Using Reengineering and Aspect-based Techniques to Retrieve Knowledge Embedded in Object-Oriented Legacy System

Vinicius Cardoso Garcia, Daniel Lucrédio, Antonio Francisco do Prado
GOES – Software Engineering Group
Federal University of São Carlos – São Carlos, SP, Brazil
{vinicius, lucredio, prado}@dc.ufscar.br

Eduardo Santana de Almeida, Alexandre Alvaro
C.E.S.A.R. – Recife Center for Advanced Studies and Systems
Federal University of Pernambuco – Recife, PE, Brazil
{esa2, aa2}@cin.ufpe.br

Abstract

This paper presents an approach to retrieve the knowledge embedded in object-oriented legacy system. This approach aids in the migration from object-oriented code, written in Java, to a combination of objects and aspects, using AspectJ. The approach uses aspect mining in order to identify possible crosscutting concerns from the object-oriented source code and extracts them through refactorings into new aspect-oriented code. Next, the aspect-oriented design is retrieved through software transformations and may be imported in a CASE tool, becoming available in higher abstraction levels. The retrieved information constitutes important knowledge that may be reused in future projects or in reengineering.

1 Introduction

Reengineering is a way to achieve software reuse and to understand the concepts underlying the application domain. Its objective is to acquire and maintain the knowledge embedded in legacy systems, using it as a base for the continuous and structured evolution of the software system. The legacy code has programming logics, project decisions, user requirements and business rules that can be retrieved and rebuilt without losing semantics.

Current reengineering process models aim at reconstructing procedural systems using object-orientation or component-based techniques. The product of these process is usually the reconstructed, updated system and its new documentation, resulting in improved maintainability and flexibility. Also, new developments may benefit from these products, reusing the retrieved assets, which include models, documentation and source code, saving effort and time.

However, the quality of these retrieved reusable assets is too dependent on the experience and level of expertise of the involved people. This happens mainly because Object-Orientation itself does not assure good modularization, where each functionality is grouped in a single module [21].

Aspect-Oriented Software Development (AOSD) [13] may help to reduce this dependency, by offering a new modular unit (Aspect). Functionalities that are necessarily dispersed in object-oriented systems, such as exception handling and logging, for example, can be grouped into a single Aspect, increasing the modularity and the reuse level of the retrieved assets.

This paper presents an approach to retrieve the knowledge embedded in object-oriented legacy systems using reengineering and AOSD techniques, such as aspect mining, refactoring and software transformation. The reengineering product has great reuse potential, due to the benefits of AOSD.

2 Background

The proposed approach combines different techniques based on our experience in software reengineering [2, 7]. This section presents the main concepts underlying the approach, discussing how they are used in order to provide an effective way to extract reusable knowledge from OO legacy systems.

2.1 Reengineering

Software reengineering, also known as renewal or recovery, has as main objective to improve software quality, maintaining the basic system functionality [18]. In [12], Jacobson and Lindstron define reengineering with the following formula: *Reengineering = Reverse engineering* + Δ + *Forward Engineering*.

Reverse Engineering is the activity of defining a more abstract, and easier to understand, representation of the system. "Δ" represents change of the system (functionality or implementation). Forward Engineering is the activity of creating an executable representation of the system.

Our approach covers only the Reverse Engineering activity, since the idea is only to extract reusable knowledge. However, once this knowledge is obtained, Forward Engineering may be easily carried out as a regular software development process.

In order to assure that the produced assets are reusable, the Reverse Engineering activity uses AOSD techniques to increase the modularity and promote the separation of concerns.

2.2 Aspect-Oriented Software Development

Some existent functionalities in the software systems, such as exception handling and logging, are inherently difficult to decompose and isolate, reducing the legibility and the maintainability of these systems. Object-orientation techniques and Design Patterns [6] may reduce these problems but they are not enough [13].

The AOSD appeared in the 90s as a paradigm addressed to separate *crosscutting concerns* (Aspects) through code generation techniques that combine (weave) Aspects into the application logic [13].

Separation of concerns is a well-established principle in software engineering. A concern is some part of the problem that must be treated as a single conceptual unit [21]. Concerns are modularized throughout software development using different abstractions provided by languages, methods and tools.

