Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

SEN

Software Engineering

Software ENgineering

Proceedings of the 1st workshop on aspect reverse-engineering

T. Tourwé, M. Bruntink, M. Marin, D. Shepherd

REPORT SEN-E0502 FEBRUARY 2005

CWI is the National Research Institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organization for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-369X

Proceedings of the 1st workshop on aspect reverse-engineering

ABSTRACT
This technical report contains the papers submitted to and presented at the 1st Workshop on Aspect Reverse-Engineering, held in conjunction with the 11th Working Conference on Reverse Engineering (WCRE), in Delft, The Netherlands. The aims of this workshop was to bring together researchers and practitioners within the field of aspect reverse engineering, to review the state-of-the-art and state-of-the-practice and to identify a list of interesting open issues that remain to be studied. The workshop was organised as a structured discussion, based on interesting and relevant topics extracted from position papers.

1998 ACM Computing Classification System: D.2.7 Distribution, Maintenance, and Enhancement
Keywords and Phrases: aspect-oriented software development; reverse engineering; refactoring

@ WCRE 2004

Proceedings of the 1st Workshop on Aspect Reverse-Engineering

Held in conjuction with the 11th Working Conference on Reverse Engineering (WCRE), Delft, The Netherlands.
November 9th, 2004.

Workshop on Aspect Reverse Engineering

Tom Tourwé & Magiel Bruntink
Centrum voor Wiskunde en Informatica
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
{Tom.Tourwe, Magiel.Bruntink}@cwi.nl

Marius Marin
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands
a.m.marin@ewi.tudelft.nl

David Shepherd
University of Delaware
Newark, DE 19716
United States
shepherd@cis.udel.edu

Abstract

This technical report contains the papers submitted to and presented at the 1^{st} Workshop on Aspect Reverse-Engineering, held in conjunction with the 11^{th} Working Conference on Reverse Engineering (WCRE), in Delft, The Netherlands. The aims of this workshop was to bring together researchers and practitioners within the field of aspect reverse engineering, to review the state-of-the-art and state-of-the-practice and to identify a list of interesting open issues that remain to be studied. The workshop was organised as a structured discussion, based on interesting and relevant topics extracted from position papers.

1 Introduction

Aspect-oriented software development [1] aims at improving the handling of crosscutting concerns by capturing them explicitly in well-modularised entities, called aspects. In this way, it tries to improve the overall quality of an application, since improved modularisation should lead to better evolvability, maintainability, understandability, reusability, and so on [1, 2].

A large body of research exists on the development of aspect-oriented programming languages and mechanisms. As this research starts to mature, AOSD techniques are adopted in many new applications. Much less attention is paid, however, to how already existing applications can be improved by adopting these techniques. In particular, we should study how applications developed without AOSD techniques can be migrated into aspect-oriented applications. Additionally, even applications using AOSD from their inception might need to be re-engineered because concern code becomes less well-organised over time and because opportunities for aspects might not be apparent when different developers are working on the same code base independently.

The subject of aspect reverse engineering thus raises several interesting issues and questions:

- How can we identify aspects in the source code? Can we automate this process? Which techniques can we apply?
- How can we extract aspects from the source code? How do we define appropriate, understandable and high-quality aspects? What criteria should we use to determine the quality of an aspect?
- When should we prefer an aspect-oriented solution over an object-oriented solution?
- How can we restructure the ordinary source code so that the aspects are removed from it?

- What is the impact of the aspect language on the extraction process? Should specific aspect languages be developed, or do general aspect languages as they exist today suffice?
- How can we ensure the behaviour of the original applications is preserved after the migration? Can we use existing tests to ensure this, or do we need other kinds of tests?
- Will aspect-oriented techniques improve the overall quality of applications? How can we measure this quality improvement?
- Do these techniques scale up to applications spanning multiple millions of lines of code?

The goal of this workshop was to address these questions, identify possible other relevant and important issues in this domain and bring together researchers interested in and working on the subject.

2 Workshop Format

The half-day workshop was split into three sessions:

1. an opening session, that introduced the topics of discussion.
2. a session consisting of position paper presentations and discussion.
3. a summary and wrap-up session, where open questions, interesting future trends and possible collaborations were discussed.

Based on the submitted position papers, two interesting tracks were scheduled: a track on *aspect mining* and a track on *aspect refactoring*. The papers were presented as follows, with the presenters underlined:

Aspect Mining

1. Silvia Breu: *Towards Hybrid Aspect Mining: Static Extensions to Dynamic Aspect Mining*
2. Jens Krinke and Silvia Breu: *Control-Flow-Graph-Based Aspect Mining*.
3. David Shepherd, Jeffrey Palm and Lori Pollock: *The Fast Prototyping and Evaluation of Aspect Mining Analyses via Timna*.
4. Magiel Bruntink: *Aspect Mining using Clone Class Metrics*.

Aspect Refactoring

1. Marius Marin: *Refactoring JHOTDRAWs Undo concern to ASPECTJ*.
2. Magiel Bruntink, Arie van Deursen and Tom Tourwé: *Isolating Crosscutting Concerns in System Software*.
3. Andy Zaidman, Toon Calders, Serge Demeyer and Jan Paredaens: *Selective Introduction of Aspects for Program Comprehension*.
4. Mariano Ceccato and Paolo Tonella: *Measuring the Effects of Software Aspectization*.

The remainder of this technical report includes all of these papers for easy reference. The presentations that accompany the papers can be downloaded from the *WARE* website, to be found at the following URL: http://www.cwi.nl/ tourwe/ware.html.

The success of this workshop was mainly due to the people that attended it, presented their ideas and participated in the discussions. We would like to thank all of these people and hope you enjoy reading their contributions.

References

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the Europeen Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[2] R. J. Walker, E. L. Baniassad, and G. C. Murpy. An Initial Assessment of Aspect-Oriented Programming. In *Proceedings of the 21st International Conference on Software Engineering (ICSE)*, 1999.

Towards Hybrid Aspect Mining: Static Extensions to Dynamic Aspect Mining

Silvia Breu
MCT/NASA Ames Research Center
silvia.breu@gmail.com

Abstract

Aspect mining tries to identify crosscutting concerns in legacy systems and thus supports the refactoring into an aspect-oriented design. This position paper describes DynAMiT, *the first aspect mining tool that detects crosscutting concerns based on dynamic analysis. Furthermore, it presents the results of several case studies, and estimates the quality of the* DynAMiT *approach. Based on that, we propose a possible combination with static program information such as static object and inheritance information to extend and improve the dynamic approach.*

1. Motivation

With increasing needs, software systems grow in size and become more and more complex. The complexity does not only lie in the requirements on the programs but also in the problem of so-called *tangled code* [9]. This notion refers to code that exists several times in the system but cannot be encapsulated by separate modules using traditional techniques (e.g., object-oriented design principles). The problem occurs if underlying functionality crosscuts the whole software system. Thus, tangled code makes software systems more difficult to maintain, to understand, and to extend. *Aspect-Oriented Programming* (AOP) [9] provides new separation mechanisms for such complex *crosscutting concerns* [12]. AOP is a design technique that retains the advantages of object-oriented programming and aims at avoiding the *tyranny of the dominant decomposition*. Traditional languages and modularisation mechanisms suffer from that limitation: The program can be modularised in only one way at a time, and the many kinds of concerns that do not align with that modularisation end up scattered across many modules and tangled with one another. This new programming paradigm with its extensions to programming languages (e.g., AspectJ [14], AspectC++ [6]) has attracted attention as it enhances design and development of software systems. However, attention is increased, also drawn to the question how AOP can serve the community in re-engineering legacy systems.

A major problem in re-engineering legacy code based on aspect-oriented principles is to find and to isolate these crosscutting concerns. This task is called *aspect mining* [13]. Detected concerns can be re-implemented as separate aspects, thereby improving maintainability and extensibility as well as reducing complexity. Aspect mining can also provide insights that enable us to classify common aspects which occur in different software systems, such as logging, timing, and communication. Several approaches based on static program analysis techniques have been proposed for aspect mining [4, 7, 8, 10, 11, 15]. This position paper describes *DynAMiT* [1, 2, 3], the first dynamic program analysis approach that mines aspects based on program traces, presents an overview of some case studies, evaluates the approach's strengths and limitations, and proposes a possible direction of extensions and improvement.

2. DynAMiT

DynAMiT is a dynamic aspect mining approach based on program traces that are generated during program execution. These traces are then investigated for recurring execution relations. Different constraints specify when an execution relation is "recurring", such as the requirement that the relations have to exist more than once or even in different calling contexts in the program trace. The dynamic analysis approach has been chosen because it is a very powerful way to make inferences about a system: It dynamically monitors actual, i.e., run-time program behaviour instead of potential behaviour, as static program analysis does. The approach has been implemented in a prototye called *DynAMiT* (*Dyn*amic *A*spect *Mi*ning *T*ool) and evaluated in several case studies over systems with more than 80 kLoC. Case studies have shown that the technique is able to identify *automatically* both seeded and existing crosscutting concerns in software systems. The full results of both algorithms can be found in [1].

2.1. DynAMiT Approach

The data on which *DynAMiT* works are program traces. Within these traces we identify recurring execution patterns which describe certain behavioural aspects of the software system. We expect that recurring execution patterns are potential crosscutting concerns which describe recurring functionality in the program and thus are possible aspects.

In order to detect these recurring patterns in the program traces, a classification of possible pattern forms is required. Therefore, we introduce so-called *execution relations*. They describe in which relation two method executions are in the program trace. Intuitively, a program trace is a sequence of method invocations and exits. We only consider entries into and exits from method executions because we can then easily keep track of the relative order in which method executions are started and finished. We focus on method executions because we want to analyse object-oriented systems where logically related functionality is encapsulated in methods. Formally, a *program trace* T_P of a program P with method signatures $\mathcal{N}_P$ is defined as a list $[t_1, \ldots, t_n]$ of pairs $t_i \in (\mathcal{N}_P \times \{ent, ext\})$, where ent marks entering, and ext marks exiting a method execution.

Crosscutting concerns are now reflected by the two different *execution relations* that can be found in program traces: A method can be executed either after the preceeding method execution is terminated, or inside the execution of the preceeding method call. However, this distinction alone is not yet sufficient for aspect mining. For example, if there exists more than one method execution inside another method execution the information which of those methods inside comes first is lost. We thus define formally:

$\mathtt{u} \rightharpoonup \mathtt{v},\ \mathtt{u}, \mathtt{v} \in \mathcal{N}_P$, is called an *outside-before-execution relation* if $[(\mathtt{u}, ext), (\mathtt{v}, ent)]$ is a sublist of T_P. $S^{\rightharpoonup}(T_P)$ is the set of all outside-before-execution relations in a program trace T_P. This relation can also be reversed, i.e., $\mathtt{v} \leftharpoonup \mathtt{u}$ is an *outside-after-execution relation* if $\mathtt{u} \rightharpoonup \mathtt{v} \in S^{\rightharpoonup}(T_P)$. The set of all outside-after-execution relations in a program trace T_P is then denoted with $S^{\leftharpoonup}(T_P)$.

$\mathtt{u} \in_\top \mathtt{v},\ \mathtt{u}, \mathtt{v} \in \mathcal{N}_P$ is called an *inside-first-execution relation* if $[(\mathtt{v}, ent), (\mathtt{u}, ent)]$ is a sublist of T_P. $\mathtt{u} \in_\bot \mathtt{v}$ is called an *inside-last-execution relation* if $[(\mathtt{u}, ext), (\mathtt{v}, ext)]$ is a sublist of T_P. $S^{\in_\top}(T_P)$ is the set of all inside-first-execution relations in a program trace T_P, $S^{\in_\bot}(T_P)$ is the set of all inside-last-execution relations. In the following, we drop T_P when it is clear from the context.

Based on the execution relations defined above, we can now try to identify crosscutting concerns in software systems. *Recurring* execution relations in the program traces can be seen as indicators for more general execution patterns. To decide under which circumstances certain execution relations are recurring patterns in traces and thus potential crosscutting concerns in a system, constraints have to be defined. The constraints will implicitely also formalize what crosscutting means. However, for technical reasons we have to encode that there is no further method execution between nested method executions or between method invocation and method exit. This absence of method executions is represented by the designated empty method signature ϵ. Therefore, the definition of execution relations is extended such that each sublist of a program trace T_P induces not only relations defined above but also additional relations involving ϵ. The program trace remains as defined before with method signatures from $\mathcal{N}_P$ whereas the execution relations now can consist of method signatures from $\mathcal{N}_P \cup \{\epsilon\}$. Thus, the sets $S^{\rightharpoonup}$, $S^{\leftharpoonup}$, $S^{\in_\top}$, and $S^{\in_\bot}$ also include execution relations involving ϵ. Now, we can define the constraints for the dynamic analysis.

Formally, an execution relation $s = \mathtt{u} \circ \mathtt{v} \in S^\circ$, $\circ \in \{\rightharpoonup, \leftharpoonup, \in_\top, \in_\bot\}$, is called *uniform* if $\forall \mathtt{w} \circ \mathtt{v} \in S^\circ : \mathtt{u} = \mathtt{w}, \mathtt{u}, \mathtt{v}, \mathtt{w} \in \mathcal{N}_P \cup \{\epsilon\}$ holds, i.e., it exists in always the same composition. $\widehat{U}^\circ$ is the set of execution relations $s \in S^\circ$ which satisfy this requirement. This constraint is easy to explain. Consider an outside-before-execution relation $\mathtt{u} \rightharpoonup \mathtt{v}$. This is defined as recurring pattern if each execution of $\mathtt{v}$ is preceded by an execution of $\mathtt{u}$. The argumentation for outside-after-execution relations is analogous. The uniformity-constraint also applies to inside-execution relations. An inside-execution relation $\mathtt{u} \in_\top \mathtt{v}$ (or $\mathtt{u} \in_\bot \mathtt{v}$) can only be a recurring pattern in the given program trace if $\mathtt{v}$ never executes another method than $\mathtt{u}$ as first (or last) method inside its body.

We now drop the ϵ-relations and define two further analysis constraints: An execution relation $s = \mathtt{u} \circ \mathtt{v} \in U^\circ$ is called *non-trivial* if $s \in_k U^\circ, k > 1$ holds, i.e., it occurs more than once in the program trace T_P. R° is the set of execution relations $s \in U^\circ$ that satisfy this requirement. An execution relation $s = \mathtt{u} \circ \mathtt{v} \in U^\circ = \widehat{U}^\circ \backslash \{\mathtt{u} \circ \mathtt{v} \mid \mathtt{u} = \epsilon \vee \mathtt{v} = \epsilon\}$ is called *crosscutting* if $\exists s' = \mathtt{u} \circ \mathtt{w} \in U^\circ : \mathtt{w} \neq \mathtt{v}, \mathtt{u}, \mathtt{v}, \mathtt{w} \in \mathcal{N}_P$ holds, i.e., it occurs in more than a single calling context in the program trace T_P. For inside-execution relations $\mathtt{u} \in_\top \mathtt{v}$ (or $\mathtt{u} \in_\bot \mathtt{v}$) the calling context is the surrounding method execution $\mathtt{v}$. For outside-execution relations $\mathtt{u} \rightharpoonup \mathtt{v}$ (or $\mathtt{u} \leftharpoonup \mathtt{v}$) the calling context is the method $\mathtt{v}$ invoked before (or after) which always method $\mathtt{u}$ is executed. R°_C is the set of execution relations $s \in U^\circ$ which satisfy this requirement. Execution relations $s \in R^\circ$ and $s \in R^\circ_C$ resp. are also called *aspect candidates* as they represent the potential crosscutting concerns of the analysed software system.

The described constraints can be implemented by two relatively straightforward algorithms *basic* and *crosscutting algorithm* resp., in order to actually compute the sets R° of uniform, non-trivial execution relations, and the sets R°_C of uniform, crosscutting execution relations that represent the aspect candidates.

2.2. DynAMiT Case Studies

Case Study "Graffiti" Graffiti [5] is an industrial-sized editor for graphs and a toolkit for implementing graph visualisation algorithms, developed using Java. It currently comprises about 450 interfaces and classes, 3.100 methods and 82.000 lines, including comments. A tracing aspect, written in AspectJ, has been woven into the existing Graffiti system and the system obtained has been executed in seven different runs. In total, the traces consist of 33706 events. The analysis revealed 40 aspect candidates from before-execution relations, 40 from after-execution relations, 33 from first-execution relations, and 25 from last-execution relations. Those numbers show that the amount of aspect candidates stays relatively small compared to the software system's size. Moreover, the candidates themselves are quite compact; on average, a candidate exists of about four pairs of relations.

The case study showed that, in particular, *DynAMiT* has detected the typical logging concern in Graffiti. The analysis of the program traces found several calls to a method `format(LogRecord record)` of class `SimpleFormatter` as first and/or last call inside several set- and add-methods. A code investigation revealed that all executions of those methods are logged in a log-file. For that, a logger provided by Java's class `Logger` is used. We have not traced calls to the Java API but the logger uses a formatter to transform the system's log messages. The API provides an abstract class `Formatter` which is implemented by several special formatter classes but Graffiti's developers have chosen to write their own class `SimpleFormatter` implementing only basic functionality. The analysis detects the formatting of the log-messages and therefore, the crosscutting logging functionality is revealed and can be encapsulated into an aspect in a re-engineering process.

Graffiti can easily be extended with graph algorithms by writing plugins. Before a plugin can be used, it has to be registered, which requires a unique string as identifier. Thus, every plugin has to implement method `getName` from interface `Algorithm` that provides the name of the corresponding algorithm. This architectural principle is reflected in aspect candidates identified by *DynAMiT*. In all appropriate algorithm classes, `getName` is always preceded by a call to `getAlgorithms` of class `GenericPluginAdapter`. Since Graffiti contains thirteen different algorithm plugins, *DynAMiT* detects thirteen individual aspect candidates; an automatic grouping reveals that they all reflect the same architecture.

In summary, the analysis has shown that a lot of the functionality concerning actions like opening, saving, or editing files or graphs is crosscutting Graffiti's architecture. It is worth to consider restructuring the system accordingly. Additionally, *DynAMiT* provides a lot of information about the control flow within the Graffiti system and about its overall architecture. Thus, the lightweight dynamic aspect mining approach has easily helped to understand both crosscutting concerns in the system and the system itself.

Case Study "AspectJ example `telecom`" A small case study has been conducted in order to verify how successful the developed analysis approach can be applied to a new problem: Can the Java-AspectJ implementation of *DynAMiT* also detect crosscutting concerns in Java programs which are already extended by aspects written in AspectJ?

For that purpose the telecom example which is included in the distribution of AspectJ has been chosen: A small simulation where one person calls another person and then the second person calls a third person is included. The simulation can be executed at three different levels: `BasicSimulation` just performs the calls with the basic functionality needed for making phone calls (call, accept, hang up etc.). `TimingSimulation` is the extension of `BasicSimulation` with a timing aspect which keeps track of a connection's duration and cumulates a customer's connection durations. `BillingSimulation` is a further extension with a billing aspect that adds functionality to calculate charges for phone calls of each customer based on connection type and connection duration. All three simulations have been traced and the resulting program traces have been fed into *DynAMiT*. A comparison of the analysis results for the three simulation versions (basic, timing, billing) clearly shows that the presented approach identifies basic functionality and the functionality added by the two different aspects. The detected concerns tell the user in a simple way what functionality the application has and what it does. They are even easier and faster to understand than a code investigation. Reading the analysis results is like reading a manual of the progression of the different steps in a phone call. Of course, this is supported by the fact that the simulation developers did choose meaningful method names: The method signatures themselves give the information what the methods perform so that analysis results as the following can be interpreted easily:

```
void telecom.Call.hangup(Customer) ⇀
    void telecom.Customer.removeCall(Call)
```

A verification of the analysis results based on code investigation certifies the developed approach to be sound. It captures the whole functionality added by the timing aspect. The same applies for the billing aspect, except that only one after-advice is not detected. This is due to its implementation: A public (!) field called `payer` of the connection is set directly. This is contrary to object-oriented design principles, which would suggest a private field with appropriate set- and get-methods. Unfortunately, only the get-method is realised. As field accesses are not interesting for run-time

behaviour, they have not been traced. Thus, they cannot be detected by *DynAMiT*.

