McHale,C., "Synchronisation in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance", Ph.D. thesis, Department of Computer Science, Trinity College, Dublin, 1994.

[48] Meyer, B., "Object-Oriented Software Construction, second edition", Prentice Hall, 1997. ISBN 0136291554.

[49] Monteiro, M. P. , Fernandes, J. M., "Catalogue of Refactorings for AspectJ", Technical Report UM-DI-GECSD-200401, Departamento de Informática, Universidade do Minho, August 2004. Available at http://www.di.uminho.pt/~jmf/PUBLI/papers/2004-TR-01.pdf

[50] Monteiro, M. P. , Fernandes, J. M., "Object-to-Aspect Refactorings for Feature Extraction", Industry paper presented at the AOSD'2004, Lancaster, March 2004. Available at http://aosd.net/2004/archive/Monteiro.pdf

[51] Monteiro, M. P. , Fernandes, J. M., "Pitfalls of AspectJ Implementations of Some of the Gang-of-Four Design Patterns", proceedings of the DSOA'2004 workshop at JISBD 2004 (IX Jornadas de Ingeniería de Software y Bases de Datos), Málaga, Spain, November 2004.

[52] Monteiro, M. P. , Fernandes, J. M., "Refactoring a Java Code Base to AspectJ – An Ilustrative Example", practitioner's report submitted to the AOSD'2005.

[53] Monteiro, M. P. , Fernandes, J. M., "Some Thoughts On Refactoring Objects to Aspects", proceedings of the DSOA'2003 workshop at JISBD 2003 (VIII Jornadas de Ingeniería de Software y Bases de Datos), Alicante, Spain, November 2003.

[54] Monteiro, M. P. , Fernandes, J. M., "Towards a Catalog of Aspect-Oriented Refactorings", accepted for publication in the proceedings of the AOSD'2005, to take place in Chicago, USA, 14-18 March 2005.

[55] Nierstrasz, O., Tsichritzis, D. (eds), "Object-Oriented Software Composition", Prentice Hall, 1995. ISBN 0132206749.

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

[57] Ossher, H., Tarr, P., "Using Multdimensional Separation of Concerns to (Re)Shape Evolving Software", pp. 43-50 of Communications of the ACM, October 2001.

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

[59] Pree, W., “Design Patterns for Object-Oriented Software Development”, Addison-Wesley, 1995. ISBN: 0201422948

[60] Sabbah, D., “Aspects – from Promise to Reality”, proceedings of the AOSD 2004, Lancaster, UK, March 2004.

[61] Savarese, D. F., "Java's Continuing Evolution", November 2002 of Java Pro magazine.

[62] Silva, A. R., “Programação Concorrente com Objectos: Separação e Composição de Facetas com Padrões de Desenho, Linguagem de Padrões e Moldura de Objectos”, Ph.D. thesis, Instituto Superior Técnico, Universidade Técnica de Lisboa, March 1999.

[63] Snyder, A., “The Essence of Objects: Concepts and Terms”, pp. 31-42 of IEEE Software, January 1993.

[64] Spurlin, V., "Aspect-Oriented Programming with Sun ONE Studio: A Demonstration", November 2002 http://forte.sun.com/ffj/articles/aspectJ.html.

[65] Störzer, M., Koppen, C., “PCDiff: Attacking the Fragile Pointcut Problem”, Interactive Workshop on Aspects in Software (EIWAS) 2004, Berlin, Germany, September 2004.

[66] Szyperski, C., "Component Software, Beyond Object-Oriented Programming", Addison-Wesley, 1999.

[67] Tarr, P., Ossher, H., Harrison, W., Sutton Jr., S.M., “N Degrees of Separation: Multi-Dimensional Separation of Concerns”, proceedings of the ICSE'99, May, 1999.

[68] Tonella, P., Ceccato, M., "Migrating Interface Implementation to Aspects", pp. 220-229 of the proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04), Chicago, USA, September 2004.

[69] Tourwé, T., Brichau, J., Gybels, K., “On the Existence of the AOSD-Evolution Paradox”, AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies, Boston, USA, 2003.

[70] Wegner, P., “Dimensions of Object-Based Language Design”, pp. 168–182 of the proceedings of OOPSLA ’87, October 1987.

[71] Whitehead, K., “Component-based Development – Principles and Planning for Business Systems”, Addison Wesley Component Software Series (C. Szyperski, Series Editor) 2002. ISBN 0-20167-528-5.

[72] Wloka, J., “Refactoring in the Presence of Aspects”, ECOOP2003 PhD workshop, July 2003.