In our approach, AOSD is used to increase the reusability of the extracted knowledge, by encapsulating functional and non-functional concerns into separate units. However, before this encapsulation, there must be a way to identify, in the legacy system, where these different concerns are located. To perform this, Aspect Mining techniques are used.

2.3 Aspect Mining

To identify different concerns inside legacy systems is not an easy task. It is common to find *"spaghetti code"* and *ad-hoc* design due to the improper use of programming and modeling techniques.

The Aspect identification requires, initially, a clear idea of where to look for them. Thus, the study of the crosscutting concern that is being mined must precede the mining itself. For example, before mining the *database persistence* concern, one must understand how this concern is usually treated, and how it is treated in this particular system. This study may involve the analysis of the documentation and the search for specific method calls, constructor calls, field access, and other well-defined execution points, aiming to discover possible *join points* involving the concern that is being mined.

The mining itself consists in scanning the whole system, marking every documentation, model and line of code that is related to the aspect, with basis on the previous study. Parsing may be useful in this activity, semi-automatic highlighting possible aspects occurrences inside the source code.

Several works focus on aspect mining [11, 16, 19, 20, 4] to help in the identification of crosscutting concerns inside software systems. There are two main approaches:

i. The type-based mining considers object types, variable types and method return types in order to determine if the code is related to the aspect that is being mined. However, it is not possible to differentiate objects that have the same type but different goals.

ii. The text-based mining considers character sequences and regular expressions in order to find the aspect. However, it ignores the type of the objects, and may not identify concerns that are not in the expression used in the search.

Ideally, these two approaches should be used together, so that one complements the other.

Once identified, the aspects must be extracted and encapsulated in separate units. In our approach, this is achieved through refactoring.

2.4 Refactoring

Refactoring [5] is a technique to restructure source code in a disciplined way. The intention of refactoring is to improve the readability and comprehensibility of source code. Most refactorings increase the modularity of code and eliminate redundancies. Also, they reduce the chance of introducing errors during program restructuring, since there is a predefined and well-tested set of steps that, if followed, produces the expected result.

By looking at existent refactoring catalogues, such as [5], we identified several similarities between code improvement refactorings and aspect extraction. Thus, we defined some refactorings [9], with basis on existing refactorings [5, 10, 17], that are used to extract the aspects from the legacy source code.

After extracting the aspects, a high-level representation of the retrieved assets must be obtained. Our approach uses software transformations to help and semi-automate this task.

2.5 Software transformation

Software transformation consists in automatically mapping among different language representations. Using transformations, it is possible, for example, to transform a program written in C++ into a program written in Java. Our experience has proven that it is possible to transform among different programming paradigms and even between different abstraction levels [7, 2], obtaining high-level models from source code.

In this way, software transformation is used to obtain high-level extended UML models that represent the aspects in the refactored code. The models may then be loaded into a modeling tool to be viewed and edited.

3 An Approach to Retrieve Knowlegde Embedded in Object-Oriented Legacy System

The approach is based on Aspect-Oriented reverse engineering techniques and is supported by two mechanisms: a transformational system (called Draco-PUC [15]) and a modeling tool (called MVCASE [1]).

Figure 1 shows the approach.

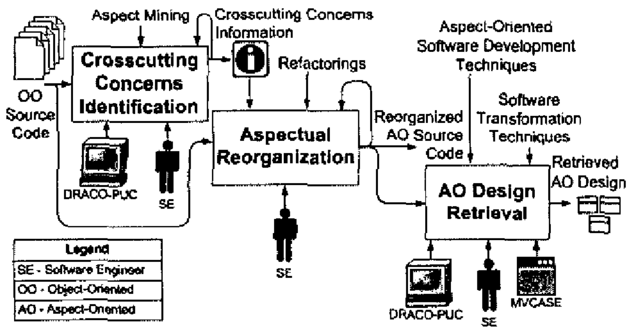
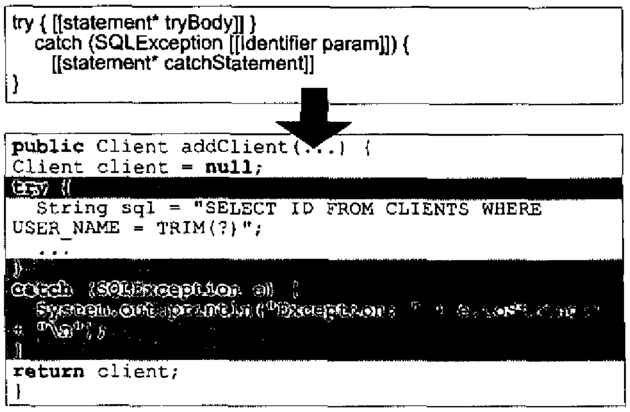


Figure 1. Reverse Engineering

In order to identify and extract crosscutting concerns in legacy systems, tree steps are performed, as follows.