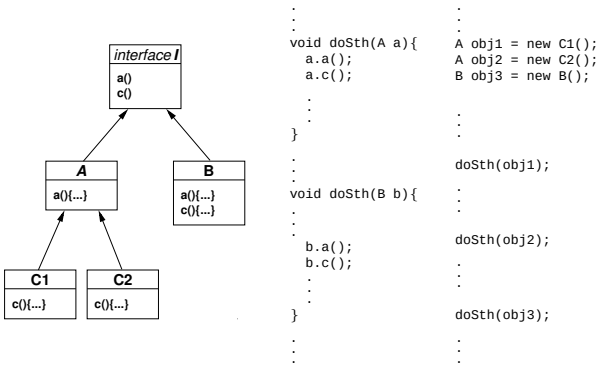
3. Evaluation

All conducted case studies show that the presented dynamic analysis approach fulfils its task with high precision. It finds crosscutting concerns in small tools as well as in industrial-sized systems. Furthermore, the introduced aspect mining technique detects crosscutting functionality which was added to systems following the AOP paradigm.

In order to work as intended the approach relies on proper tracing of executed programs. However, *DynAMiT* uses AspectJ and is thus dependent on the implementation of AspectJ. Therefore, the tool relies on an important point: Functionality has to be encapsulated into methods as assignments like `int x = 42;` are not traceable with AspectJ. This fact leads to a certain degree of impression which can be both, good or disadvantageous. If an assignment is essential in every occurrence of a specific execution relation as it changes object values used in one of the involved methods, it is a disadvantage that the analysis results do not automatically provide that information. On the other hand, if an assignment is not necessary in each case (maybe because it is dependent on certain program conditions), it is good that the analysis—especially the uniformity-constraint—does not consider those assignments. The case distinctions can be made once the detected crosscutting concern will be implemented as an aspect.

There are some more drawbacks due to limitations of AspectJ. Java API method executions do not appear in the program traces if the classes itself are not present as source code. On the one hand, it is good that those methods are not traced as with each analysis we would also analyse Java API classes. But on the other hand, this leads to false positives or imprecise aspect candidates in the analysis. Furthermore, noise in the resulting aspect candidates caused by dynamic binding complicates their retrieval in the program's source code. This problem also exists partly due to AspectJ. Thus, we can say that the realisation of the dynamic aspect mining approach suffers from certain AspectJ implementation details, which cause some imprecision and incompleteness in the analysis results.

Moreover, both the basic and the crosscutting algorithm yield redundant aspect candidates. This especially happens if symmetric relations exist in the program traces. For inside-aspect candidates this sometimes means, that they really exist twice and in both directions, e.g. in `A.b(){ C.d(){} ... C.d(){} }`, and sometimes not, e.g. in `A.b(){ C.d(){} }`; in both situations the analysis would produce two inside-aspect candidates: `C.d()` $\in_\top$ `A.b()`, and `C.d()` $\in_\bot$ `A.b()`. An inside- and an outside-aspect candidate together can also



(a) Inheritance hierarchy (b) Code fragments

Figure 1. Example excerpt of a software system

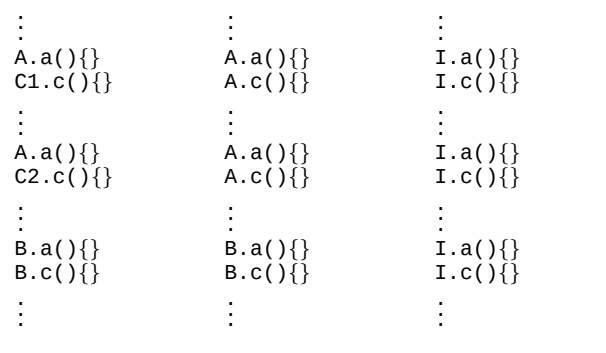
be redundant. Consider a program trace fragment like `A.b()` { `B.c()`{} `C.d()` {} ...}. Then it can happen, that the analysis identifies `B.c()` $\in_\top$ `A.b()` and `B.c()` $\rightharpoonup$ `C.d()`, which is redundant in this case. Therefore, the analysis does not provide perfect results which can immediately be transfered into aspects without further program code investigation, but it gives clear descriptions of recurring execution patterns and helps the developer to understand a system's behaviour and to re-factor it faster and easier.

To summarise, we can say that the aspect mining technique was able to identify automatically and with high precision both seeded and existing crosscutting concerns in the software systems while producing only a relatively small number of false positives. Furthermore, the results provided by both algorithms provided additional insights in the programs' general behaviour and architecture.

4. Extensions to the Analysis Algorithms

Since the presented algorithms work on method signatures only, they can produce false positives due to dynamic binding at run-time, i.e., methods with the same name but defined in different classes can get identified and thus result in wrong aspect candidates. It could thus be helpful to extend the existing data structure and relations with static information about class name and/or line number where a method call is located in the source file. Details about the static type of an object would also improve the outcomes of the analysis.

Figure 1 outlines a small example to illustrate those improvements. Figure 1(a) shows a part of the inheritance tree for the example: The `interface I` has two method declarations `a()` and `c()`. The `class B` implements that interface, while the `abstract class A` only implements method `a()` of `I`. The `abstract class A` is extended by



(a) 'Traditionally' dynamic (b) With static object info (c) With inheritance info

Figure 2. Dynamic vs 'static' vs 'inherited' trace

two subclasses, namely `C1` and `C2` which both provide implementations of method `c()` whose declaration is inherited from `I` (via `A`). Assume that the code fragments shown in 1(b) exist in the system and are executed.

This scenario could result in traces including the part shown in 2(a). There, the crosscutting algorithm would identify incorrect before-aspect candidates `A.a` $\rightharpoonup$ `C1.c`, and `A.a` $\rightharpoonup$ `C2.c`. This kind of functionality exists only once in the code in `void doSth(A a)`. If we now consider for example the static type of the objects in the traces, the program trace will look different, as we see in 2(b). In turn, this would result in the crosscutting algorithm not detecting the incorrect crosscutting concerns mentioned above (which may be part of a real crosscutting concern, but are none on their own). A similar improvement can be achieved if the dynamic trace is augmented by the line number and source file where a method call is located. Thus, an integration of some or all of this static information into the traces and the analysis could often avoid that an invocation of the same functionality (i.e., one occurring only once in the code) appears to be crosscutting in the traces.

The dynamic approach was chosen to monitor real runtime behaviour of software systems. However, there are different facets in run-time behaviour which can be of interest. While sometimes we want to know which method implementation is used at run-time, the approach presented in [3] is based on the dynamic information which functionalities are executed after or within what other functionalities. Thus, we could discard the run-time information about the used implementation while executing methods and use the fully-qualified signature of the method declarations, instead. That information can be extracted from the inheritance hierarchy. For the small example in Figure 1 this would result in an "inherited" trace (shown in Figure 2(c)). Instead of identifying only parts of the crosscutting concern (as with the crosscutting algorithm in the "traditional" dynamic trace), the basic algorithm would now find the full and real crosscutting concern: Methods `a` and `b` are invoked in succession at different places in the code. Together with the static information about source file and line number proposed before, the developer would easily find the appropriate occurrences of that pattern in the code. Thus, an impact in recall could be achieved by combining the traces with information of a program's inheritance hierarchy before the analysis algorithms are applied to the obtained aspect candidates. The analysis results are then more accurate as noise produced by dynamic binding is gone.

References

[1] Silvia Breu. Aspect Mining Using Event Traces. Master's thesis, U Passau, Germany, March 2004.

[2] Silvia Breu. Case Studies in Aspect Mining. 6. Workshop Software-Reengineering, Bad Honnef. In: Softwaretechnik-Trends, 24(2), pp. 30–32, 2004.

[3] Silvia Breu and Jens Krinke. Aspect Mining Using Event Traces. In *Proc. 19th Conf. Automated Software Engineering*. IEEE Press, 2004.

[4] William G. Griswold, Y. Kato, and J. J. Yuan. Aspect Browser: Tool Support for Managing Dispersed Aspects. Technical Report CS99-0640, UC, San Diego, 1999.

[5] Gravisto homepage. http://www.gravisto.org.

[6] Andreas Gal, Wolfgang Schroeder-Preikschat, and Olaf Spinczyk. AspectC++: Language Proposal and Prototype Implementation. In *Proc. OOPSLA Workshop on Language Mechanisms for Programming Software Components*, 2001.

[7] Jan Hannemann and Gregor Kiczales. Overcoming the Prevalent Decomposition of Legacy Code. In *Workshop on Advanced Separation of Concerns*, 2001.

[8] Doug Janzen and Kris De Volder. Navigating and Querying Code Without Getting Lost. In *2nd Intl. Conf. on Aspect-Oriented Software Development (AOSD)*, pp. 178–187, 2003.

[9] Georg Kiczales et. al. Aspect-Oriented Programming. In *ECOOP*, 1997.

[10] Neil Loughran and Awais Rashid. Mining Aspects. In *Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design (AOSD Sat. workshop)*, 2002.

[11] David Shepherd and Lori Pollock. Ophir: A Framework for Automatic Mining and Refactoring of Aspects. Technical Report 2004-03, U Delaware, 2003.

[12] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *ICSE-21*, pp. 107–119, 1999.

[13] Arie van Deursen, Marius Marin, and Leon Moonen. Aspect Mining and Refactoring. In *First Intl. Workshop on REFactoring: Achievements, Challenges, Effects (REFACE)*, 2003.

[14] Xerox PARC. *Aspect-Oriented Programming with AspectJ (Tutorial)*, 1998.

[15] Charles Zhang and Hans-Arno Jacobsen. Quantifying Aspects in Middleware Platforms. In *2nd Intl. Conf. on Aspect-Oriented Software Development (AOSD)*, pp. 130–139, 2003.

Control-Flow-Graph-Based Aspect Mining

Jens Krinke
FernUniversität in Hagen, Germany
krinke@acm.org

Silvia Breu
NASA Ames Research Center, USA
silvia.breu@gmail.com

Abstract

Aspect mining tries to identify crosscutting concerns in existing systems and thus supports the adaption to an aspect-oriented design. This paper describes an automatic static aspect mining approach, where the control flow graphs of a program are investigated for recurring execution patterns based on different constraints, such as the requirement that the patterns have to exist in different calling contexts. A case study done with the implemented tool shows that most discovered crosscutting candidates are most often perfectly good style.

1. Introduction

The notion of *tangled code* refers to code that exists several times in a software system but cannot be encapsulated by separate modules using traditional module systems because it crosscuts the whole system. This makes software more difficult to maintain, to understand, and to extend. *Aspect-Oriented Programming* [4] provides new separation mechanisms for such complex *crosscutting concerns* [7].

A major problem in re-engineering legacy code based on aspect-oriented principles is to find and to isolate these crosscutting concerns. This task is also called *aspect mining*. The detected concerns can be re-implemented as separate aspects, thereby improving maintainability and extensibility as well as reducing complexity. Aspect mining can also provide insights that enable us to classify common aspects which occur in different software systems, such as logging, timing, and communication.

Several approaches based on static program analysis techniques have been proposed for aspect mining [3, 5, 6, 10, 8, 2]. We have developed a dynamic program analysis approach [1] that mines aspects based on program traces. During program execution, program traces are generated, which reflect the run-time behavior of a software system. These traces are then investigated for recurring execution patterns. Different constraints specify when an execution pattern is "recurring". These include the requirement that the patterns have to exist in different calling contexts in the program trace. The dynamic analysis approach monitors actual (i.e., run-time) program behavior instead of potential behavior, as static program analysis does. To explore the differences between static and dynamic analyses in aspect mining, we have started to develop a static analysis variant of our approach. From early results we experienced two things:

- The results of the static and dynamic analysis are different due to various reasons.
- Crosscutting concerns are often perfectly good style, because they result from delegation and coding style guides.

The first point is obvious and thus, only the second point will be discussed in the following. The next section contains an introduction to our dynamic aspect mining approach. A static aspect mining approach based on the dynamic variant is presented in Section 3. Section 4 contains a case study, Section 5 discusses the results and concludes, before Section 6 discusses related work.

2. Dynamic Aspect Mining

The basic idea behind dynamic analysis algorithms is to observe run-time behavior of software systems and to extract information from the execution of the programs. The dynamic aspect mining approach introduced here is based on the analysis of program traces which mirror a system's behavior in certain program runs. Within these program traces we identify recurring execution patterns which describe certain behavioral aspects of the software system. We expect that recurring execution patterns are potential crosscutting concerns which describe recurring functionality in the program and thus are possible aspects.

In order to detect these recurring patterns in the program traces, a classification of possible pattern forms is required. Therefore, we introduce so-called *execution relations*. They describe in which relation two method executions are in the program trace.

2.1. Classification of Execution Relations

The definition of execution relations in our analysis approach is based on program traces. Intuitively, a program trace is a sequence of method invocations and exits. We only consider entries into and exits from method executions because we can then easily keep track of the relative order in which method executions are started and finished. We focus on method executions because we want to analyze object-oriented systems where logically related functionality is encapsulated in methods. Formally, a *program trace* T_P of a program P with method signatures $\mathcal{N}_P$ is defined as a list $[t_1, \ldots, t_n]$ of pairs $t_i \in (\mathcal{N}_P \times \{ent, ext\})$, where ent marks entering a method execution, and ext marks exiting a method execution.

To make the program traces easier to read, the ent- and ext-points are represented by { and } respectively, and the redundant $name$-information is discarded from the ext-points as the trace structure implies to which $name$ the ext belongs. Figure 1 shows an example trace.

```
 1 B() {          17   J() {}       33   }
 2   C() {        18 }               34 }
 3     G() {}     19 F() {           35 D() {
 4     H() {}     20   K() {}       36   C() {}
 5   }            21   I() {}       37   A() {}
 6 }              22 }               38   B() {
 7 A() {}         23 J() {}          39     C() {}
 8 B() {          24 G() {}          40   }
 9   C() {}       25 H() {}          41   K() {}
10 }              26 A() {}          42   I() {
11 A() {}         27 B() {           43     J() {}
12 B() {          28   C() {}       44   }
13   C() {        29   G() {}       45   G() {}
14     G() {}     30   F() {         46   E() {}
15     H() {}     31     K() {}     47 }
16   }            32     I() {}
```

Figure 1. Example trace

Crosscutting concerns are now reflected by the two different *execution relations* that can be found in program traces: A method can be executed either after the preceding method execution is terminated (e.g., `H()` in line *4* is executed after `G()` in line *3*), or inside the execution of the preceding method call (e.g., `C()` in line *2* is executed inside `B()` in line *1*). We distinguish between these two cases and say that there are outside- and inside-execution relations in program traces. However, this distinction alone is not yet sufficient for aspect mining. For example, the execution of `B()` in line *27* has three methods executed inside its execution, `C()`, `G()`, and `F()` in lines *28* ff., but the information which of those methods comes first is lost. We thus define formally:

$u \rightharpoonup v,\ u, v \in \mathcal{N}_P$, is called an *outside-before-execution relation* if $[(u, ext), (v, ent)]$ is a sublist of T_P. $S^{\rightharpoonup}(T_P)$ is the set of all outside-before-execution relations in a program trace T_P. This relation can also be reversed, i.e., $v \leftharpoonup u$ is an *outside-after-execution relation* if $u \rightharpoonup v \in S^{\rightharpoonup}(T_P)$. The set of all outside-after-execution relations in a program trace T_P is then denoted with $S^{\leftharpoonup}(T_P)$.

$u \in_\top v,\ u, v \in \mathcal{N}_P$ is called an *inside-first-execution relation* if $[(v, ent), (u, ent)]$ is a sublist of T_P. $u \in_\bot v$ is called an *inside-last-execution relation* if $[(u, ext), (v, ext)]$ is a sublist of T_P. $S^{\in_\top}(T_P)$ is the set of all inside-first-execution relations in a program trace T_P, $S^{\in_\bot}(T_P)$ is the set of all inside-last-execution relations. In the following, we drop T_P when it is clear from the context.

For the example trace shown in Figure 1 we thus get the following set $S^{\rightharpoonup}$ of outside-before-execution relations:

$$\begin{aligned} S^{\rightharpoonup} = \{ & \texttt{B()} \rightharpoonup \texttt{A()}, \texttt{G()} \rightharpoonup \texttt{H()}, \texttt{A()} \rightharpoonup \texttt{B()}, \texttt{C()} \rightharpoonup \texttt{J()}, \\ & \texttt{B()} \rightharpoonup \texttt{F()}, \texttt{K()} \rightharpoonup \texttt{I()}, \texttt{F()} \rightharpoonup \texttt{J()}, \texttt{J()} \rightharpoonup \texttt{G()}, \\ & \texttt{H()} \rightharpoonup \texttt{A()}, \texttt{B()} \rightharpoonup \texttt{D()}, \texttt{C()} \rightharpoonup \texttt{G()}, \texttt{G()} \rightharpoonup \texttt{F()}, \\ & \texttt{C()} \rightharpoonup \texttt{A()}, \texttt{B()} \rightharpoonup \texttt{K()}, \texttt{I()} \rightharpoonup \texttt{G()}, \texttt{G()} \rightharpoonup \texttt{E()} \} \end{aligned}$$

The set $S^{\leftharpoonup}$ of outside-after-execution relations can be found directly in the trace or simply by reversing $S^{\rightharpoonup}$. The sets $S^{\in_\top}$ of inside-first-execution relations and $S^{\in_\bot}$ of inside-last-execution relations are as follows:

$$\begin{aligned} S^{\in_\top} = \{ & \texttt{C()} \in_\top \texttt{B()}, \texttt{G()} \in_\top \texttt{C()}, \texttt{K()} \in_\top \texttt{F()}, \texttt{C()} \in_\top \texttt{D()}, \\ & \texttt{J()} \in_\top \texttt{I()} \} \\ S^{\in_\bot} = \{ & \texttt{H()} \in_\bot \texttt{C()}, \texttt{C()} \in_\bot \texttt{B()}, \texttt{J()} \in_\bot \texttt{B()}, \texttt{I()} \in_\bot \texttt{F()}, \\ & \texttt{F()} \in_\bot \texttt{B()}, \texttt{J()} \in_\bot \texttt{I()}, \texttt{E()} \in_\bot \texttt{D()} \} \end{aligned}$$

2.2. Execution Relation Constraints

Recurring execution relations in the program traces can be seen as indicators for more general execution patterns. To decide under which circumstances certain execution relations are recurring patterns in traces and thus potential crosscutting concerns in a system, constraints have to be defined. The constraints will implicitly also formalize what crosscutting means.

However, for technical reasons we have to encode that there is no further method execution between nested method executions or between method invocation and method exit. This absence of method executions is represented by the designated empty method signature ϵ. Therefore, the definition of execution relations is extended such that each sublist of a program trace T_P induces not only relations defined above but also additional relations involving ϵ. Table 1 summarizes this conservative extension. It shows for each two-element sublist of the trace (on the left side) the execution relations that follow from that sublist (on the right side). The execution relations added by the introduction of ϵ are annotated with an asterisk ($*$).

The program trace remains as defined before with method signatures from $\mathcal{N}_P$ whereas the execution relations now can consist of method signatures from $\mathcal{N}_P \cup \{\epsilon\}$.

Trace-sublist $(\mathcal{N}_P)$		Relation s $(\mathcal{N}_P \cup \{\epsilon\})$
(u, ext)	(v, ent)	$u \rightharpoonup v, v \leftharpoonup u$
(v, ent)	(u, ent)	$\epsilon \rightharpoonup u^*, u \leftharpoonup \epsilon^*, u \in_\top v$
BOL	(u, ent)	$\epsilon \rightharpoonup u^*, u \leftharpoonup \epsilon^*, u \in_\top \epsilon^*$
(u, ext)	(v, ext)	$u \rightharpoonup \epsilon^*, \epsilon \leftharpoonup u^*, u \in_\bot v$
(u, ext)	EOL	$u \rightharpoonup \epsilon^*, \epsilon \leftharpoonup u^*, u \in_\bot \epsilon^*$
(w, ent)	(w, ext)	$\epsilon \in_\top w^*, \epsilon \in_\bot w^*$

BOL/EOL denote begin/end of list

Table 1. Extended execution relations

Thus, the sets $S^{\rightharpoonup}$, $S^{\leftharpoonup}$, $S^{\in_\top}$, and $S^{\in_\bot}$ also include execution relations involving ϵ. Now, we can define the constraints for the dynamic analysis.