Crosscutting Concerns Identification. Initially, the software engineer analyzes the legacy system aiming to identify possible crosscutting concerns that are present. These will serve as input to the aspect mining, which determines where these concerns are located inside the system. In our approach, aspect mining is performed using character sequence and regular expression analysis, and parser-based mining, implemented in the transformational system Draco-PUC[1]. The idea is to find static join points, occurring in the context of the program, that refer to specific crosscutting concerns.

Figure 2 shows an example of how parser-based mining may help in the identification of the exception handling crosscutting concern.



```
try { [[statement* tryBody]] }
   catch (SQLException [[identifier param]]) {
      [[statement* catchStatement]]
}
```

```
public Client addClient(...) {
Client client = null;
try {
  String sql = "SELECT ID FROM CLIENTS WHERE
USER_NAME = TRIM(?)";
  ...
}
catch (SQLException e) {
  System.out.println("Exception: " + e.toString() +
+ "\n");
}
return client;
}
```

Figure 2. Exception Handling Crosscutting Concern Identification

The parser recognizes the *"try-catch"* syntactic structure in Java code and the occurrence of a *SQLException* exception type, that indicates the presence of a non-functional requirement (exception handling). This information is stored to be later consulted, in order to aid the software engineer to extract and encapsulate that crosscutting concern into aspects. However, it must be stressed that parser-based mining, as well as character sequence and regular expression analysis, is just an aid to perform the mining, which must be carried out manually by the software engineer.

Aspectual Reorganization. After the crosscutting concerns are identified, the software engineer uses refactorings to extract and encapsulate these concerns into aspects. These refactorings [9] consider the interlaced nature of the legacy system code, and thus the transfer of individual members from classes to an aspect should not be isolated. In most cases, they are part of a set of transfers that comprise all the implementation elements of the concern that is being extracted. Such concerns typically include multiple code fragments scattered across multiple modular units (e.g. methods, classes, packages). Therefore, in many cases, more than one refactoring should be applied to extract a particular concern.

Aspect-Oriented Design Retrieval. Next, the software engineer, using software transformations in the Draco-PUC Transformational System, obtains the AO design, in UML

[1]Although Draco-PUC is mainly a transformational system, it can operate as a parser generator as well.

descriptions. The transformations map descriptions in a programming language, corresponding to the reorganized AO code, into descriptions in a modeling language, which can be loaded into MVCASE tool. Then, the software engineering may edit the retrieved models in MVCASE, inserting minor corrections and refinements [8]. We currently use an UML extension that is capable of representing AOSD concepts, and is implemented in MVCASE. More information on automatic design retrieval using transformations may be seen in [7].

Figure 3 shows an example of AO design retrieval using software transformations. The AO Source code (1) is analyzed by the transformations (2), which are responsible for mapping the code into descriptions in a modeling language (3). These descriptions are then loaded into MVCASE, becoming available for edition (4).

The retrieved design and the Aspect-Oriented source code constitute the knowledge of the legacy system. Next, this knowledge is encapsulated in highly modularized, reusable assets, with updated and consistent documentation. These assets may then be used in future developments or in the reconstruction of the legacy system, which is now more maintainable and extensible.

4 Partial Evaluation

In order to obtain a partial evaluation of the proposed approach, a pilot project was performed.

The pilot project has involved the reverse engineering of a Bank Teller System, which was developed in Java. It is composed of 11 classes, where 3 contain business rules and 8 are related to Graphical User Interfaces, through the *java.swing* package. The system had 1354 lines of code.