Formally, an execution relation $s = u \circ v \in S^\circ$, $\circ \in \{\rightharpoonup, \leftharpoonup, \in_\top, \in_\bot\}$, is called *uniform* if $\forall w \circ v \in S^\circ : u = w, u, v, w \in \mathcal{N}_P \cup \{\epsilon\}$ holds, i.e., it exists in always the same composition. $\widehat{U}^\circ$ is the set of execution relations $s \in S^\circ$ which satisfy this requirement. This constraint is easy to explain. Consider an outside-before-execution relation $u \rightharpoonup v$. This is defined as recurring pattern if each execution of v is preceded by an execution of u. The argumentation for outside-after-execution relations is analogous. The uniformity-constraint also applies to inside-execution relations. An inside-execution relation $u \in_\top v$ (or $u \in_\bot v$) can only be a recurring pattern in the given program trace if v never executes another method than u as first (or last) method inside its body.

We now drop the ϵ-relations and define a further analysis constraint: An execution relation $s = u \circ v \in U^\circ = \widehat{U}^\circ \backslash \{u \circ v \mid u = \epsilon \vee v = \epsilon\}$ is called *crosscutting* if $\exists s' = u \circ w \in U^\circ : w \neq v,\ u, v, w \in \mathcal{N}_P$ holds, i.e., it occurs in more than a single calling context in the program trace T_P. For inside-execution relations $u \in_\top v$ (or $u \in_\bot v$) the calling context is the surrounding method execution v. For outside-execution relations $u \rightharpoonup v$ (or $u \leftharpoonup v$) the calling context is the method v invoked before (or after) which always method u is executed. R° is the set of execution relations $s \in U^\circ$ which satisfy this requirement. Execution relations $s \in R^\circ$ are also called *aspect candidates* as they represent the potential crosscutting concerns of the analyzed software system.

2.3. Aspect Mining Algorithm

The constraints described above can be implemented by a relatively straightforward algorithm to actually compute the sets R° of uniform, crosscutting execution relations that represent the aspect candidates. In our running example, uniformity narrows down the potential aspect candidates to the following sets of execution relations:

$$U^{\rightharpoonup} = \{\texttt{B()} \rightharpoonup \texttt{D()}, \texttt{G()} \rightharpoonup \texttt{E()}, \texttt{G()} \rightharpoonup \texttt{H()}, \texttt{K()} \rightharpoonup \texttt{I()}\}$$
$$U^{\leftharpoonup} = \{\texttt{B()} \leftharpoonup \texttt{A()}, \texttt{I()} \leftharpoonup \texttt{K()}\}$$
$$U^{\in_\top} = \{\texttt{C()} \in_\top \texttt{B()}, \texttt{C()} \in_\top \texttt{D()}, \texttt{K()} \in_\top \texttt{F()}\}$$
$$U^{\in_\bot} = \{\texttt{E()} \in_\bot \texttt{D()}, \texttt{I()} \in_\bot \texttt{F()}\}$$

After we enforce the crosscutting constraint, we obtain the final sets R° of aspect candidates which comply with uniformity *and* crosscutting.

$$R^{\rightharpoonup} = \{\texttt{G()} \rightharpoonup \texttt{H()}, \texttt{G()} \rightharpoonup \texttt{E()}\},\ R^{\leftharpoonup} = \varnothing$$
$$R^{\in_\top} = \{\texttt{C()} \in_\top \texttt{B()}, \texttt{C()} \in_\top \texttt{D()}\},\ R^{\in_\bot} = \varnothing$$

3. Static Aspect Mining

Based on the experience with the dynamic approach, we implemented a similar static analysis. This analysis extracts the execution relations from a control flow graph of the analyzed program. In particular, we immediately extract uniform and crosscutting execution relations without a previous step to extract unconstrained execution relations. However, the extraction is different for outside and inside execution relations. Here, we will only present inside-first ($R^{\in_\top}$) and outside-before ($R^{\rightharpoonup}$) execution relations.

Inside-First Execution Relations. For these kind of execution relations, we extract the method invocations immediately following the entry of (invoked) methods from the control flow graph. Such a relation is uniform, if every path through the method starts with the same method call. Moreover, a possible simplification just considers the single-entry-single-exit regions starting at the methods' entry nodes. Such a relation $u \in_\top v$ means now that method u is the first method invocation inside the single-entry-single-exit region starting at the entry node of method v. The definition of crosscutting stays the same, thus u is a crosscutting method invocation if there are at least two uniform execution relations $u \in_\top v$ and $u \in_\top w$ ($v \neq w$).

Outside-Before Execution Relations. Here we extract all pairs of method invocations u, v if there exists a path from an invocation of method u to an invocation of method v without any method invocation in between. Such a pair is a uniform outside-before execution relation $u \rightharpoonup v$, if all paths from an invocation of method u contain an invocation of v as the next invocation. The first possible simplifications is to require that an invocation of u is post-dominated by an invocation of v without another invocation in between. The second simplifications is to require that any invocation of method u is followed by an invocation of v in all single-entry-single-exit regions containing an invocation of u.

4. Experiences

We have implemented the presented static mining on top of the Soot framework [9], which is used to compute the

size	relations	size	relations
2	127	13	4
3	55	15	2
4	30	16	1
5	12	17	2
6	9	18	1
7	7	19	1
8	7	20	1
9	3	22	1
10	3	24	2
11	3	32	1
12	4	49	1

1236 relations ($R^{\in\top}$) in 277 candidates

Table 2. Inside-First Execution Relations

size	relations	size	relations
2	53	8	1
3	19	9	1
4	4	11	1
5	6	12	1
6	3	13	1
7	2		

294 relations ($R^{\rightharpoonup}$) in 92 candidates

Table 3. Outside-Before Execution Relations

control flow graph of the analyzed program. Our tool traverses these control flow graphs and extracts the uniform and crosscutting inside-first and outside-before execution relations. As a first test case we have analyzed JHotDraw, version 5.4b1. Tables 2 and 3 show the results. For inside-first execution relations, the tool has identified 277 candidates with 1236 uniform and crosscutting relations, and for outside-before relations, 92 candidates with 294 relations.

It is interesting, that there are many more candidates for inside-first than for outside-before. Furthermore, there are a lot of candidates with just a small amount of crosscutting, e.g., 127 candidates that just crosscut two methods.

We will next discuss some of the identified candidates in detail. However, due to the large amount of identified candidates, we will only present the six largest candidates of each category.

4.1. Inside-First Relations

The largest candidate consists of 49 uniform and crosscutting execution relations. The invoked method is "*...CollectionsFactory.current*". It is obvious that this is a method to access the current factory object, needed in many other methods of the system. This is clearly crosscutting, however, not a refactorable aspect.

The second largest candidate consists of 32 relations for the method "*...DrawingView.view*". This is again an accessor method that returns the currently active view. Thus, it is crosscutting but not refactorable.

The same holds for the third and fourth candidate, which both consist of 24 relations. The relevant methods are "*...DecoratorFigure.getDecoratedFigure*" and "*...AbstractHandle.owner*" which are once again accessor methods.

For the fifth candidate, things are not different: It consists of 22 relations for the method "*...UndoadableAdapter.undo*" that checks whether the current object represents an undo-able action.

Things change for the sixth candidate consisting of 20 candidates for method "*...AbstractFigure.willChange*". That method informs a figure that an operation will change the displayed content. This is the first candidate that is a crosscutting concern which could be refactored into an aspect.

4.2. Outside-Before Relations

The largest discovered candidate consists of 13 uniform and crosscutting execution relations for the method "*...Iterator.next*". A closer look to the 13 invocations reveals that this crosscutting is more or less incidental: An operation is performed on the next element of a container.

The second largest candidate is somewhat interesting: It consists of 12 invocations before a call to "*...AbstractCommand.execute*", from which 11 are invocations of method "*createUndoActivity*". The other is an invocation of "*...ZoomDrawingView.zoomView*", which seems to be an *anomaly*. However, the other 12 invocations are of classes representing operations that change the figure and *zoomView* (probably) does not change it.

The next three largest candidates (consisting of 11, 9, and 8 relations) are again more or less incidental crosscutting concerns related to methods "*...DrawingView.drawing*", "*...List.add*", and "*...DrawingView.view*". However, it is interesting to see that *DrawingView.view* was also part of a large inside-first candidate.

Again, only the sixth largest candidate can be seen as crosscutting concern that can be refactored into an aspect. It consists of seven relations for method "*...AbstractFigure.willChange*". It is immediately called before methods that will change the displayed figure. However, it is interesting to see that this method has also appeared as an inside-first candidate, where the candidate is larger (20 relations).

5. Discussion, Conclusions, and Future Work

This initial evaluation of the static aspect mining tool has shown that most of the identified crosscutting candidates are not concerns refactorable into aspects. This is not much different from results in our previous dynamic aspect mining [1]. However, both approaches give interesting insights into the crosscutting behavior of the analyzed program. Moreover, as seen in the example for method *AbstractCommand.execute*, they can probably be used to discover *crosscutting anomalies*, an anomaly in the discovered execution relation pattern.

These results are preliminary because of the small amount of analyzed candidates (12) in a single test program. However, based on the previous results from the dynamic approach, our hypothesis is that the results will not change and are general. This would mean that aspect mining will have hard times to identify candidates that are really refactorable into aspects. Therefore, future work will continue in three directions:

1. A large-scale analysis of discovered candidates for a large set of programs with static and dynamic analysis.
2. Development of a filter which extracts the refactorable candidates from the discovered candidates.
3. A comparison with other aspect mining approaches.

6. Related Work

There only exists a small set of automatic aspect mining approaches. In most approaches one has to specify a pattern that can be searched for in the source code [3, 10].

Tourwe [8] uses concept analysis to identify aspectual views in programs. The extraction of elements and attributes from the names of classes, methods, and variables, formal concept analysis is used to group those elements into concepts that can be seen as aspect candidates.

Some other approaches rely on clone detection techniques to detect tangled code in the form of crosscutting concerns:

Bruntink [2] evaluated the use of those clone detection techniques to identify crosscutting concerns. Their evaluation has shown that some of the typical aspects are discovered very well while some are not.

Ophir [6] identifies initial re-factoring candidates using a control-based comparison. The initial identification phase builds upon code clone detection using program dependence graphs. The next step filters undesirable re-factoring candidates. It looks for similar data dependencies in subgraphs representing code clones. The last phase identifies similar candidates and coalesces them into sets of similar candidates, which are the re-factoring candidate classes.

References

[1] S. Breu and J. Krinke. Aspect mining using event traces. In *Proc. International Conference on Automated Software Engineering*, pages 310–315, 2004.

[2] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwe. An evaluation of clone detection techniques for identifying cross-cutting concerns. In *Proc. International Conference on Software Maintenance*, 2004.

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

[4] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *European Conf. on Object-Oriented Programming (ECOOP)*, 1997.

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

[6] D. Shepherd and L. Pollock. Ophir: A Framework for Automatic Mining and Refactoring of Aspects. Technical Report 2004-03, University of Delaware, 2003.

[7] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *21st Intl. Conf. on Software Engineering (ICSE)*, pages 107–119, 1999.

[8] T. Tourwe and K. Mens. Mining aspectual views using formal concept analysis. In *Proc. IEEE International Workshop on Source Code Analysis and Manipulation*, 2004.

[9] R. Vallee-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot – a java bytecode optimization framework. In *Proc. CASCON*, 1999.

[10] C. Zhang and H.-A. Jacobsen. Quantifying Aspects in Middleware Platforms. In *2nd Intl. Conf. on Aspect-Oriented Software Development (AOSD)*, pages 130–139, 2003.

Fast Prototyping and Evaluation of Aspect Mining Analyses via Timna

David Shepherd[1], Jeffrey Palm[2], and Lori Pollock[1]

[1]Computer and Information Sciences, University of Delaware, Newark, DE 19716
[2]Computer and Information Sciences, Northeastern University, Boston, MA 02115

ABSTRACT

In this paper, we show how Timna, our framework for combining aspect mining analyses, enables quick prototyping and evaluation of new mining analyses. Timna can quickly determine the contribution of a new analysis to the aspect mining field by measuring the increase in effectiveness that the new analysis achieves.

1. INTRODUCTION

Aspect Oriented Software Development (AOSD) is a software development paradigm that enables the separation of concerns. AOSD allows programmers to modularize certain concerns that would be scattered throughout code in other paradigms. Although AOSD improves code readability and maintainability, software developers do not necessarily apply AOP techniques in every situation that would produce benefits. They typically avoid porting legacy systems into AOSD, primarily due to (currently) high cost. Developers also miss many opportunities to apply AOSD, due to their limited understanding of AOSD and their limited view of code. Any automation of the process of identifying potential aspects would increase the application of AOSD to new software and ease the process of porting legacy programs.

The process of aspect mining involves three phases: The identification of refactoring candidates (i.e. seeds) [8, 11, 3, 12, 7, 6], the expansion of candidates into full concerns [9, 13], and the refactoring of concerns [5]. We believe that the most important research problem to address first is the identification problem, because the other problems depend on the results from identifying candidates. Several researchers have already developed individual analyses that mark a piece of code (at differing granularities) as a candidate [8, 11, 3, 12, 7, 6]. However, there has been no examination of the potential combination of the results of these mining analyses. Intuitively, combining analyses' results should increase accuracy of a candidate identifier. If two analyses both agree that a piece of code is a candidate, then we can have more confidence in the marking than a marking by only one analysis. Our framework, Timna, builds upon this observation [10]. This paper focuses on describing how Timna enables quick prototyping and evaluation of individual analyses within the context of existing analyses. We also show how Timna allows researchers to quickly evaluate the contribution of new mining analyses.

2. TIMNA ANALYSIS FRAMEWORK

A detailed description of the theoretical foundations of Timna can be found in [10]. Timna is designed to leverage machine learning techniques to intelligently combine the results of many aspect mining analyses. As shown in Figures 1 and 2, Timna operates in two phases, the learning phase, and the classifying phase.

In the learning phase, Timna takes as input a training program that has been manually marked such that every marked method is considered to be a part of a concern that was not modularized well. Timna performs a set of analyses on this program, generating a set of attributes for each method. A classification table of method names, attributes, and classifications is then fed to a machine learning algorithm, which generates classification rules. The rules are propositional conditions over attributes which result in a classification. For example, if Timna was using the set of analyses:

{ Fan-in degree, Is Void, Has Parameters }

Mining Analyses would generate rows of:

<method name><number><boolean1><boolean2>

where *number* is the degree of fan-in in the call graph for that method, *boolean1* is true if the method's return type is `void`, and *boolean2* is true if the method has parameters. A rule generated by feeding a table of such rows into Machine Learning could be:

If(Fan-in>2 and IsVoid = true)
Then Classification = Is a candidate

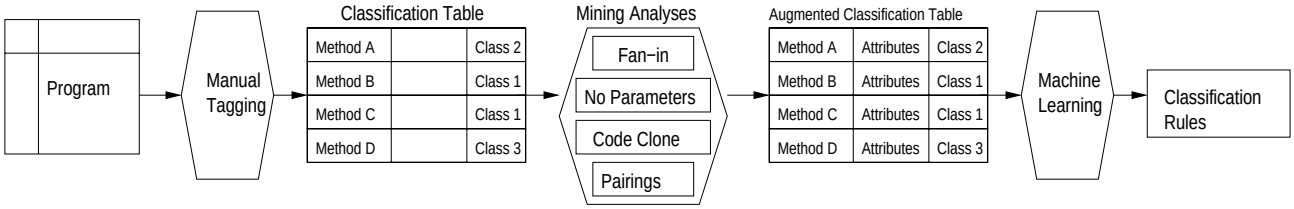


Figure 1: Learning Phase

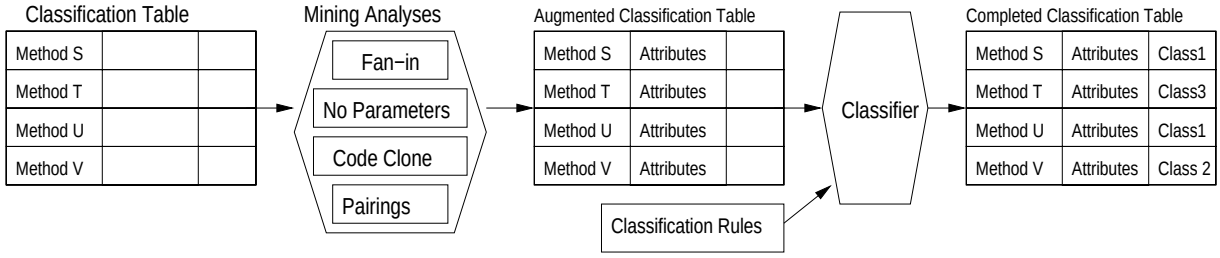


Figure 2: Classification Phase

In the classifying phase, the rules generated by the learning phase are used on new programs to classify methods as refactoring candidates.

3. FRAMEWORK IMPLEMENTATION

We implemented Timna as a combination of an Eclipse plug-in and several small Java applications. While using Eclipse to explore a Java project (in the Java Perspective) a developer can trigger Timna's mining analyses via a menu attached to any Java item (shown in Figure 3). The learning phase outputs the augmented classification table to a file which the user can specify in Eclipse's preferences.

Table 1 shows an example of the classification table, output by the learning phase. Here, the analyses' return types are specified at the top, next to each analysis's name. In the data section, the attribute vector, generated by each analysis, precedes the file and line number corresponding to each given method.

This classification table, during the learning phase, is processed by our Java application WekaCat or WekaBool, which both generate classification rules using a machine learning package [2]. WekaCat recognizes different categories of tagged candidates as distinct, while WekaBool simply considers methods as either a candidate or not. Either WekaCat or WekaBool could be used as the Machine Learning piece in Figure 1.

During the classification phase, users must manually input these rules into TestRules, which is our Java Application that uses the rules to classify a set of methods whose classification is unknown, or to check the effectiveness of rules on methods whose classification is known. TestRules outputs file names and line numbers

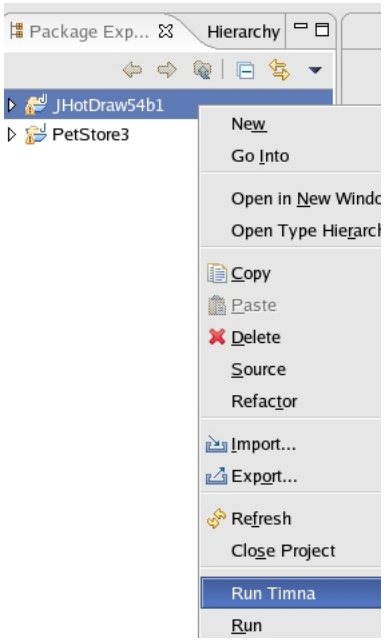


Figure 3: Begin Learning Phase

```
@relation isCandidate

@attribute FanInAnalysis NUMERIC
@attribute HighestPairingAnalysis NUMERIC
. . .
@attribute IsPublicAnalysis y,n
@attribute IsClassifiedAnalysis 0,1

@data
1,0,61,y,n,n,n,n,n,n,y,n,1,n,0,100,y,n,n,0 % PertFigure@172
4,25,0,n,n,n,y,n,n,n,y,n,1,n,0,0,y,n,n,0 % HTMLTextAreaFigure@1185
0,0,0,y,y,n,n,n,n,n,y,n,0,n,0,0,y,n,y,0 % PertFigure@39
0,0,0,n,n,n,y,n,n,n,y,n,0,n,0,0,y,n,y,1 % StandardDrawingView@919
. . .
```

Table 1: Intermediary Output of Learning Phase

```
static class A extends AnalysisAdapter.Numeric {
   public Object value(MethodDeclaration node) {
       . . .
       //implementation of analysis here
       . . .
       //return value for this analysis here
       return new Integer(someInt);
   }
}
```

Figure 4: Generic Analysis

```
public Object value(MethodDeclaration node) {
   Type t = node.getReturnType();
   boolean is = false;
   if(t instanceof PrimitiveType){
     is = ((PrimitiveType)t).
       getPrimitiveTypeCode()==PrimitiveType.VOID;
   }
   return Boolean.valueOf(is);
}
```

Figure 5: IsVoid Analysis' value method

that point to methods that it classifies as candidates. TestRules corresponds to the Classifier in Figure 2.

4. FAST PROTOTYPING OF ANALYSES

4.1 Methodology

Part of Timna's goal is to provide quick prototyping and evaluation for new aspect mining analyses. To this end, we attempt to provide users with a simple template for implementing new analyses, in the form of a subclass of AnalysisAdapter.Numeric (shown in Figure 4) or AnalysisAdapter.YesNo, which is very similar to AnalysisAdapter.Numeric but returns a boolean instead of a numeric value. These classes provide the method value that is executed on every method declaration in a program.

Prototyping an analysis in Timna is as simple as implementing the method value for a new analysis, and then including that new analysis in an array (specifically, the array that CallGraphAnalysis.analyses() returns). Timna runs all analyses in this array. The method value also provides analysis implementors with access to MethodDeclaration, which is the JDT component corresponding to that method [1]. This provides access to the powerful functionality of Eclipse's JDT, which we used to implement almost every analysis.

4.2 Case Studies

In order to illustrate how quickly and easily analyses can be implemented within the Timna Framework, we demonstrate the implementation of three analyses.

4.2.1 IsVoid

The first analysis, called IsVoid, returns a boolean attribute. The value of the attribute is true if the corresponding method's return type is void, false if it is not. Because of the power of Eclipse's JDT, the implementation of this analysis is only a few lines long. Specifically, the code (shown in Figure 5) accesses the JDT node, of type MethodDeclaration, and finds its return type. The code then checks to see if the return type is void, and returns the appropriate boolean value. When performing manual mining on code, we found that methods that had a void return type were often part of a poorly modularized concern.

4.2.2 OddLengthName

OddLengthName reports whether the length of a method's name is an odd number. Again, the code for this method, since it leverages the JDT, is fairly short (shown in Figure 6). We do not believe that this analysis provides any information that helps classify a piece of code. However, we include it in our evaluation to show how Timna will determine that a useless analysis is in fact useless. Finally, we discuss Fan-in analysis, because it is an example of a more complex analysis [8]. Fan-in analysis counts the number of incoming edges a method declaration has in a call graph, which is the number of call sites to that method in the system. Since Timna provides a call graph to users, the implementation of this analysis

```
public Object value(MethodDeclaration node) {
  SimpleName s = node.getName();
  return
    Boolean.valueOf(s.getIdentifier().length()%2==0);
}
```

Figure 6: OddLength Name Analysis' value method

```
public Object value(MethodDeclaration node) {
  CallGraph cg= callGraph();
  Collection c = cg.getCallers(node);
  return new Integer(c.size());
}
```

Figure 7: Fan-in Analysis' value method

is straightforward (shown in Figure 7). Marius et al. showed that the results of this analysis can be used to identify poorly modularized concerns [8].

5. ENABLING EVALUATION

5.1 Measures of Effectiveness

When researchers perform aspect mining, the two most important measures for evaluation of the miner's effectiveness are precision and recall, which are defined as follows, for a particular code:

> **Precision For Technique T** = *(Number Of Good Candidates Identified By T) / (Total Number Of Candidates Identified By T)*

> **Recall For T** = *(Number Of Good Candidates Identified By T) / (Total Number of Known Good Candidates)*

Precision is easier to measure, because we must only know or decide whether the results that technique T returns are good candidates. In order to calculate recall, we must know how many good candidates exist in the entire code base. Precision and recall are weakly complementary measures; the increase of one often leads to the decrease of the other [10].

5.2 Evaluation of Analyses with Timna

Timna was designed to facilitate the quick evaluation of mining analyses, especially in the context of existing analyses. Timna determines whether a new analysis helps improve overall effectiveness (i.e., gives new information) or whether it finds the same candidates as existing analyses.

In order to demonstrate Timna's usefulness in evaluating new analyses, we evaluated the IsVoid and OddLengthName analyses. For simplicity, we did not use Timna's full set of analyses during this evaluation, but only used:

Set	Recall	Precision
Base	2.6%	70.4 %
Base+IsVoid	21.1%	71.6%
Base+OddLengthName	4.8%	70.6 %

Table 2: Performance of Timna Configurations

{NoParameters, Fan-in, NumOfCallsAtBeginOrEnd}

as the existing set of analyses (we call this set Base). NoParameters analysis reports whether the analyzed method has any return value. NumOfCallsBeginOrEnd reports the percentage of calls to a method that are either at the beginning or the end of a method's body. Fan-in was described earlier. We first added OddLengthName to this set, and we determined the effectiveness of the generated rules on the training data. Then, we removed OddLengthName and added IsVoid to the existing analysis set, and we again measured the effectiveness. We performed measurements on the training data because the training data's classification is known, and we can thus calculate recall as well as precision.

In this study we measured the recall of the candidate methods and the precision of the overall system, in order to highlight the changes in effectiveness due to adding a single analysis.

5.3 Results

As Table 2 shows, the precision of all three configurations of Timna is similar, but Base+IsVoid performs the best by approximately one percent. When evaluating the effectiveness of these configurations, it is important to understand that the non-candidates greatly outnumber the candidates. The non-candidates are (in the base case) classified very accurately, and so the only improvements come from classifying the candidates more accurately than the base case. Therefore, we report the recall of the candidates (not the non-candidates) only. We still report the precision of the overall system, to demonstrate how the improvement of the candidate's recall increases the precision of the overall system. Table 2 shows that Base+IsVoid achieves much better recall than the base case or Base+OddLengthName, which leads to slightly better system precision as well.

6. COMBINING ANALYSES

Timna allows researchers to combine many analyses quickly and to determine which analyses are most relevant. There are two specific ways in which researchers can use Timna to evaluate the relative benefits of combining certain mining analyses.

Similar to the way we evaluated *new* analyses in our case study, researchers can use results from our framework to determine whether new mining analyses help improve effectiveness of Timna. If analyses do improve Timna's effectiveness, that means that the new analyses provide new, important insight into classifying methods. Conversely, if the analyses do not improve Timna's effectiveness, then the new analyses are returning results

that are similar to previous analyses' results, or they are returning results that have no correlation with a method's candidacy.

Researchers can also simply include all known analyses in Timna and observe which analyses appear in the rules that Timna generates after training. The analyses that Timna uses in these rules are the most important analyses, because Timna learns from the training data which analyses are most relevant to its classification.

7. RELATED WORK

Researchers have investigated several individual mining analyses. These analyses are largely focused on the identification of seeds. Once a miner finds seeds, and has a high degree of confidence in those seeds, the exploratory tools can be used to expand those seeds into a full concern. Of course, the higher the recall of an analysis (the more seeds it finds out of the total possible seeds), the less work that the miner must do using exploratory tools.

Two groups have investigated the use of code clone detection tools for the discovery of seeds. Shepherd et al. [11] used PDG-based clone detection to discover seeds with very high precision. Magiel and van Deursen compared token-based and AST-based clone detection techniques for seed discovery, finding no clear winner between these methods, but confirming that cross-cutting functionality is often implemented using code clones [4].

Silvia et al. [3] performed several experiments to use program traces to identify seeds. They search for specific patterns in a trace, identifying these methods as seeds. This technique appears very promising; we hope to integrate it into our framework soon. Tonella et al. [12] used formal concept analysis to analyze program traces. They examined the generated concept lattice and used it to assist in making a classification.

The call graph fan-in analysis by Marius et al. [8] is a promising analysis. They produced reasonably precise results with their automated tool, which they then refined with a manual filtering. They provided a thorough discussion of the candidates that they found.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we have shown how Timna facilitates quick prototyping of analyses, and the evaluation of these analyses. In the case study, we showed that Base+IsVoid achieves a significant improvement in effectiveness over Base. If an analysis achieves similar improvement in Timna's effectiveness in practice, that analysis should be added to Timna's canonical set of analyses.

In the future, we hope to improve the implementation of Timna, and perform experiments using combinations of existing aspect mining analyses in order to determine which analyses we should include in a canonical set. By combining the results of all of the best analyses, we hope to achieve performance that is greater than any individual analysis. We also plan to improve Timna's integration into the Eclipse framework, and the integration of Timna's phases.

9. REFERENCES

[1] *Eclipse's JDT*. http://www.eclipse.org/jdt/, October 2004.

[2] *Weka Data Mining Software*. http://www.cs.waikato.ac.nz/ml/weka/, October 2004.

[3] S. Breu and J. Krinke. Aspect mining using event traces. In *Auto. Soft. Eng. Conf.*, 2004.

[4] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwe. An evaluation of clone detection techniques for identifying cross-cutting concerns. In *Int. Conf. on Soft. Maintenence*, 2004.

[5] R. Ettinger and M. Verbaere. Untangling: a slice extraction refactoring. In *Proceedings of the 3rd International Conference on Aspect-oriented Software Development*, pages 93–101. ACM Press, 2004.

[6] W. G. Griswold, Y. Kato, and J. J. Yuan. Aspect browser: Tool support for managing dispersed aspects. In *Workshop on Multi-Dimensional Separation of Concerns*, 2000.

[7] J. Hannemann and G. Kiczales. Overcoming the prevalent decomposition of legacy code. In *Wkshp on Advanced Separation of Concerns*, 2001.

[8] M. Marin, A. van Deursen, and L. Moonen. Identifying aspects using fan-in analysis. In *Working Conf. on Reverse Eng.*, 2004.

[9] M. P. Robillard and G. C. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *International Conference on Software Engineering*, 2002.

[10] D. Shepherd, J. Palm, L. Pollock, and M. Chu-Carroll. Timna: A framework for combining aspect mining analyses. In *Submitted to the International Conference of Aspect Oriented Software Development*, 2005.

[11] D. Shepherd, L. Pollock, and E. Gibson. Design and evaluation of an automated aspect mining tool. In *Int. Conf. on Soft. Eng. Research and Practice*, 2004.

[12] P. Tonella and M. Ceccato. Aspect mining through the formal concept analysis of execution traces. In *Working Conf. on Reverse Eng.*, 2004.

[13] K. D. Volder and D. Janzen. Navigating and querying code without getting lost. In *Aspect Oriented Software Development*, 2003.

Aspect Mining using Clone Class Metrics

Magiel Bruntink
Centrum voor Wiskunde en Informatica
P.O. Box 94079, 1090 GB Amsterdam
The Netherlands
Magiel.Bruntink@cwi.nl

Abstract

This paper outlines how clone detection results can be filtered such that useful aspect candidates remain. In particular, our goal is to identify aspect candidates that are interesting for the purpose of improving maintainability. To reach this goal, clone class metrics are defined that measure known maintainability problems such as code duplication and code scattering. Subsequently, these clone class metrics are combined into a grading scheme designed to identify interesting clone classes for the purpose of improving maintainability using aspects.

1. Introduction

Large-scale industrial software applications are inherently complex, and thus a good separation of concerns within such applications is indispensable. Unfortunately, recent insight reveals that current means for separation of concerns, i.e. functional decomposition or object-oriented programming, are not sufficient [17]. No matter how well large applications are decomposed using current means, some functionality, typically called *crosscutting concerns*, will not fit the chosen decomposition. As a result, implementations of such crosscutting concerns will be *scattered* across the entire system, and become *tangled* with other code. Obviously, the consequences for maintenance of the system, and its future evolution, are dire.

Aspect-oriented software development (AOSD) has been proposed as an improved means for separation of concerns. Aspect-oriented programming languages add an abstraction mechanism (called an *aspect*) to existing (object-oriented) programming languages. This mechanism allows a developer to capture crosscutting concerns in a modular way. In order to use this new feature, and make the code more maintainable, existing applications written in ordinary programming languages should be transformed into aspect-oriented applications. To that end, (scattered and tangled) code implementing crosscutting concerns should be identified, and subsequently be refactored into aspects.

Identifying crosscutting concerns is an important part of a process referred to as *aspect mining*. One of the goals of aspect mining is to identify opportunities for transforming (parts of) the code of an application into aspect-oriented code. Since aspects[1] are specifically designed to deal with crosscutting concerns, aspect mining is naturally focused on crosscutting concerns. In previous work we demonstrated that two *clone detection* techniques can be used to identify code fragments that belong to relevant crosscutting concerns [4]. Given these encouraging results, we are now challenged to apply these clone detection techniques in the field of aspect mining. In other words, how can we use clone detection results to find good candidate aspects? In particular, can we identify aspects which can be applied such that the maintainability of the system is improved?

In this paper we will discuss a possible approach to the application of clone detection to aspect mining. The basic idea is to develop a method to filter the output of a clone detector, the so-called *clone classes* [10]. Based on observations made during an earlier case study [4], we propose a number of *clone class metrics*. These metrics are used to attach a 'grade' to a clone class, which indicates how relevant the clone class is for the aspect mining process. Clone classes which score below a threshold value can then be filtered out, and hopefully only relevant classes remain. Furthermore, these metrics reflect our expectations of aspect mining: to find aspects which can improve the maintainability of the system by reducing the amount of scattering and code duplication.

In Sections 2 and 3 we give a short overview of existing aspect mining and clone detection techniques, respectively. Section 4 describes the system that we used for our case study, and the crosscutting concerns we considered. Additionally, we mention some of our earlier results from [4]. The clone class metrics are described in Section 5, together with some initial results and discussion.

[1] Different flavours of aspects exists within the aspect-oriented paradigm, and therefore aspect mining could target any of them.

2. Aspect Mining

Aspect mining is typically described as a specialised reverse engineering process, which is to say that legacy systems (source code) are investigated (mined) in order to discover which parts of the system can be represented using aspects. This knowledge can be used for several goals, including re-engineering and program understanding. Several tools are in existence that may help automate this process [8, 7, 15, 20]. Aspect mining techniques vary mainly in the kind of information they extract from a legacy system. Marin et. al. calculate the fan-in metric for the methods in a system [13]. Shepherd et. al. perform PDG-based clone detection [16]. Breu and Krinke generate execution traces and identify recurring execution relations [3]. Tourwe and Mens group identifiers using concept analysis [19]. Tonella and Ceccato generate program traces using use cases, and employ concept analysis to discover concepts and computational units that are implemented in multiple modules which contribute to multiple use cases [18].

3. Clone Detection

Clone detection techniques attempt at finding duplicated code, which may have undergone minor changes afterward. The typical motivation for clone detection is to factor out copy-paste-adapt code, and replace it by a single procedure.

Several clone detection techniques have been described and implemented:

- **Text-based** techniques [9, 6] perform little or no transformation to the 'raw' source code before attempting to detect identical or similar (sequences of) lines of code. Typically, white space and comments are ignored.
- **Token-based** techniques [10, 1] apply a lexical analysis (tokenization) to the source code, and subsequently use the tokens as a basis for clone detection.
- **AST-based** techniques [2] use parsers to first obtain a syntactical representation of the source code, typically an abstract syntax tree (AST). The clone detection algorithms then search for similar subtrees in this AST.
- **PDG-based** approaches [11, 12] go one step further in obtaining a source code representation of high abstraction. Program dependence graphs (PDGs) contain information of semantical nature, such as control- and data flow of the program.
- **Metrics-based** techniques [14] are related to hashing algorithms. For each fragment of a program the values of a number of metrics is calculated, which are subsequently used to find similar fragments.

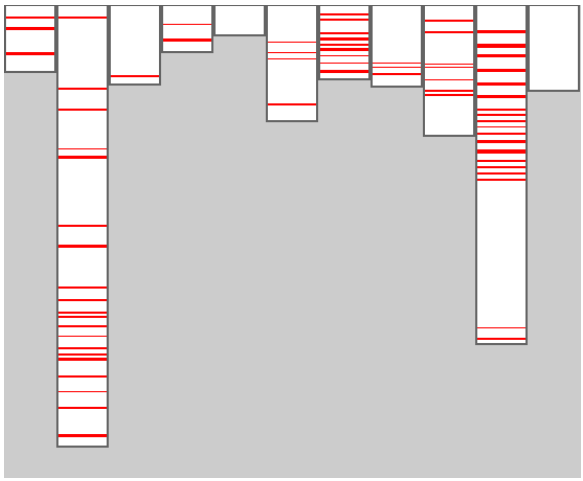


Figure 1. Scattering of the parameter checking concern.

Concern	LOC	Fraction
Error handling	1716	9%
Dynamic execution tracing	1539	8%
Function parameter checking	1441	7%
Memory allocation handling	1110	6%
Total	5806	31%

Table 1. Code percentages devoted to various concerns, in a 20 KLOC component.

Following Walenstein [21], clone detection adequacy depends on application and purpose. Finding crosscutting concerns is a completely new application area, potentially requiring specialized types of clone detection.

4. Case Study

4.1. Background

Our paper is based on a software component (called *CC*) of 20,000 lines of C code, part of the larger code base (comprising over 10 million lines of code) of ASML, the world market leader in lithography systems based in Veldhoven, The Netherlands. The CC component is responsible for the conversion of data between several data structures and other utilities used by communicating components.

Developers working on this component express the feeling that a disproportional amount of effort is spent implementing 'boiler plate' code, i.e., code that is not directly related to the functionality the component is supposed to implement. Instead, much of their time is spent dealing with concerns like error handling and parameter checking (explained below).

This problem is not limited to just the component we selected; it appears in nearly the entire code base. Since the developers at ASML use an idiomatic approach to implement these crosscutting concerns in all applicable modules, similar pieces of code are scattered throughout the system. Clearly, large benefits in code size, quality and comprehensibility are to be expected if such concerns could be handled in a more systematic and controlled way.

4.2. Crosscutting Concerns

A domain expert manually marked places in the CC component dealing with four different crosscutting concerns. Each line in the application was annotated with at most one mark, and as a result, each line belongs to at most one of the concerns described below, or to no concern.

- **Error handling.** General error handling and administration; this code is responsible for roughly three tasks: the initialisation of variables that will hold return values of function calls, the conditional execution of code depending on the occurrence of errors and finally administration of error occurrences in a data structure.
- **Tracing.** Dynamic execution tracing; logging the values of input and output parameters of C functions to facilitate debugging.
- **Parameter checking.** Responsible for two requirements: (1) making sure that parameters of type pointer are checked against null values before they are dereferenced, and (2) checking whether parameter values are within allowable ranges.
- **Memory error handling.** Dedicated handling of errors originating from C memory management.

All together, these concerns comprise roughly 31% of the code. The details are shown in Table 1, while Figure 1 illustrates the scattered nature of these concerns by highlighting the code fragments belonging to the parameter checking concern. The vertical bars represent the files of the 20 KLOC component, and within each vertical bar, horizontal lines of pixels correspond to lines of source code within the file. Coloured lines are part of the memory error handling concern. The other concerns exhibit a similarly scattered distribution.

4.3. Previous Results

In [4] we demonstrated to what extent two clone detection techniques (AST-based and token-based) can be used to identify crosscutting concern code. The experiment compared lines of code belonging to the concerns described above to the output of two clone detectors, Bauhaus' ccdiml (AST-based) and CCFinder (token-based).

The first step of the experiment consisted of obtaining so-called *clone classes* from the clone detectors. A clone class is a set of code fragments that are duplicated (or cloned) according to a clone detector. Subsequently, for each clone class it was determined how many lines of each concern are *covered* by the clone class. A line of a concern, i.e. one of the four concerns described above, is covered by a clone class if the line occurs in one of code fragments of the clone class.