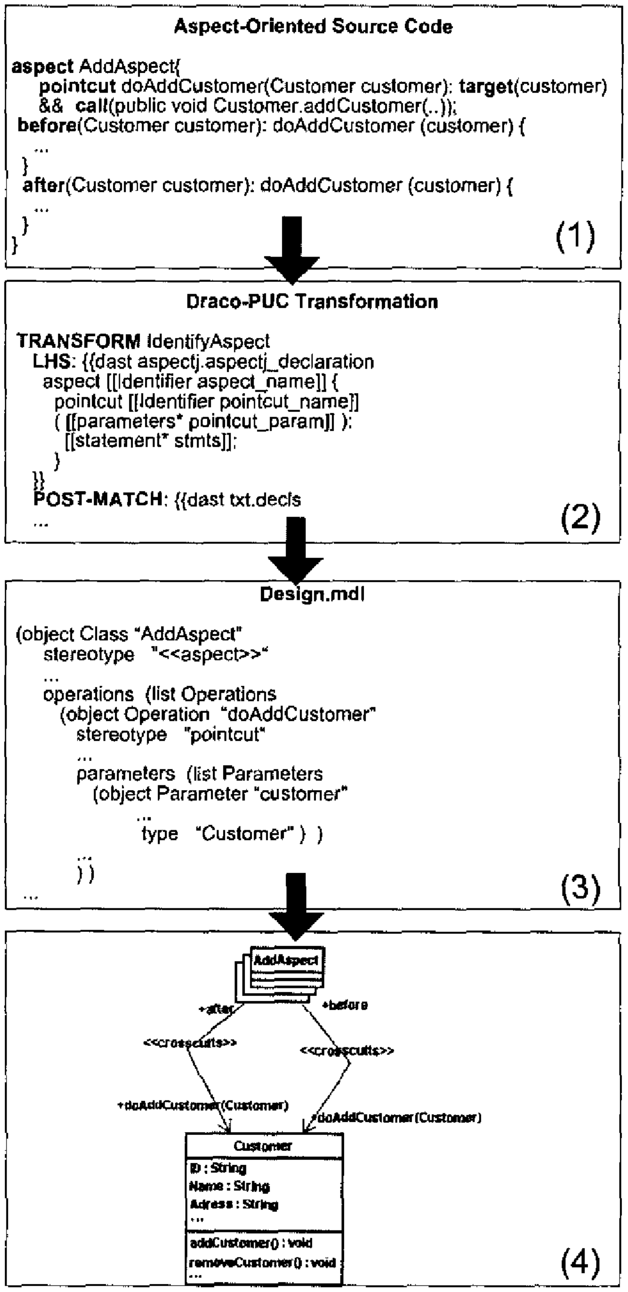
No documentation was available other than source code comments. The system understanding was performed through its execution, which generated a document containing its main functionalities, such as:

i. The system allows the customer to register itself, as well as its accounts. The account information includes the initial balance, and the account type; and

ii. For the user to access his accounts, it is necessary to inform an user name and a previously stored password. After this identification, the customer informs the account number and the value that he wants to draw or deposit.

After understanding, the system was analyzed in order to identify the possible crosscutting concerns, that were interlaced and spread through the classes. In this pilot project, two concerns were identified: database persistence and exception handling.

Refactorings were applied to extract these concerns, and the AO Design was retrieved through transformations. In order to verify if the retrieved assets were still in conformance with the original system requirements, a new implementation of the system was performed in a forward engineering step. The new AO system was executed, and its observation verified that the functionalities of the OO system were maintained.


Aspect-Oriented Source Code

```
aspect AddAspect{
   pointcut doAddCustomer(Customer customer): target(customer)
   && call(public void Customer.addCustomer(..));
 before(Customer customer): doAddCustomer (customer) {
   ...
 }
 after(Customer customer): doAddCustomer (customer) {
   ...
 }
}
```
(1)

Draco-PUC Transformation

```
TRANSFORM IdentifyAspect
  LHS: {{dast aspectj.aspectj_declaration
   aspect [[Identifier aspect_name]] {
     pointcut [[Identifier pointcut_name]]
     ( [[parameters* pointcut_param]] ):
       [[statement* stmts]];
     }
  }}
  POST-MATCH: {{dast txt.decls
  ...
```
(2)

Design.mdl

```
(object Class "AddAspect"
   stereotype   "<<aspect>>"
   ...
   operations  (list Operations
     (object Operation  "doAddCustomer"
       stereotype   "pointcut"
       ...
       parameters  (list Parameters
         (object Parameter "customer"
           ...
           type   "Customer" ) )
       ...
     ))
...
```
(3)

Figure 3. AO Design Retrieval

Table 1 shows a brief comparison between the OO and the AO systems.