The final evaluation of the clone detection techniques consisted of finding the number of clone classes required to reach an acceptable level of coverage (80%) for each concern. Complementary to evaluating the clone detectors based on coverage, the experiment also considered the resulting *precision*. The ideal case is a clone class that includes nothing but lines of code belonging to one of the concerns described above. However, as the results in [4] have shown, many clone classes also include other lines of code. Clone classes that have a high ratio of other lines compared to lines belonging to a concern, are evidence that a clone detector is not a suitable tool to identify the code of that particular concern.

Code belonging to either the parameter checking or memory error handling concerns tends to be covered well by both clone detection techniques, while tracing and error handling code is not. Furthermore, the results showed that clone classes which have good coverage of one of the concerns, tend to have a higher precision in case of the AST-based technique than in case of the token-based technique.

5. Approach

5.1. Goals

The main goal of mining for aspects (and subsequent re-engineering) in the *CC* component –and the entire ASML source base– is improving its maintainability. In other words, the mining process should point out opportunities for re-engineering using aspects such that the maintainability of the component/system can be improved. It is out of the scope of this paper to detail how to validate the actual maintainability improvement offered by the aspects that are found. However, this is an important issue that will require future attention by the aspect mining community. Aspect mining techniques should be designed for specific purposes, and careful validation is needed to justify the use of aspect mining techniques instead of more traditional techniques (like re-engineering using objects or procedures). Furthermore, comparison of aspect mining techniques requires that the purposes of these techniques are specified and compatible.

A maintainability issue with the current *CC* component (and the entire ASML code base) is duplication of code belonging to known crosscutting concerns. This issue was explored in [4], and summarised in Section 4. The validation

of the use of aspects to improve the maintainability of the parameter checking concern is work in progress [5]. Since results of this validation have been promising, aspect mining for the purpose of improving maintainability is (at least) required to identify the parameter checking concern as an opportunity for aspect use. Additionally, the aspect mining technique should suggest aspects for concerns that are similar in nature.

5.2. Clone Class Metrics

To reach the goals outlined above, we propose to employ clone detection in combination with a set of metrics to filter the resulting clone classes. The use of clone class metrics in order to filter clone detection results was previously suggested and implemented by Kamiya et. al. [10]. However, their work does not focus on the mining of aspects.

In order to find aspects that could improve maintainability, the metrics should be designed such that they capture maintainability problems (of the ASML source base). Duplication of code is a well-known cause for maintainability problems, which justifies the use of clone detection techniques. Clone class metrics that capture the severity of code duplication are thus interesting for the purpose of maintainability improvement. The following metrics capture the severity of code duplication for a clone class C, such that higher values correspond to more severe cases of code duplication:

- **Number of Clones (NC).** The number of clones that are included in C. Equivalent to the POP(C) metric defined by Kamiya et. al. [10].
- **Number of Lines (NL).** The number of (distinct) lines of code (SLOC, non-comment/white space) in C.
- **Average Clone Size (ACS).** The number of lines (**NL**) divided by the number of clones (**NC**).

Many instances of code duplication can be resolved by means of traditional re-engineering techniques, like replacing clones with calls to a procedure which factors out the duplicated code. Therefore, metrics are required that differentiate between the "simple" cases of code duplication, i.e. those that can be fixed by traditional means, and those cases that require aspects. The parameter checking concern is known to benefit from the use of aspects, due to its scattered nature [5]. In particular, the parameter checking concern implements the requirement that *every public function* has to make sure that parameters of type pointer are checked against null before they are dereferenced. It turns out that a high percentage of the scattered implementation is covered by a small number of clone classes [4]. These clone classes therefore contain clones from many different modules of the system. The following two metrics capture the notion of scattering for a clone class, such that higher values correspond to higher amounts of scattering:

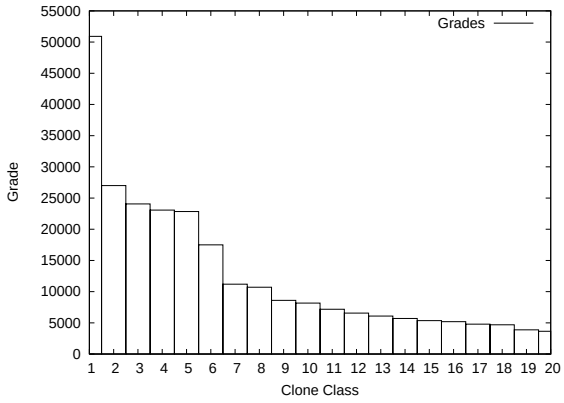


Figure 2. First 20 highest graded clone classes.

- **Number of Files (NFI).** The number of distinct files in which the clones of C occur.
- **Number of Functions (NFU).** The number of distinct functions in which the clones of C occur.

The set of metrics defined above is not intended to be complete or minimal. A clone class also needs to be evaluated with respect to the constructibility of an aspect for that clone class. It is also required to consider the system-wide implications of re-engineering a clone class using aspects. With regard to minimality, some of the metrics defined above may measure the same factors (for example, $\mathbf{NL}(C) = \mathbf{ACS}(C) \cdot \mathbf{NC}(C)$), and thus some may turn out to be redundant.

5.3. Grading

Using the clone class metrics a 'grade' can be attached to each clone class. For our purpose, such a grade should give an indication of the maintainability improvement obtained if the clone class is re-engineered using aspects. Clearly a large number of grading schemes is possible, even given the limited set of metrics defined here. This paper will focus only on the following simple grading scheme:

$$\text{Grade}(C) = \mathbf{NL}(C) \cdot \mathbf{NFU}(C)$$

Consequently, clone classes which are both big (**NL**) and scattered (**NFU**) will be assigned high grades. For purposes other than maintainability improvement, different grading schemes or metrics may be more applicable. The use of different metrics and grading schemes is the subject of further research.

5.4. Initial Results

The component described in Section 4 was used to generate some initial results of the aspect mining process using

the grading scheme defined above. First, clone classes were calculated using the AST-based clone detector (ccdiml) of the Bauhaus toolkit[2]. The minimum clone length was set to 2 lines, leaving all other settings at their defaults. 756 clone classes were found by the clone detector. Second, for each clone class, the clone class metrics and the resulting grades were calculated. Figure 2 shows the grades of the first 20 highest graded clone classes, ranked according to their grades. Observe that a small number of clone classes has high grades, while the remaining grades are near the average. For the purpose of maintainability improvement this is a desirable property, since large improvements (as defined by the grading scheme) can be obtained by using aspects for a small number of clone classes.

The clone class with the highest grade consists of a large number (265) of very small clones (1.43 average size), that are scattered across 134 functions. A number of clones from this class contains code belonging to the error handling concern, in particular the code responsible for initialising the variables used for error administration. However, a large number of other clones from this class contain similar pieces of initialisation code which are not related to error handling. For this reason, the use of an aspect for this clone class is probably not desirable. Class 17 is an anomalous result in the sense that many of its clone are overlapping. It consists of 1252 clones, but the average clone size is only 0.29. This particular class does not cover any known concerns.

Clones classes 2, 3, 6, 9, 10-12 and 19 cover large parts of the parameter checking concern. This result is as expected, since the clone class metrics and the grading scheme were designed with this particular concern in mind. Concerns which are known to be similar, in particular the tracing and memory error handling concerns, are represented in the top 20 as well. Clone classes 4, 5, 7, 8 and 14 cover parts of the tracing concern, while the remaining classes 13, 15, 16, 18, 20 cover parts of memory error handling. Thus, except for classes 1 and 17, the 20 highest graded classes all cover parts of the four known crosscutting concerns. As was shown in [4], most of these clone classes also contain varying amounts of noise, i.e. lines of code that do not belong to any known concerns.

6. Conclusions

This paper outlined how clone detection results can be filtered such that useful aspect candidates remain. For the purpose of improving the maintainability of a component of a large C code base, a number of clone class metrics was defined that capture the severity of code duplication and scattering of a clone class. Subsequently, these metrics were combined into a grading scheme that allows interesting clone classes to be pointed out. It was shown that the approach can point out a concern which is known to have a beneficial implementation using aspects. Additionally, concerns of similar scattered and duplicated nature are also identified. Future work lies in the extension of the clone class metrics and refinement of the grading scheme. Other open issues include the constructibility of aspects for a given clone class, and measurement of the impact of aspect use at the system level.

Acknowledgements Partial support was received from SENTER (CWI, project IDEALS, hosted by the Embedded Systems Institute).

[2] URL: `http://www.bauhaus-stuttgart.de/`

References

[1] Brenda S. Baker. On finding duplication and near-duplication in large software systems. In *Second Working Conference on Reverse Engineering (WCRE'95)*, pages 86–95, Los Alamitos, California, 1995. IEEE Computer Society Press.

[2] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance (ICSM'98)*, pages 368–377. IEEE Computer Society Press, 1998.

[3] Silvia Breu and Jens Krinke. Aspect mining using dynamic analysis. In *GI-Softwaretechnik-Trends, Mitteilungen der Gesellschaft für Informatik*, volume 23, pages 21–22, Bad Honnef, Germany, May 2003.

[4] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwé. An evaluation of clone detection techniques for identifying crosscutting concerns. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*. IEEE Computer Society Press, 2004.

[5] M. Bruntink, A. van Deursen, and T. Tourwé. Isolating crosscutting concerns in embedded C code, 2004. Submitted for publication.

[6] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of the International Conference on Software Maintenance (ICSM'99)*, pages 109–118, September 1999.

[7] W. G. Griswold, Y. Kato, and J. J. Yuan. Aspectbrowser: Tool support for managing dispersed aspects. Technical Report CS1999-0640, 3, 2000.

[8] Jan Hannemann and Gregor Kiczales. Overcoming the prevalent decomposition in legacy code. In *Proceedings of the ICSE Workshop on Advanced Separation of Concerns*, Toronto, Canada, May 2001.

[9] J.H. Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of the IBM Centre for Advanced Studies Conference*, pages 171–183, 1993.

[10] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):645–670, July 2002.

[11] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis*, pages 40–56. Springer-Verlag, 2001.

[12] Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the Eight Working Conference On Reverse Engineering (WCRE'01)*, pages 301–109. IEEE Computer Society Press, October 2001.

[13] M. Marin, A. van Deursen, and L. Moonen. Identifying aspects using fan-in analysis. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society Press, November 2004.

[14] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of International Conference on Software Maintenance*, pages 244–254. IEEE Computer Society Press, November 1996.

[15] M.P. Robillard and G.C. Murphy. Concern graphs: Finding and describing concerns. In *Proc. Int. Conf. on Software Engineering (ICSE)*. IEEE, 2002.

[16] David Sheperd, Emily Gibson, and Lori Pollock. Automated mining of desirable aspects. Technical Report 4, Department of Computer and Information Sciences, University of Delaware, Newark, DE 19716, 2004.

[17] Peri Tarr, Harold Ossher, William Harrison, and Jr. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software engineering*, pages 107–119. IEEE Computer Society Press, 1999.

[18] P. Tonella and M. Ceccato. Aspect mining through the formal concept analysis of execution traces. Technical report, IRST, May 2004.

[19] T. Tourwé and K. Mens. Mining aspectual views using formal concept analysis. In *Proceedings of the Source Code Analysis and Manipulation (SCAM) Workshop*. IEEE Computer Society Press, September 2004.

[20] K. De Volder. The jquery tool: A generic query-based code browser for eclipse. Presentation at Eclipse BoF at OOPSLA 2002, 2002.

[21] A. Walenstein. Problems creating task-relevant clone detection reference data. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'03)*, pages 285–294. IEEE Computer Society Press, 2003.

Refactoring JHOTDRAW's Undo concern to ASPECTJ

Marius Marin

Software Evolution Research Lab
Delft University of Technology
The Netherlands
A.M.Marin@ewi.tudelft.nl

Abstract

In this paper we discuss an approach to the aspect-oriented refactoring of the Undo *concern in an open-source Java system. A number of challenges and considerations of the proposed solution are analyzed for providing useful feedback about how the employed aspect language could better support the refactoring to aspects. We also consider the* unpluggability *property of a concern as an estimate of its refactoring costs and propose a number of research questions to measure the improvements due to aspect refactoring.*

1. Introduction

Aspect oriented programming(AOP) is aimed at overcoming the modularization limitations of object orientation, and in particular at reducing code tangling and scattering. Refactoring is a technique for improving the internal structure of the code without affecting its external behavior [1]. Refactoring object oriented systems to aspects is a natural step towards AOP adoption. However, it is important to see how the existing approaches to AOP can support the refactoring process. Starting from this consideration, we propose an aspect solution to the *Undo* crosscutting concern in an open-source Java system, using ASPECTJ [1] as the implementation language. The analyzed system is JHOTDRAW [2], a model framework for two-dimensional graphics of around 18,000 non-comment lines of code.

The case for the aspect refactoring of the *Undo* concern in JHOTDRAW was introduced in our previous work [2], where *fan-in* analysis was employed to identify crosscutting concerns. The results have shown about 30 undo activities defined for various elements of the graphical framework. A classification of these elements would comprise *command*, *tool*, and *handle* classes as well as one class for dragging figures. We will discuss the refactoring of the *commands* group as the largest in terms of defined undo activities and also as a very common (undoable) task in a drawing application.

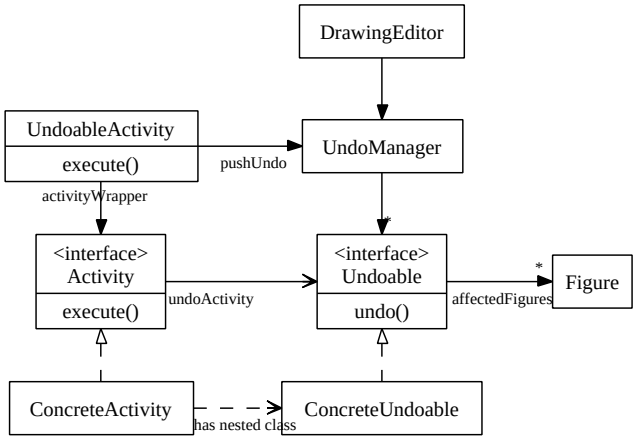


Figure 1. Participants for *undo* in JHOTDRAW.

2. Current Undo implementation

A number of activities in JHOTDRAW, such as handling font sizes and colors, image rotation, or inserting the clipboard's content into a drawing, support the undo functionality. A representation of the elements in the implementation of the undo concern is given in figure 1.

The *Activity* components participate in the implementation of the *Command* design pattern. The pattern provides a generic interface (*Activity*) for the operations to be executed when menu items are selected by the user, which allows to separate the user-interface from the model. Item-selection actions result in invocations to the *execute* method of the associated, specific activities. Many of these activities also have support for undo functionality, which in JHOTDRAW is implemented by means of nested (undo) classes. The nested class knows how to undo the given activity and maintains a list of *affected figures* whose state is also affected if the activity must be undone. Whenever the activity modifies its state, it also updates fields in its associated undo-

[1] www.eclipse.org/aspectj/
[2] jhotdraw.org, v.5.4b1

activity needed to actually perform the undo. The application supports repeated undo operations (the *Undo Command*) by recording the last executed commands in reversed order. This is achieved by wrapping the commands that can be undone into an *Undoable Command* object, which serves three roles: first, it assumes the request to execute the command, second, delegates the command's execution to the wrapped command, and last, acquires a reference to the undo activity associated with the wrapped command and pushes it into a stack managed by an *UndoManager* object. When executing an *Undo Command*, the top undo activity in the stack is extracted and, after the execution of its *undo()* method, is pushed into a redo stack managed by the same *UndoManager* object.

The *Command* hierarchy in JHOTDRAW, shown in figure 5, implements the design pattern bearing the same name. The (12) undo-able commands store a reference to their associated undo activity. These references are obtained in the control flow of the command's execution through dedicated factory methods.

Given the described implementation, it is apparent that the primary decomposition of *Command* is crosscut by a number of elements, as follows:

(1) the field declared by *AbstractCommand* for storing the reference to the associated undo activity,

(2) the accessors for this field implemented by the same class,

(3) the *UndoActivity* nested classes implemented by most of the concrete commands that support undo functionality,

(4) the factory methods for the undo activities declared by each concrete command that can be undone,

(5) the references to the before enumerated elements from non-undo related members, e.g., the *execute()* method of the command class.

These crosscutting elements are outlined in the figures 2 and 6 for two command classes: (1) *ChangeAttribute Command* modifies the predefined attributes of a figure, such as the text color or the font size for a *Text Figure*, and (2) *Paste Command* is an activity that supports the insertion of the clipboard content into the active drawing of the graphics editor. The same elements are also used as criteria for grouping the command classes in figure 5, as it will be described in section 4.

```
public class ChangeAttributeCommand extends AbstractCommand {

    //constructor and private fields ...

    //the command's execute() method
    public void execute() {
        super.execute();

        setUndoActivity(createUndoActivity());
        getUndoActivity().setAffectedFigures(
            view().selection());
        FigureEnumeration fe = getUndoActivity().
            getAffectedFigures();
        while (fe.hasNextFigure()) {
            fe.nextFigure().setAttribute(fAttribute, fValue);
        }

        view().checkDamage();
    }
    // Factory method for undo activity
    protected Undoable createUndoActivity() {
        return new ChangeAttributeCommand.UndoActivity(
            view(), fAttribute, fValue);
    }
    public static class UndoActivity extends UndoableAdapter {
        //implementation of the undo nested functionality...
        public void undo() {...}; // ...
    }
}
```

Figure 2. The original ChangeAttributeCommand class to change a figure's attribute.

```
public void execute() {
    //super.execute(); - added by a separate aspect, not
    //undo-related, with a higher priority than the undo aspect
    FigureEnumeration fe = view().selection();
    while (fe.hasNextFigure()) {
        fe.nextFigure().setAttribute(fAttribute, fValue);
    }

    view().checkDamage();
}
```

Figure 3. The refactored ChangeAttributeCommand.

```
public privileged aspect ChangeAttributeCommandUndoActivity {

    pointcut inChangeAttributeCommand(ChangeAttributeCommand cmd) :
        this(cmd) &&
        execution(void ChangeAttributeCommand.execute());

    before(ChangeAttributeCommand cmd) :
        inChangeAttributeCommand(cmd) {
            cmd.setUndoActivity(cmd.createUndoActivity());
            cmd.getUndoActivity().setAffectedFigures(
                cmd.view().selection());
    }

    Undoable ChangeAttributeCommand.createUndoActivity() {
        return new ChangeAttributeCommandUndoActivity.
            UndoActivity(view(), fAttribute, fValue);
    }

    public static class UndoActivity extends UndoableAdapter {
        // the same implementation as for the original nested class
    }
}
```

Figure 4. The aspect solution for ChangeAttributeCommand.

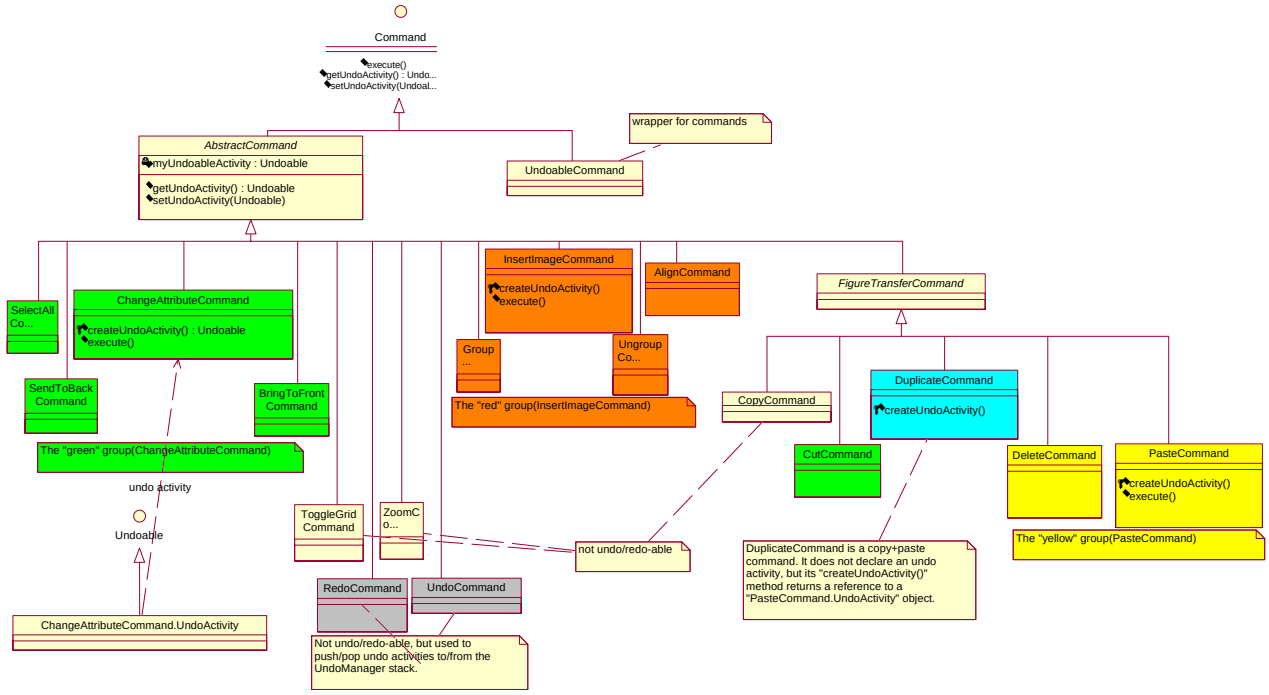


Figure 5. Command hierarchy in JHOTDRAW.

3. Example refactorings

3.1. Tirsen's Generic Undo Aspect

A general solution for handling the undo functionality was proposed by Tirsen [3]. It mainly consists of keeping track of the fields that are set when a command executes. However, the solution suffers from several limitations when considering the *undo* implementation in JHOTDRAW: it does not capture all the state modifications caused by a command's execution, such as changes in data structures, and it requires filtering the set fields, as not all these fields are of interest to the undo process. Given the complexity of the undo process in JHOTDRAW and the way it is handled, the approach is problematic.

3.2. A Simple Case: Undoing the ChangeAttribute Command

The systematic refactoring we propose for the undo functionality consists of several steps. First, an undo-dedicated aspect is associated to each undo-able command. The aspect will implement the entire undo functionality for the given command, while the undo code is removed from the command class. By convention, each aspect will consistently be named by appending "UndoActivity" to the name of its associated command class to enforce the relation between the two, as in figure 4. In a successive step, the command's nested *UndoActivity* class moves to the aspect. The factory methods for the undo activities (*createUndoActivity()*) also move to the aspect, from where are introduced back, into the associated command classes, using inter-type declarations. Lastly, the undo setup is attached to those methods from which was previously removed, namely the *execute()* method, by means of an advice. Figure 2 shows the original implementation of the command, while figures 3 and 4 illustrate the refactored class and the aspect solution, respectively.

3.3. A Complex Case: Undoing the Paste Command

The general strategy outlined above for the case of the *ChangeAttribute Command* requires some supplementary steps for commands with a higher degree of tangling for the undo functionality. An interesting case for its complexity is that of the *PasteCommand* class, shown in figure 6. Both the command's main logic and the undo-related setup depend on common condition checks. The proposed solution looks for a clean separation of the two concerns, hence it captures the calls that set the variables checked in the command's execution, and re-uses the same values when executing the undo functionality as a separate, post-command operation. The common conditions, emphasized in figure 6, are also shown in figure 7 and have the associated pointcuts marked in figure 8. The aspect defines its own set of variables that are set to the same values as the ones checked in the control

flow of the advised (*execute*) method.

4. Levels of Unpluggability

The refactoring we propose tries to stay close to the original design of the application and to ensure an easy migration to the aspect-based solution. After identifying the crosscutting concern, this is removed from the system and re-added in an aspect-specific manner, as previously discussed. However, the concern's removal has different levels of complexity for various commands. Given the identified elements of the crosscutting concern, it is possible to distinguish common characteristics for grouping the commands as a complexity assessment. The classification shown as colors in figure 5 is based on two main criteria:

1. the degree of tangling of the undo setup in the command's logic, particularly the activity's *execute()* method, and
2. the impact of removing the undo-related part from its original site, which can be estimated by the number of references to the factory method and to the methods of the nested undo activity.

These characteristics define the concern as *unpluggable*; that is, the core logic of the method executing the command is separated in the method's flow from the crosscutting undo elements, thus making possible to have the command executing correctly after removing the lines of code appertained to undo.

The "green"(ChangeAttributeCommand) group exhibits a number of properties that permit a clean feature extraction:

- the references to the nested undo activities and the factory methods for these activities are exclusively from inside the enclosing class, or from other (extending) undo activities,
- the undo-related code in the enclosing classes is unpluggable as previously described, and thus suitable for extraction and refactoring by means of advice constructs [3]
- the other methods related to the crosscutting functionality of undo (*set/getUndoActivity*) are inherited from top level classes (and not overridden locally) where they can be refactored by means of introduction.

The "red" group (*InsertImageCommand*) of commands does not exhibit the undo unpluggability. The commands

[3]In practice, small local refactorings that eliminate one layer of indirection are needed before having the concern's statements separated from the rest of the code. In figures 2 and 3, for instance, the enumeration of the selected figures in the view, *fe*, is obtained differently.

```
//The class extends AbstractCommand that implements
//the accessors for the associated UndoActivity
public class PasteCommand
                extends FigureTransferCommand {
    // ...
    public void execute() {
        super.execute();

        Point lastClick = view().lastClick();
        FigureSelection selection =
            (FigureSelection)Clipboard.getClipboard().
                getContents();
        if (selection != null) {
            setUndoActivity(createUndoActivity());
            getUndoActivity().setAffectedFigures(
                (FigureEnumerator)selection.getData(
                    StandardFigureSelection.TYPE));

            if (!getUndoActivity().getAffectedFigures().
                hasNextFigure()) {
                    setUndoActivity(null);
                    return;
            }
            Rectangle r = getBounds(getUndoActivity().
                getAffectedFigures());
            view().clearSelection();

            // get an enumeration of inserted figures
            FigureEnumeration fe = insertFigures(
                getUndoActivity().getAffectedFigures(),
                lastClick.x-r.x, lastClick.y-r.y);
            getUndoActivity().setAffectedFigures(fe);

            view().checkDamage();
        }
    }

    // Factory method for undo activity
    protected Undoable createUndoActivity() {
        return new PasteCommand.UndoActivity(view());
    }

    public static class UndoActivity
        extends UndoableAdapter {
            //implementation of the nested class ...
    }
}
```

Figure 6. The original PasteCommand class - command to insert clipboard's content into the drawing.

can not yield the expected results in the absence of the functionality defined by the nested undo-related classes. This dependency has been considered a candidate for a preliminary (object-oriented) refactoring with more implications for the original code, but able to produce the concerns' uncoupling.

The refactoring of the "yellow" group (PasteCommand) affects a larger number of classes. The multiple references from outside the class enclosing the *UndoActivity* to the corresponding factory method or to the undo constructor are specific to this group. Moreover, the undo-related calls from the various methods can be more tangled than for the "green" group.

```
public void execute() {
    super.execute();
    Point lastClick = view().lastClick();
    FigureSelection selection = (FigureSelection)
        Clipboard.getClipboard().getContents();
    if (selection != null) {
        //introduced variable for affected figures
        FigureEnumerator figEnum = (FigureEnumerator)
            selection.getData(StandardFigureSelection.TYPE);
        if (!figEnum.hasNextFigure())
            return;
        Rectangle r = getBounds(figEnum);
        view().clearSelection();
        figEnum.reset();
        //the 'fe' enumeration is not needed here anymore
        insertFigures(figEnum, lastClick.x-r.x,
                               lastClick.y-r.y);
        view().checkDamage();
    }
}
```

Figure 7. The refactored execute() method in PasteCommand.

5. Improved Language Support for Refactoring to Aspects

We generally appreciate the results of the refactoring process as leading to a cleaner separation of concerns and to a better modularization. By aspect-refactoring, the two concerns are separately modularized and the secondary concern of undo is no longer tangled into the implementation of the primary one. Our systematic approach is intended to ensure a gradual and possible automatic process of migration, with some of the steps turned into general refactorings, as for instance, migrating nested classes to aspects or extracting features into inter-type declarations. However, a number of drawbacks that, we think, can be overcome by a better aspect language support, can be discussed in relation to this experiment.

The original design uses static nested classes to enforce a syntactical relation between the undo activity and its enclosing command class. Since the ASPECTJ mechanisms do not allow introduction of nested classes, the post-refactoring association will only be an indirect one, based on naming conventions. This is a weaker connection than the one provided by the original solution.

Another drawback is the change of the visibility for the methods introduced from aspects, i.e. inter-type declarations. The visibility declared in the aspect refers to the aspect and not to the target class. For instance, it is not possible in ASPECTJ to introduce members into a class that are *protected* for that class. This is the case for the undo factory methods whose visibility cannot be preserved by the refactoring process. Having caller methods unable to access the callee after refactoring will require changes in the visibility that can weaken the boundaries imposed by the original design.

For the discussed case of the "yellow" group, code in

```
public aspect PasteCommandUndoActivity {
    //store the Clipboard's contents - common condition
    FigureSelection selection;

    pointcut execute_callClipboardgetContents() :
        call(Object Clipboard.getContents())
        && withincode(void PasteCommand.execute());

    after() returning(Object select) :
        execute_callClipboardgetContents() {
            selection = (FigureSelection)select;
    }

    //The variable stores the value returned by insertFigures()
    FigureEnumeration insertedFiguresEnumeration;

    pointcut execute_callinsertFigures() :
        call(FigureEnumeration FigureTransferCommand.
            insertFigures(FigureEnumeration, int, int))
        && withincode(void PasteCommand.execute());

    after() returning(FigureEnumeration figs) :
        execute_callinsertFigures() {
            insertedFiguresEnumeration = figs;
    }

    FigureEnumerator selectedData;

    pointcut execute_callselectiongetData() :
        call(Object FigureSelection.getData(String))
        && withincode(void PasteCommand.execute());

    after() returning(FigureEnumeration dataSel) :
        execute_callselectiongetData() {
            ArrayList al = new ArrayList();
            while(dataSel.hasNextFigure()) {
                al.add(dataSel.nextFigure());
            }
            dataSel.reset();
            selectedData = new FigureEnumerator((Collection)al);
    }

    pointcut executePasteCommand(PasteCommand cmd) :
        this(cmd) &&
        execution(void PasteCommand.execute());

    /**
     * Execute the undo setup.
     */
    void after(PasteCommand cmd) : executePasteCommand(cmd) {
        //the values for the variables that have to be checked here,
        //e.g., selection, have been captured by means of advices

        // the same condition as in the advised method
        if(selection != null) {
            cmd.setUndoActivity(cmd.createUndoActivity());
            cmd.getUndoActivity().setAffectedFigures(selectedData);
            // the same condition as in the advised method
            if (!cmd.getUndoActivity().getAffectedFigures().
                    hasNextFigure()) {
                cmd.setUndoActivity(null);
                return;
            }
            cmd.getUndoActivity().setAffectedFigures(
                insertedFiguresEnumeration);
        }
    }
    /**
     * Factory method for undo activity - cannot be protected anymore
     */
    Undoable PasteCommand.createUndoActivity() {
        return new PasteCommandUndoActivity.
                    UndoActivity(view());
    }

   //the nested class moves to the aspect
    public static class UndoActivity
        extends UndoableAdapter {
        // the undo actvity nested class
    }
}
```

Figure 8. The undo aspect for PasteCommand.

both the method's logic and the undo setup part is executed if a common condition holds. This means that the same condition will be checked in the advice executing the undo setup functionality and in the advised method, too. While we believe this is not a reason for concern from the design point of view, it can be from the perspective of the compiler work. In the same time, the conditions strengthen the relation between the two concerns, and affect the modular reasoning about the undo concern, which has to be aware of the execution particularities of its associated command.

5.1. Research questions

The downsides of the proposed aspect solution, despite an overall improvement, pose several questions.

> How to measure the code improvements due to refactoring to aspects? Is it possible to define a set of metrics for this?
>
> Would it be possible to use these metrics to compare different aspect solutions? How can these solutions be compared from the perspective of easy migration?
>
> What is a good aspect solution? Could we define a set of good practices in aspect oriented programming?

We think that some of these questions can be answered by improving the support of the aspect language for the refactoring process. Preserving the advantages of the original implementation will prove beneficial in eliminating potential tradeoffs. A set of good AOP practices is an open issue, and just as the language itself, is part of the evolution process of the aspect oriented technique. Reliable solutions to common problems, as the undo functionality one, are also critical to avoid intrusive code due to the language mechanisms. All these are important concerns for building confidence in an AOP adoption for existing systems.

6. Conclusions

The solution achieved by applying the aspect oriented techniques to refactor the *undo* concern in an existing, well-designed object-oriented system shows improvements in terms of modularity and separation of concerns. Yet, the downsides of the aspect-based solution raise questions about how the improvements can be quantified and what are the desired aspect solutions for specific crosscuttings. The *unpluggability* property gives a measure of how clear the concern is distinguished in the original code and is a good estimate of the refactoring costs.

References

[1] Martin Fowler et al. *Refactoring: Improving the Design of Existing Code*. Addison Wesley Professional, 1999.

[2] M. Marin, A. van Deursen, and L. Moonen. Identifying aspects using fan-in analysis. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE2004)*. IEEE Computer Society Press, 2004.

[3] J. Tirsen. Undo in AspectJ. Codehause Jutopia discussion forum, April 25 2004. `blogs.codehause.org`.

Isolating Crosscutting Concerns in System Software

Magiel Bruntink
Centrum voor Wiskunde en Informatica
P.O. Box 94079
1090 GB Amsterdam, NL
Magiel.Bruntink@cwi.nl

Arie van Deursen *
Centrum voor Wiskunde en Informatica
P.O. Box 94079
1090 GB Amsterdam, NL
Arie.van.Deursen@cwi.nl

Tom Tourwé
Centrum voor Wiskunde en Informatica
P.O. Box 94079
1090 GB Amsterdam, NL
Tom.Tourwe@cwi.nl

ABSTRACT

This paper reports upon our experience in automatically migrating the crosscutting concerns of a large-scale software system, written in C, to an aspect-oriented implementation. We zoom in on one particular crosscutting concern, and show how detailed information about it is extracted from the source code, and how this information enables us to characterise this code and define an appropriate aspect automatically. Additionally, we compare the already existing solution to the aspect-oriented solution, and discuss advantages as well as disadvantages of both in terms of selected quality attributes. Our results show that automated migration is feasible, and can lead to significant improvements in source code quality.

1. INTRODUCTION

Aspect-oriented software development (AOSD) [5] aims at improving the modularity of software systems, by capturing crosscutting concerns in a well-modularised way. In order to achieve this, aspect-oriented programming languages add an extra abstraction mechanism, an *aspect*, on top of already existing modularisation mechanisms such as functions, classes and methods.

In the absence of such a mechanism, crosscutting concerns are implemented explicitly using more primitive means, such as naming conventions and coding idioms (an approach we will refer to as the *idioms-based approach* throughout this paper). The primary advantage of such techniques is that they are lightweight, i.e. they do not require special-purpose tools, are easy to use, and allow developers to readily recognise the concerns in the code. The downside however is that these techniques require a lot of discipline, are particularly prone to errors, make concern code evolution extremely time consuming and often lead to code size explosion.

In this paper, we report on an experiment involving a large-scale, embedded software system written in the C programming language, that features a number of typical crosscutting concerns implemented using naming conventions and coding idioms. Our first aim is to investigate how this idioms-based approach can be turned into a full-fledged aspect-oriented approach automatically. In other words, our goal is to provide tool support for identifying the concern in the code, implementing it in the appropriate aspect(s), and removing all its traces from the code. Our second aim is then to evaluate the benefits as well as the penalties of the aspect-oriented approach over the idioms-based approach. We do this by comparing the quality of both approaches in terms of the amount of tangling, scattering and code duplication, the lines of code devoted to the concern and the correctness and consistency of its implementation.

*Also affiliated with Delft University, Software Evolution Research Laboratory (SWERL), Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS), Mekelweg 4, 2628 CD Delft, The Netherlands.

1.1 Approach

Our approach to achieving our goals is to zoom in on one particular crosscutting concern, the *parameter checking* concern. Based on the existing source code and the requirements extracted from the manuals, we implement a *concern verifier* for the parameter checking concern. Its primary task is to reason about the current implementation of the concern in order to "characterise" it: the verifier reports where the code deviates from the standard idioms, which allows developers to correct the code when necessary. Manual inspection may also reveal that a particular deviation is in fact on purpose, in which case it will be marked as *intended*. Additionally, the verifier also recovers the specific locations where particular parameters are checked.

The information recovered by the concern verifier is used by the *aspect extractor* and the *concern eliminator*. The former defines an appropriate aspect for the parameter checking concern. This aspect will add parameter checks to the source code wherever necessary, and will make sure this code is not added for the intended deviations. The latter will remove the parameter checking concern from the original source code.

The aspect extractor outputs the aspect in a special-purpose aspect language. This definition is then translated automatically by our *DSL compiler* to an already-existing, general-purpose aspect language, that can weave the parameter checking concern back into the source code.

Once the correct aspect has been constructed, we can assess the quality of the aspect-oriented solution and compare that to the idioms-based solution.

1.2 Outline

The remainder of the paper is structured as follows. The next section introduces the parameter checking concern, its requirements and the idioms used to implement it. Section 3 discusses the concern verifier, its implementation, and the results of running it on our case study.. Section 4 presents the domain-specific aspect language we implemented for the parameter checking concern, discusses its implementation in terms of an already-existing aspect weaver, and compares this solution to the idioms-based solution. Section 5 then discusses the (conservative) migration of the idioms-based approach to the aspect-oriented approach. Section 6 considers various quality attributes to compare the aspect-oriented solution to the

idioms-based solution. Finally, Section 7 presents our conclusions and future work.

2. CURRENT PARAMETER CHECKING IDIOM

The subject system upon which we perform our experiments is an embedded system developed at ASML, the world market leader in lithography systems. The entire system consists of more than 10 million lines of C code. Our experiment, however, is based on a relatively small, but representative, software component (which we will call the *CC* component in this paper), consisting of about 19.000 lines of code.

Because the C language lacks explicit support for crosscutting concerns, ASML uses an idiomatic approach for implementing such concerns, based on coding idioms. As a consequence, a large amount of the code of each component is "boiler plate" code. A code template is typically reused and adapted slightly to the context.

2.1 Parameter Checking Requirements

The parameter checking concern is responsible for implementing pointer checks for function parameters and raising warnings whenever such a pointer contains a non-expected (`NULL`) value. The requirement for the concern is that each parameter that has type pointer and is defined by a public (i.e. not `static`) function should be checked against `NULL` values. If a `NULL` value occurs, an error variable should be assigned, and an error should be logged in the global log file. Some exceptions to this requirement exist, as a limited number of functions can explicitly deal with `NULL` values, so the corresponding parameters should not be checked.

The implementation of a check depends on the *kind* of parameter. The ASML code distinguishes between three different kinds: *input*, *output* and the special case of *output pointer* parameters. Input parameters are used to pass a value to a function, and can be pointers or values. Output parameters are used to return values from a function, and are represented as pointers to locations that will contain the result value. The actual values returned can be references themselves, giving rise to a double pointer. The latter kind of output parameters are called *output pointer* parameters. Note that the set of output pointer parameters is a subset of the set of output parameters. Since output and output pointer parameters are always of type pointer, they should always be checked, but only input parameters that are passed as pointers should be checked.

2.2 Idioms Used

Parameter checks occur at the beginning of a function and always look as follows:

```
if(queue == (CC_queue *) NULL) {
   r = CC_PARAMETER_ERR;
   CC_LOG(r,0,("%s: Input parameter %s error (NULL)",
          "CC_queue_empty", "queue"));
}
```

where the type cast of course depends on the type of the variable (`queue` in this case). The second line sets the error that should be logged, and the third line reports that error in the global log file. It is not strictly specified which string should be passed to the `CC_LOG` function. Checks for output parameters look exactly the same, except for the string that is logged.