The reduced number of lines of code and the increased number of modules (aspects) indicate that the retrieved assets are smaller and better divided. Since each aspect

	OO	AO
Classes	11	11
Aspects	-	6
Lines of Code (LOC)	1354	1286

Table 1. Result Evaluation

groups a single concern, the modules are also more cohesive. Therefore, we may deduct that the retrieved assets are more reusable than the original ones.

The consequences on identifying and separating crosscutting concerns are equivalent to those stated by AOSD:

i. Requirements traceability: After the separation of concerns, it is easier to trace each module to a specific requirement;

ii. Easier Maintenance: Requirement changes, functionality improvements and code restructuring are also easier to perform; and

iii. Readability: The new code is lighter and less polluted, because attributes and methods are not spread through the system.

iv. Reuse: The identified and extracted aspects can be implemented in such a way that they can be later reused. This may even give origin, with the accomplishment of different case studies, to a framework of aspects.

v. Maintainability: With the adoption of AOSD, the new system will be more maintainable, since the non-functional requirements are encapsulated in specific aspects and not dispersed throughout all the system.

5 Related Works

The first relevant work involving the OO technology and retrieval of knowledge embedded in legacy system was presented by Jacobson and Lindstron [12], who applied reengineering in legacy systems that were implemented in procedural languages. The authors state that reengineering should be accomplished in a gradual way, because it would be impracticable to substitute an old system for a completely new one.

Today, on the top of OO techniques, an additional layer of software development, based on components, is being established. The goals of *"componentware"* are very similar to those of OO: reuse of software is to be facilitated and thereby increased, software shall become more reliable and less expensive [14].

Among the first research works in this direction, Caldiera and Basili [3] have explored the automated extraction of reusable software components from existing systems. They propose a process that is divided in two phases. First, it chooses, from the existing system, some candidates and packages them for possible independent use. Next, an engineer with knowledge of the application domain analyzes each component to determine the services it can provide.

Another work involving software components and reengineering to retrieve knowledge embedded in legacy systems may be seen in [2], where we have presented a *CASE* environment for component-based software reengineering, called *Orion-RE*. The environment uses software reengineering and Component-Based techniques to rebuild legacy systems, reusing the available documentation and the built-in knowledge in their source code. We observed some benefits in the reconstructed systems, such as grater reuse degree and easier maintenance. We also observed benefits due to the automation achieved through *CASE*.

6 Conclusions and Future Work

Many reverse engineering and reengineering approaches have been proposed to retrieve knowledge from legacy systems. The goal is to develop a global picture on the subject system, which is the first major step toward its understanding or transformation into a system that better reflects the quality needs of the application domain.

A new tendency is that the research moves toward the separation of concerns area, behind AOSD, integrated with already existent reverse engineering and reengineering techniques. This will represent one step forward in the direction of the post-OO technologies, seeking even higher maintainability and flexibility.

This paper has presented an approach proposal, using reengineering techniques to retrieve knowledge embedded in OO legacy systems. This approach integrates different techniques and mechanisms to guide and help software engineers during the process.

The approach uses aspect-oriented software development concepts, in order to obtain a better reuse degree in the retrieved assets. It also helps to improve development productivity and support for changes in the requirements.

Additionally, an evaluation was accomplished to show the reengineering proccess usefulness. By following the approach, we could verify that the AOSD brings several and important benefits to software development. The way the aspects are combined with the system modules allows the inclusion of additional responsibilities without committing the code clarity, maintainability, reusability, and providing a greater reliability.

As a future work, the UML extension used in the AO modeling (used by the MVCASE tool) will be validated. Also, the transformations defined in the Draco-PUC transformational system must be proven correct, through the accomplishment of more case studies, involving different domain systems.

Graphical visualization of the possible crosscutting concerns source code is also being developed. In this way, the task of identifying different concerns in the legacy system should be facilitated.

References

[1] E. Almeida, C. Bianchini, A. Prado, and L. Trevelin. MVCASE: An integrating technologies tool for distributed component-based software development. In *Proceedings of the 6th Asia-Pacific Network Operations and Management Symposium. (APNOMS'2002) Poster Session.* IEEE Computer Society Press, 2002.

[2] A. Alvaro, D. Lucrédio, V. C. Garcia, E. S. de Almeida, A. F. do Prado, and L. C. Trevelin. Orion-RE: A Component-Based Software Reengineering Environment. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE)*, pages 248–257. IEEE Computer Society Press, November 2003.

[3] G. Caldiera and V. R. Basili. Identifying and qualifying reusable software components. *IEEE Computer*, 24(2):61–71, Feb. 1991.

[4] A. v. Deursen, M. Marin, and L. Moonen. Aspect Mining and Refactoring. In *Proceedings of the First International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE03). Held in conjunction with WCRE 2003.* University of Waterloo, Canada, November 2003.

[5] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: improving the design of existing code.* Object Technology Series. Addison-Wesley, 1999.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison Wesley Professional Computing Series. Addison-Wesley, 1995.

[7] V. C. Garcia, V. Fontanette, A. B. Perez, and A. F. Prado. RST - Software Reengineering using Transformations (in portuguese), RHAE/CNPQ. Technical Report 610.069/01-2, Department of Computer, Federal University of São Carlos, São Carlos-SP, Brazil, March 2004.

[8] V. C. Garcia, D. Lucrédio, L. Frota, A. Alvaro, E. S. de Almeida, and A. F. do Prado. A case tool for aspect-oriented software development (*in portuguese*). In *XI Tools Section - XVIII Brazilian Symposium on Software Engineering (SBES 2004). (to appear)*, October 2004.

[9] V. C. Garcia, E. K. Piveta, D. Lucrédio, A. Alvaro, E. S. de Almeida, A. F. do Prado, and L. C. Zancanella. Manipulating Crosscutting Concerns. *4th Latin American Conference on Patterns Languages of Programming (SugarLoaf-Plop 2004)*, 2004.

[10] S. Hanenberg, C. Oberschulte, and R. Unland. Refactoring of aspect-oriented software. In *Net.Object Days 2003*, October 2003.

[11] J. Hannemann and G. Kiczales. Overcoming the prevalent decomposition in legacy code. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE'2001). Workshop on Advanced Separation of Concerns in Software Engineering.*, May 2001.

[12] I. Jacobson and F. Lindstrom. Reengineering of old systems to an object-oriented architecture. In *Proceedings of the Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'91)*, pages 340–350. ACM Press, 1991.

[13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the 11st European Conference Object-Oriented Programming (ECOOP'97)*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[14] E. Lee, B. Lee, W. Shin, and C. Wu. A reengineering process for migrating from an object-oriented legacy system to a component-based system. In *Proceedings of the 27th Annual International Computer Software and Applications Conference (COMPSAC)*, pages 336–341. IEEE Computer Society Press, November 2003.

[15] J. C. Leite, M. Sant'anna, and F. G. Freitas. Draco-PUC: A Technology Assembly for Domain Oriented Software Development. In *Proceedings of the 3rd International Conference on Software Reuse (ICSR'94)*, pages 94–100. IEEE Computer Society Press, November 1994.

[16] N. Loughran and A. Rashid. Mining aspects. In *Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design (AOSD-2002)*, Mar. 2002.

[17] M. P. Monteiro and J. ao M. Fernandes. Object-to-aspect refactorings for feature extraction. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD'2004).* ACM Press, March 2004.

[18] R. S. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, fifth edition edition, 2001.

[19] M. P. Robillard and G. C. Murphy. Capturing concern descriptions during program navigation. In *Workshop on Tools for Aspect-Oriented Software Development (OOPSLA 2002)*, November 2002.

[20] J. K. Silvia Breu. Aspect mining using dynamic analysis. In *5. Workshop Software-Reengineering (Published in: GI-Softwaretechnik-Trends, Mitteilungen der Gesellschaft für Informatik*, volume 2, pages 21–22, May 2003.

[21] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of 21st International Conference on Software Engineering (ICSE'99)*, pages 107–119, Los Angeles CA, USA, 1999. IEEE Computer Society Press.