Since output pointer parameters are output parameters as well, they should also be checked for null values. Additionally, one extra check is required to prevent memory leaks. The requirement at ASML is that output pointer parameters may not point to a location that already contains a value, because the function will overwrite the pointer to that value. Since the original value is then never freed, a memory leak could occur. In order to avoid such leaks, the following test is added for each output pointer parameter:

```
if(*item_data != (void *) NULL) {
   r = CC_PARAMETER_ERR;
   CC_LOG(r,0,("%s: Output parameter %s may already "
          "contain data (!NULL). This data will "
          "be overwritten, which may lead to memory "
          "leaks.", "queue_extract", "item_data"));
}
```

The only difference with the previous test lies in the condition of the `if`, that now checks whether the dereferenced parameter already contains some data (`!= NULL`), and in the string that is written to the log file.

3. CONCERN VERIFIER

The concern verifier is an automated tool that reasons about the idioms-based implementation of the parameter checking concern. This section motivates why we need such an automated tool, explains the information that it recovers from the source code, the coding idioms used, as well as the implementation of the algorithm that verifies these idioms, and the results of running this algorithm on our case study.

3.1 Motivation

If we want to transform the idioms-based approach into an aspect-oriented one, we should first "characterise" the implementation. In other words, we should first locate places where parameters checks occur and mandatory parameter checks are missing, and identify parameters that do not need to be checked.

We achieve this characterisation by implementing a *concern verifier* which checks the implementation of the concern with respect to the coding idioms that hold for it. The verifier outputs a list of *locations*, i.e. functions where parameter checks occur, and a list of *deviations*, i.e. locations in the source code that lack a parameter check although it should be present according to the idioms. This latter list is inspected by a domain expert, who identifies the *intended* and *unintended* deviations. The intended deviations indicate exceptional cases (e.g, parameters that are allowed to be `NULL`), whereas unintended deviations indicate parameters for which a check was forgotten and should be implemented. As we will see later on, our concern verifier is able to identify some intended deviations automatically. In those cases, these deviations are not reported, but simply registered as exceptions.

Thus the following important information is recovered from this code:

- the list of intended deviations informs us which parameters form an exception to the rule. As such, this important information becomes explicitly available, whereas it was not before;
- the number of unintended deviations is a measure for the quality of the idioms-based approach. The smaller this number, the better the quality of the implementation. We expect this number to increase linearly with the size of the source code;
- the verifier identifies the specific location in the code where a particular parameter is checked. Remember that the requirement does not specify where the check should occur, as long as it occurs before the parameter is used.

	required	actually checked	deviations detected	unintended deviations	intended deviations
input	57	40	26	17	9
output	143	94	49	49	0
out pntr	45	15	35	30	5
total	245	149	110	96	14

Table 1: Number of top level parameter checks found for the CC component.

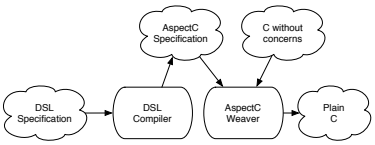


Figure 2: Merging C and DSL code

As we will see in the next sections, this information is vital to our aspect extractor. The aspect it defines should add all necessary parameter checks to the code, but should not insert checks for exceptional parameters. Additionally, it should make sure that the aspect preserves the behaviour of the original idioms-based implementation, which it does by simply implementing the checks at the same locations.

We continue this section by explaining the implementation of the concern verifier in more detail.

3.2 Verifier Implementation

The concern verifier has been developed as a *plugin* in the *CodeSurfer* source code analysis and navigation tool[1]. This tool provides us with programmable access to data structures such as system- and program-dependence graphs, and defines advanced analysis techniques over these structures, such as control- and data-flow analysis and program slicing.

Our verifier needs to consider each public function and verify if the necessary parameter checks occur in it or in the functions it calls. This requires knowledge about the particular kind of a parameter: whether it is input, output or output pointer. Our verifier first extracts this knowledge from the source code by simply checking for assignments to a parameter, looking at *kill* (or *def*) statements for that parameter inside a function's body.

Once the particular kind of a parameter is determined, we can verify whether the necessary checks for it occur in the implementation. If a parameter is not checked, the concern verifier tries to infer if the function is robust for exceptional values, before it registers an unintended deviation. for the parameter At the moment, it uses a simple heuristic: if the function compares the value of a parameter to `NULL` each time before it uses that parameter, we assume it can deal with a `NULL` value. This heuristic does not suffice for identifying all exceptions, however. Distinguishing intended from unintended deviations thus still requires a manual effort. More elaborate heuristics are possible, but are considered future work.

3.3 Verification Results

Applying the verifier to the case at hand yields the data displayed in Figure 1. The CC component implements 157 functions, with 386 parameters in total. 245 of these parameters must be checked, since they are defined by public functions and have pointer type. This is indicated in the first column of Figure 1, which also provides the distribution among the different kinds of parameters. The locations obtained from the verifier tell us which of these 245 parameters are actually checked, as displayed in the second column. It turns out that only 149 (i.e., 60%) of the parameters requiring a check are in fact checked.

The deviations obtained from the verifier aim to help in identifying the remaining 96 parameters that need to be checked. The verifier reports a total of 110 deviations (column 3). Manual inspection of these deviations eliminated 14 intended deviations (for 9 input parameters and 5 output pointer parameters, cfr. column 5).

[1]www.grammatech.com

4. A DOMAIN-SPECIFIC LANGUAGE FOR PARAMETER CHECKING

In order to arrive at a more rigorous treatment of parameter checks (avoiding the situation that as many as 40% of them deviated from the specifications), we propose a domain-specific language (DSL) for representing the kind of parameter checks that are required. In this section we describe the language and corresponding tool support — in the next we explain how existing components can be migrated to this target solution.

4.1 Specification

The idea underlying the language is that a developer annotates a function's signature, by documenting the specific kind of its parameters, i.e. either input or output. Output parameters that are of output pointer kind can also be specified. When a parameter does not require a check, for whatever reason, this can be annotated as well. Additionally, the developer can specify *advice code*, i.e. the code that will perform the actual check. Since this code can differ for the different kinds of parameters, we allow advice code for input, output and output pointer parameters to be specified separately. Although in this paper we do not need it, the DSL also has provisions to express advice code for deviations.

As an example, consider the (partial) specification of the parameter checking aspect for the CC component as depicted in Figure 1. It states that the parameters `CC_queue *queue` and `void **queue_data` of the functions `CC_queue_peek_front` and `CC_queue_peek_back` are output and output pointer parameters, respectively, and that parameter `CC_queue *queue` of function `CC_queue_init` is an output parameter, whereas parameter `void *queue_data` does not need to be checked. Additionally, the advice code implements the required checks for input, output and output pointer parameters. The special-purpose `thisParameter` variable denotes the parameter currently being considered by the aspect, and exposes some context information, such as the name and the type of the parameter and the function defining it. In this respect, it is similar to the `thisJoinPoint` construct in AspectJ. Due to the generality introduced by this variable, we only need to provide three advice definitions in order to cover the implementation of the concern in the complete ASML source code.

4.2 Compilation and Weaving

Rather than implementing our own aspect weaver for the parameter checking DSL, we translate it into an already-existing general-purpose aspect language for the C programming language. As such, we get the benefits of both worlds: we can use a special-purpose, intuitive and concise DSL, for which we don not need to implement a sophisticated weaver ourself. This process is illustrated in Figure 2.

The general-purpose aspect language is a stripped-down variant of the AspectC language [2]. It has only one kind of joinpoint, function exection, and allows us to specify around advice only. Of course, before and after advice can be simulated easily using such around advice. Figure 4 contains an example, which shows how

```
component CC {
     CC_queue_peek_front(output CC_queue *queue, output output_pointer void **queue_data);
     CC_queue_peek_back(output CC_queue *queue, output output_pointer void **queue_data);
     CC_queue_empty(input CC_queue *queue, output bool *empty);
     CC_queue_init(output CC_queue *queue, deviation void *queue_data);
     ...
     input advice {
        if(thisParameter.name == (thisParameter.type) NULL) {
           r = CC_PARAMETER_ERR;
           CC_LOG(r,0,(%s: "Input parameter %s error (NULL)",
                                    thisParameter.function.name, thisParameter.name));
        }
     }
     output advice {
        if(thisParameter.name == (thisParameter.type) NULL) {
           r = CC_PARAMETER_ERR;
           CC_LOG(r,0,(%s: "Output parameter %s error (NULL)",
                                    thisParameter.function.name, thisParameter.name));
        }
     }
     output pointer advice {
        if(*thisParameter.name != (thisParameter.type) NULL) {
           r = CC_PARAMETER_ERR;
           CC_LOG(r,0,(%s: "Output parameter %s may already contain a value. This value will be"
                           "overwritten, which may lead to a memory leak",
                           thisParameter.function.name, thisParameter.name));
        }
     }
}
```

Figure 1: DSL specification of the parameter checking concern

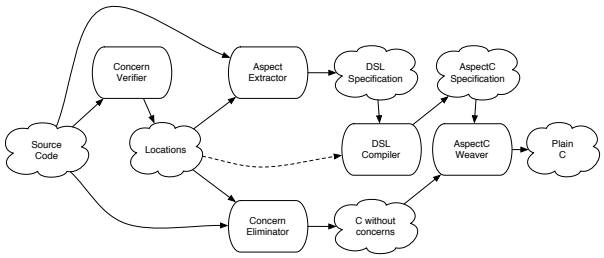


Figure 3: Migrating Existing Components to the DSL

the `advice on` keyword is used to specify advice code for a particular function.

The translation process itself proceeds as follows: the translator considers each parameter of each function in the original DSL specification, looks at its kind(s), retrieves the corresponding advice code, expands that code into the actual check that should be performed, and inserts the expanded code in the function where the parameter is defined. The expansion phase is responsible for assembling and retrieving the necessary context information (i.e. setting up the `thisParameter` variable), and substituting it in the advice code where appropriate. At the end, this advice code will call the original function by calling the special `proceed` function, but only if none of the parameters contain an illegal value (i.e. the error variable is still equal to the `OK` constant). Note that, two checks are implemented for a parameter of output and output pointer kind, since both the output and output pointer advices are substituted for such parameters.

4.3 Application in Case Study

The parameter checking concern in the original CC implementation required 961 lines of C code (see Figure 2). Using the parameter checking DSL, only 133 lines are needed instead: One line for each of the 109 functions that require one their parameters to be checked, $(2 * 7) + 8$ lines for the three different kinds of advice required, and a start and an end line.

	Lines of code
Original C code	961
DSL representation	132
AspectC code	1200

Table 2: Lines of code figures for various parameter checking representations

5. MIGRATION

5.1 Motivation

The steps involved in migration of the idioms-based approach to the DSL approach are depicted in Figure 3. The key steps involved are the extraction of aspect code from the source code, and the elimination of the parameter checking code from the original sources. As we will see, for both steps, the locations obtained by the verifier discussed in Section 3 provides essential information. Moreover, these locations will play a role in the DSL compiler, which can use them in order to regenerate code that is as close as possible to the original code.

5.2 Aspect Extraction

When developing new code, a developer can use the DSL to specify parameter checking aspects, instead of implementing the checks manually. In a migration setting, however, we don't want developers to wade trough millions of lines of already existing source code to annotate function signatures and define an appropriate aspect. Rather, we want to extract such an aspect definition from the existing code automatically. The information required to perform this extraction consists of just (i) the *kind* of each parameter; (ii) whether it requires a check or not; and (iii) if so, the code that needs to be executed for such a check (i.e. the advice code). Apart from

```
int advice on (queue_extract) {
     int r = OK;
     if(queue == (CC_queue *) NULL) {
        r = CC_PARAMETER_ERR;
        CC_LOG(r,0,(%s: " Output parameter %s error (NULL)", "queue_extract", "queue"));
     }
     if(item_data == (void **) NULL) {
        r = CC_PARAMETER_ERR;
        CC_LOG(r,0,(%s:" Output parameter %s error (NULL)", "queue_extract", "item_data"));
     }
     if(item_data != (void **) NULL) {
        r = CC_PARAMETER_ERR;
       CC_LOG(r,0,(%s:"Output paramater %s may already contain data (!NULL). This data will be"
                        "overwritten which may lead to memory leaks", "queue_extract", "item_data"));
     }
     if (r == OK)
        r = proceed();
     return r;
}
```

Figure 4: AspectC specification of the parameter checking concern

this advice code, all this information has already been computed by the concern verifier. Recall from Section 3 that the verifier automatically identifies input, output and output pointer parameters, and that the list of deviations is split into intended and unintended deviations. Our aspect extractor thus merely reuses this information. The advice code, on the other hand, is not considered by our concern verifier. As explained in Section 2, the advice code for input, output and output pointer parameters always consists of an if-test, an assignment and a call to a log function. Our aspect extractor simply constructs this code as the advice code definition.

5.3 Concern Elimination

Besides extracting the aspect specification, the code originally implementing the concern has to be removed from the source code as well. The locations obtained by the verifier indicate where the checks occur, and can be used for these purposes. We currently use a fairly simple solution to eliminate the concern code, based on a prototype implementation in Perl. This is possible because the parameter checking concern is not tangled with the other code, and is easy to recognise and remove. This works well enough for the cases under study at the moment.

5.4 Conservative Translation

The DSL code recovered can be used directly to generate intermediate AspectC code, which then in turn can be woven with the C code from which we eliminated the concern code.

However, when adopting the generated C code in a production environment, we would like to eliminate as many risks as possible. In other words, it is preferable to make the compiler as conservative as possible, trying to stay very close to the original C code. For that reason, the DSL compiler offers the possibility to re-introduce the parameter checks at exactly the same locations as where they were found originally. To that end, it uses information obtained from the verifier (as indicated by the dashed arrow in Figure 3). Naturally, this is only possible for parameters that were already checked correctly, and not for newly introduced checks.

An illustration of the translation of the specification of Figure 1 is given in Figure 4. The example states that the `queue_extract` function should implement two output parameter checks and one output pointer parameter checks. This function is a non-public function, and a specification for it did thus not appear in the DSL specification. The reason it is included in the AspectC specification is that both the `CC_queue_peek_front` and `CC_queue_peek_back` functions

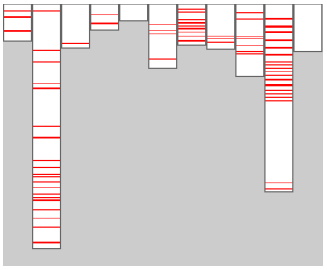


Figure 5: Parameter checking code in the CC component

call the `queue_extract` function, and both parameters of those former functions are checked in the `queue_extract` function in the original code. When translating the specification of the `CC_queue_peek_front` and `CC_queue_peek_back` functions, the translator consults the verifier to see where their parameters are checked, and generates advice code correspondingly.

6. DISCUSSION

In this section we discuss the pros and cons of the DSL approach for the parameter checking concern.

Code Size The aspect-oriented solution reduces the code size of the component by 7%, since the DSL allows us to specify the parameter checking concern in a concise way. The complete aspect definition is specified in only 132 lines, whereas the parameter checking concern in the original component comprised a total of 961 lines.

Naturally, reduced code size alone is an insufficient indicator for increased code quality. However, less code does give the benefits of fewer chances of error, fewer lines to write or understand, and, following Boehm's maintenance cost prediction model [1],, lower maintenance costs.

Scattering and Tangling Figure 5 (generated using the AspectBrowser [4]) shows how the parameter checking concern, implemented using the idioms-based approach, is distributed over the code of the CC component.

The aspect-oriented solution cleanly captures the concern in a modular and centralised way, and thus removes the scattering all together. This does not only make the concern more explicit and tangible in the source code, but also improves its reusability, understandability and maintainability

Apart from system-wide benefits, the adoption of the DSL has consequences for the quality of the parameter checking concern implementation as well.

Unintended Deviations In Section 3 we have seen that as many as 40% of the parameters that ought to be checked are in fact never checked.

It is not immediately clear why so many parameters are left unchecked. One reason is probably that the punishment or reward for the developer is uncertain, and much later in time, happening only when another developer starts using the component in a wrong way that could have been prevented by a proper null pointer warning. Moreover, this figure seems to indicate that developers consider implementing this concern for each parameter too much effort.

Intended Deviations 13% of the reported deviations are intended deviations, i.e. parameters that need not be checked. Although we are presently investigating this issue, we do not see many opportunities to further refine our verifier in order to reduce this figure. These checks are simply "exceptions to the rule" to which the code should adhere. Note however, that it is important to identify these exceptions, because the aspect extractor relies on this information. Moreover, it can improve the understandability of the code. For example, we observed that most intended deviations for output pointer parameters are due to the parameter being used as a *cursor* when iterating over a composite data structure. Since the parameter points to an item in the list, it doesn't matter that it's value is overwritten, and hence, no output pointer check is needed.

Uniform Parameter Checking The advice code specifies how a parameter should be checked, and this code is specified only once and reused afterwards. Consequently, all parameters are checked and logged in the same way. This was not the case for the idioms-based implementation, where the logged strings often differ, or checks are implemented in slightly different ways. For example, all functions except one implement the checks according to the format explained in Section 2. When logging a possible error, 7 different strings are logged for an input parameter error, 4 different strings for an output parameter error and 4 for an output pointer parameter error.

The uniformity of the log file is important for automated tools that reason about the logged errors in order to identify and correct the primary cause of a particular error.

Documentation One of the benefits of using a declarative DSL, is that it can be used for additional purposes than compilation to C [3]. In particular, the parameter checking aspect acts as documentation of the component's functions, or it can be used as input to a documentation generator. In the current implementation of the component, the kind of the parameter is documented inside comments. These comments are often not consistent with the source code however, and are sometimes outdated (e.g. a function defines new parameters that are not document, or vice versa). Moreover, such documentation does not include information about the exceptional parameters that do not need to be checked. The aspect however, makes all this information explicit, and thus improves the understandability of the concern. Additionally, since the aspect is extracted from the source code automatically, it is up to date, and as already explained, we believe it will remain so because developers profit from it.

Scalability Although our tools and approach show promising results when applied on the CC component, it remains to be investigated whether they scale up to other components of the ASML code base. In particular, the question can be raised whether our results can be generalised to larger components, developed by other developers. This may have an effect on the way the parameter checking concern is implemented, for example.

7. CONCLUDING REMARKS

In this paper, we have shown how a idioms-based solution to crosscutting concerns as occurring in systems software can be migrated automatically into a domain-specific aspect-oriented solution. The approach is illustrated by zooming in on a particular concern, namely parameter checking. Our approach includes a number of different elements:

- Characterization of the idioms-based approach, resulting in a *concern verifier* that can check the way the concern is coded;
- Representation of the concern in an aspect-oriented domain-specific language, which can be mapped to a dialect of the general purpopse AspectC language;
- A migration strategy for existing components, including an aspect extractor and a conservative translator.

We also discussed the advantages of the aspect-oriented solution compared to the idioms-based solution. Our results indicate that introducing aspects significantly reduces the code size, removes the scattering and code duplication, and improves the correctness and consistency of the concern implementation as well as the understandability of the application.

8. REFERENCES

[1] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[2] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code. In *Proceedings of the Joint European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)*, pages 88–98. ACM Press, 2001.

[3] A. van Deursen and P. Klint. Little languages: Little maintenance? *Journal of Software Maintenance*, 10:75–92, 1998.

[4] William G. Griswold, Yoshikiyo Kato, and Jimmy J. Yuan. AspectBrowser: Tool Support for Managing Dispersed Aspects. Technical Report CS1999-0640, University Of California, San Diego, 3, 2000.

[5] Gregor Kiczales, John Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the Europeen Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

Selective Introduction of Aspects for Program Comprehension

Andy Zaidman*, Toon Calders+, Serge Demeyer*, Jan Paredaens+

+ Advanced Database Research and Modelling (ADReM)
* Lab On Re-Engineering (LORE)

University of Antwerp
Department of Mathematics and Computer Science
Middelheimlaan 1, 2020 Antwerp, Belgium
{Andy.Zaidman, Toon.Calders, Serge.Demeyer, Jan.Paredaens}@ua.ac.be

Abstract

We propose a technique that uses webmining principles on event traces for uncovering important classes in a system's architecture. These classes can form starting points for the program comprehension process. Furthermore, we argue that these important classes can be used to define pointcuts for the introduction of aspects. Based on a medium-scale case study – Apache Ant – and detailed architectural information from its developers, we show that the important classes found by our technique are prime candidates for the introduction of aspects.

1 Introduction

Program comprehension is the process of understanding a system through feature and documentation analysis [11]. Gaining understanding of a program is a time-consuming task taking up to 40% of the time-budget of a maintenance operation [15]. The manner in which a programmer gets understanding of a software system varies greatly and depends on the individual, the magnitude of the program, the level of understanding needed, the kind of system, ... [10]

Studies and experiments reveal that the success of decomposing a program into effective mental models depends on one's general and program-specific domain knowledge. While a number of different models for the cognition process have been identified, most models fall into one of three categories: top-down comprehension, bottom-up comprehension or a hybrid model combining the previous two [12]. The top-down model is traditionally employed by programmers with code domain familiarity. By drawing on their existing domain knowledge, programmers are able to efficiently reconcile application source code with system goals. The bottom-up model is often applied by programmers working on unfamiliar code [4]. To comprehend the application, they build mental models by evaluating program code against their general programming knowledge [11].

For large industrial-scale systems, the program comprehension process requires the inspection and study of a significant number of packages, classes and code. As such, a semi-automated process in which an analysis tool supports the identification of key classes in a system's architecture and presents these to the user suits the hybrid cognitive model that is frequently used in large-scale systems [11].

Program understanding can be attained by using one of several strategies, namely (1) static analysis, i.e., by examining the source code, (2) dynamic analysis, i.e., by examining the program's behavior, or (3) a combination of both. In the context of object-oriented systems, due to polymorphism, static analysis is often imprecise with regard to the actual behavior of the application. Dynamic analysis, however, allows to create an exact image of the program's intended runtime behavior. Our actual goal is to find frequently occurring interaction patterns between classes. These interaction patterns can help us (1) build up understanding of the software, and (2) locate candidate introduction points for aspects.

In this paper we propose a technique that applies datamining techniques to event traces of program runs. As such, our technique can be catalogued in the dynamic analysis context. The technique we use was originally developed to identify important *hubs* on the Internet, i.e., pages with many links to authorative pages, based on only the links between web pages [9]. Hence, the Internet is viewed as a large graph. We verify that important classes in the pro-

gram correspond to the hubs in the dynamic call-graph of a program trace.

We apply the proposed technique to a medium-scale case study, namely Apache Ant. The results show that the *hubiness* is indeed a good measure for finding important classes in the system's architecture. Furthermore, based on these results we verify the hypothesis that these classes are good candidates for aspect introduction.

The organization of the paper is as follows. First, in Section 2, we give an overview of the different steps in the process and the different algorithms we use. Section 3 explains the datamining algorithm in detail, while in Section 4 the results of applying our technique on the case study are discussed. Section 5 explores related work, while Section 6 points to future research and concludes the paper.

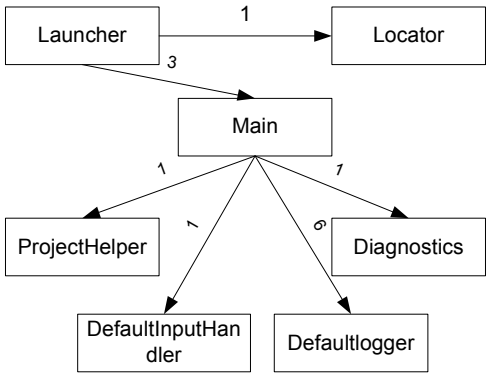


Figure 1. A compacted call graph.

2 Overview of our proposed technique

The technique we propose can be seen as a 4-step process. In this section we explain each of the 4 steps.

Define execution scenario. Applying dynamic analysis requires that the program is executed at least once. The execution scenario, i.e., which functionality of the program gets executed, is very important as it has a great influence on the results of the technique. For example, if the software engineer is reverse engineering a banking application and more specifically wants to know the inner workings of how interest rates are calculated, the execution scenario should at least contain one interest rate calculation. Furthermore, by keeping the execution scenario specific, i.e., only calculating the interest rate, the final results will be more precise.

Non-selective profiling. Once the execution scenario has been defined, the program must be executed according to the defined scenario. During the execution all calls to and returns from methods are logged in the event trace. For this step, we relied on a custom-made JVMPI[1] profiler.
Please note however that even for small and medium-scale software systems and precisely defined execution scenarios event traces become very large (for our case study the trace consisted of 24 270 064 events for an execution time of 23s).

Datamining. By examining the event trace we want to discover the classes in the system that play an active role in the execution scenario. Classes that have an active role are classes that call upon many other classes to perform functions for them.

In Figure 1 we show an example of a *compacted call graph*. The compacted call graph is derived from the dynamic call graph; it shows an edge between two classes A $\rightarrow$ B if an instance of class A sends a message to an instance of class B. The weights on the edges give an indication of the tightness of the collaboration as it is the number of unique messages that are sent between instances of both classes.
This compacted call graph is the input to the datamining algorithm that is presented in detail in section 3.

Selective introduction of aspects. The goal we wish to attain is guiding the software engineer through the software in order to help him/her in his/her program comprehension process. Because the original event trace is (1) too large to study directly (even in a visualized form), and (2) shows too many unimportant sections, e.g. long loops in the execution, we want to be able to deliver the software engineer with a number of *slices* of the trace that form good starting points for program understanding purposes.
To the user, these starting points can be:

- Pointers to classes: the user should begin his/her investigation from these classes and analyze them and their collaborating classes manually.
- A visualization, e.g. an interaction diagram, of the classes deemed important and their immediate collaborators. This set of classes can e.g. be found by introducing aspects with the `cflow` pointcut designator [8] on all classes deemed important.

As a side effect of this heuristical detection of important classes, we expect to find opportunities for aspect refactorings [14, 13].

As validation we propose to verify whether the classes our technique marks as important are also deemed important

[1] Java Virtual Machine Profiler Interface: for more information see: http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html

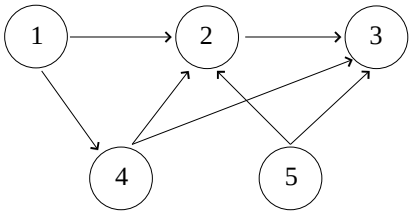


Figure 2. Example web-graph

by the developers. Furthermore, we will also compare the importance of these classes with the *Coupling Between Objects* (CBO) metric [3]. CBO can be seen as a typical static coupling measure which can help in identifying classes with a coordinating role.

3 Applying webmining techniques for program comprehension

In datamining, many successful techniques have been developed to analyze the structure of the web [2, 5, 9]. Typically, these methods consider the Internet as a large graph, in which, based solely on the hyperlink structure, important web pages can be identified. In this section we show how to apply these successful web mining techniques to a compacted call graph of a program trace, in order to uncover important classes.

First we introduce the HITS webmining-algorithm [9] to identify so-called hubs and authorities on the web. Then, the HITS algorithm is combined with the compacted call graph. We argue that the classes that are associated with good "hubs" in the compacted call graph are good candidates for the introduction of aspects as well.

3.1 Identifying hubs in large webgraphs

In [9], the notions of *hub* and *authority* were introduced. Intuitively, on the one hand, hubs are pages that rather refer to pages containing information then being informative themselves. Standard examples include web directories, lists of personal pages, ... On the other hand, a page is called an authority if it contains useful information. Hence, a web-page is a good hub if it points to important information pages, e.g., to good authorities. A page can be considered a good authority if it is referred to by many good hubs. The HITS algorithm is based on this relation between hubs and authorities.

Example Consider the webgraph given in Figure 2. In this graph, 2 and 3 will be good authorities, and 4 and 5 will be good hubs, and 1 will be a less good hub. The authority of 2 will be larger than the authority of 3, because the only in-links that they do not have in common are $1 \rightarrow 2$ and $2 \rightarrow 3$, and 1 is a better hub than 2. 4 and 5 are better hubs than 1, as they point to better authorities.

The HITS algorithm works as follows. Every page i gets assigned two numbers; a_i denotes the authority of the page, while h_i denotes the hubiness. Let $i \rightarrow j$ denote that there is a hyperlink from page i to page j. The recursive relation between authority and hubiness is captured by the following formula's.

$$h_i = \sum_{i \rightarrow j} a_j \quad (1)$$

$$a_j = \sum_{i \rightarrow j} h_i \quad (2)$$

The HITS algorithm starts with initializing all h's and a's to 1, and repeatedly updates the values for all pages, using the formula's (1) and (2). If after each update the values are normalized, this process converges to stable sets of authority and hub weights [9].

It is also possible to add weights to the edges in the graph. Adding weights to the graph can be interesting to capture the fact that some edges are more important than others. This extension only requires a small modification to the update rules. Let $w[i,j]$ be the weight of the edge from page i to page j. The update rules become $h_i = \sum_{i \rightarrow j} w[i,j] \cdot a_j$ and $a_j = \sum_{i \rightarrow j} w[i,j] \cdot h_i$.

Example For the graph given in 2, the hub and authority weights converge to the following (normalized) values:

h_1	=	64	a_1	=	0
h_2	=	48	a_2	=	100
h_3	=	0	a_3	=	94
h_4	=	100	a_4	=	24
h_5	=	100	a_5	=	0

In the context of webmining, the identification of hubs and authorities by the HITS algorithm has turned out to be very useful. Because HITS only uses the links between webpages, and not the actual content, it can be used on arbitrary graphs to identify important hubs and authorities.

3.2 Identifying aspect candidates

Within our problem domain, hubs can be considered *coordinating classes*, while authorities correspond to classes providing small functionalities that are used by many other classes. As such, the hub classes play a pivotal role in a system's architecture. Therefore, hubs are excellent candidates for the introduction of aspects to monitor the runtime behavior of these coordinators.

Furthermore, by using the `cflow` pointcut designator, we are not only able to monitor these coordinating classes, but also the classes that get their orders from these coordinators. This strategy can furthermore be used for efficient dynamic slicing.

4 Case study – Apache Ant

Ant is an XML based Java build tool. We chose Apache Ant 1.6.1 because we consider it to be a medium-size program (98 681 LOC, 127 classes) and because of the extensive design information that is publicly made available by the developers. As such we have clear evidence about the classes the developers consider to be important[2]. This knowledge will help us in validating our technique.

As execution scenario we have chosen to let Ant build itself, i.e., we supplied the XML build file that comes with the Apache Ant 1.6.1 source code edition. This scenario was chosen because (1) the Ant build file is representative for typical Ant functionality and (2) it allows for easy verification of the results presented in this paper.

We applied our technique two times on our case study. The first time, we set the weights of the compacted call graph all to 1, for the second experiment we used as weights the number of methods called upon from another class; see also Section 2.

In Table 1 we list the result of the first experiment. We show the highest 15% of classes according to their hubiness. We compare these classes with the CBO metric and with the opinion of the Ant development team.

Table 1 shows that:

- The number of *false positives*, i.e. classes reported but not considered important by the developers, is 6 out of 15 (40%). In the case of the CBO metric this amounts to 7/12 (58%).
- *False negatives* on the other hand remain limited to just 1 out of 10. For the CBO metric this number equals 5 out of 10.

The number of false negatives can be considered very low and shows the value of using our technique. The number of false positives however is – at first sight – alarmingly high. This can be attributed to several facts:

1. the developers opinion is *subjective* and only mentions those classes (or constructions) they are most proud of or they themselves find most interesting.
2. the classes our technique finds should also be considered important, albeit less important than those mentioned in the design documents.

[2]The design documentation of Ant can be found at: http://codefeed.com/tutorial/ant_config.html

Class	Proposed algorithm	CBO	Ant docs
Project	x	x	x
UnknownElement	x		x
AntTypeDefinition	x		
Task	x	x	x
ComponentHelper	x	x	
Main	x	x	x
IntrospectionHelper	x	x	x
AbstractFileSet	x	x	
ProjectHelper	x	x	x
RuntimeConfigurable	x		x
SelectSelector	x		
DirectoryScanner	x		
Target	x		x
TaskAdapter	x		
ElementHandler	x		x
FileUtils		x	
BaseSelectorContainer		x	
XMLCatalog		x	
AntClassLoader		x	
FilterChain		x	
TaskContainer			x

Table 1. Correlation between hubiness, static coupling, and expert opinion.

Close inspection of the project's source code reveals that the results can be explained by a mixture of the above reasons. All classes that are highly-ranked through their hubiness are in fact classes that have a *coordinating role* in the system and as such make them interesting for program comprehension purposes.

Furthermore, Table 1 shows there is a big difference in precision with regard to the CBO metric.

The results of the second experiment, where we used the real weights calculated during the transformation from a call graph to a compacted call graph, are very similar. The important classes are now however not strictly in the upper 15%, but more in the upper 25%. Furthermore, a number of helper classes to the classes deemed important, now also have a high degree of hubiness. This comes from the fact that many of these helper classes make use of only a limited number of classes, but do use a lot of different methods. Hence, these helper classes do not use many other classes, but the ones they do use, are used very intensively. This intensity results in a large weight, which, on its turn, increases the relative hubiness.

Keeping this in mind, we advocate the use of the `cflow` pointcut on the important classes of the experiment with the weights set to 1. This way, the helper classes will also be

touched by the pointcut.

5 Related work

Tourwé and Mens [13] describe an experiment in which formal concept analysis is used to mine for *aspectual views*. An aspectual view is a set of source code entities, such as class hierarchies, classes and methods, that are structurally related in some way, and often crosscut a particular application. These aspectual views are used for aspect mining, but also for program comprehension purposes.

Breu and Krinke experimented with finding sets of methods that are always executed in the same sequence [1]. They argue that the found sets of classes are candidates for aspect refactoring.

6 Conclusion and future work

In this paper, we proposed a technique that uses webmining principles for uncovering important classes in a system's architecture. We believe that the automatic classification of classes w.r.t. their importance is a critical step in the identification of aspects candidates. A case study showed that the approach is promising.

In the future, we will pursue the idea of applying datamining techniques to uncover important trends and relations in dynamic traces. First of all, we will continue the work on the identification of uncovering important classes. In the future we want to explore the connections and differences with other, dynamic or static, coupling metrics.

Besides the application of the HITS algorithm, there are many other datamining techniques that might help the analysis of large event traces. Especially because of the potentially large scale of event traces, the use of scalable datamining techniques seems very promising. The following datamining techniques are good candidates for helping the analysis of large event traces:

- Besides the hubs and authorities framework, there are many other graph mining concepts that can be interesting in the context of event traces. For example, Pagerank [2] is another method for ranking pages according to importance. Also the identification of web communities might prove useful in identifying classes or methods that are intimately connected.

- The event trace is in fact a large call tree. There exist tree mining algorithms that search for frequent occurring subtrees. The identification of such subtrees allows for compacting the presentation of the event trace [6].

- It can be interesting to find frequently occurring sequences in event traces. This problem might be solved by applying episode mining algorithms.

As can be seen from this list of candidates, the possibilities for applying datamining for automating program understanding are numerous. For an overview of the datamining techniques, see [7]. We believe this approach is very promising, and therefore think that it can become an important research direction.

References

[1] S. Breu and J. Krinke. Aspect mining using dynamic analysis, 2003.

[2] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998.

[3] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactons on Software Engineering*, 20(6):476–493, 6 1994.

[4] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2003.

[5] D. Gibson, J. M. Kleinberg, and P. Raghavan. Inferring web communities from link topology. In *UK Conference on Hypertext*, pages 225–234, 1998.

[6] A. Hamoe-Lhadj and T. C. Lethbridge. An efficient algorithm for detecting patterns in traces of procedure calls, 2003. Workshop on Dynamic Analysis.

[7] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2000.

[8] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings ECOOP*, volume 1241, pages 220–242. Springer-Verlag, 1997.

[9] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999.

[10] A. Lakhotia. Understanding someone else's code: Analysis of experiences. *Journal of Systems and Software*, pages 269–275, Dec. 1993.

[11] D. Ng, D. R. Kaeli, S. Kojarski, and D. H. Lorenz. Program comprehension using aspects. In *ICSE 2004 Workshop WoDiSEE'2004*, 2004.

[12] N. Pennington. Comprehension strategies in programming. In *Empirical studies of programmers: second workshop*, pages 100–113. Ablex Publishing Corp., 1987.

[13] T. Tourwe and K. Mens. Mining aspectual views using formal concept analysis. In *Proceedings of SCAM Workshop*. IEEE, 2004.

[14] A. Van Deursen, M. Marin, and L. Moonen. Aspect mining and refactoring. In *Proceedings of REFACE03*, 2003.

[15] N. Wilde. Faster reuse and maintenance using software reconnaissance, 1994. Technical Report SERC-TR-75F, Software Engineering Research Center, CSE-301, University of Florida, CIS Department, Gainesville, FL.

Measuring the Effects of Software Aspectization

Mariano Ceccato and Paolo Tonella
ITC-irst
Centro per la Ricerca Scientifica e Tecnologica
38050 Povo (Trento), Italy
{ceccato, tonella}@itc.it

Abstract

The aim of Aspect Oriented Programming (AOP) is the production of code that is easier to understand and evolve, thanks to the separation of the crosscutting concerns from the principal decomposition. However, AOP languages introduce an implicit coupling between the aspects and the modules in the principal decomposition, in that the latter may be unaware of the presence of aspects that intercept their execution and/or modify their structure. These invisible connections represent the main drawback of AOP. A measuring method is proposed to investigate the trade-off between advantages and disadvantages obtained by using the AOP approach. The method that we are currently studying is based on a metrics suite that extends the metrics traditionally used with the OO paradigm.

1 Introduction

When existing software is migrated to Aspect Oriented Programming (AOP), crosscutting concerns are separated from the principal decomposition and are encapsulated inside dedicated modularization units (aspects). Maintenance of the resulting code is expected to be easier, thanks to the possibility of modifying locally the crosscutting behavior. However, a novel kind of (implicit) coupling is introduced by AOP languages. In fact, the code that belongs to the principal decomposition might be unaware of the presence of aspects that intercept its execution and/or modify its structure. This creates a twofold dependence: on one hand, the aspect code works properly only under given assumptions on the code in the principal decomposition. Such assumptions may become invalid during code evolution. On the other hand, the overall behavior depends both on the code in the principal decomposition *and* on the aspect code, so that a change in the latter might affect the former. If not controlled, such kind of coupling might reduce or cancel at all the potential benefits coming from the separation of crosscutting functionalities from the principal decomposition.

The position of the authors is that the trade-off between the advantages obtained from the separation of concerns and the disadvantages caused by the coupling introduced by the aspects must be investigated in more detail, in order for AOP to gain a wider acceptance. Empirical studies should be conducted to evaluate costs and benefits offered by the AOP solution with respect to the more traditional, Object-Oriented (OO) one, in terms of code understandability, evolvability, modularity and testability. Moreover, alternative AOP solutions could be contrasted empirically, in order to identify good/bad AOP practices, to be possibly encoded into a catalog of AOP patterns/anti-patterns.

The first step in this direction is the definition of a set of metrics to quantitatively assess the effects of the software "aspectization". Such metrics can be based on those widely used with OO software. Although some extensions of OO metrics to AOP are available in the literature [8, 9, 10, 11, 12, 13], none seems to address explicitly all the different kinds of coupling that aspects and objects can have between each other.

In the remaining of this paper we discuss OO metrics (Sec. 2) and consider their extension to AOP (Sec. 3). Then, our AOP metrics tool is described (Sec. 4), followed by its usage on an example (Sec. 5). Related works (Sec. 6) and conclusions (Sec. 7) terminate the paper.

2 OO metrics

The inadequacy of the metrics in use with procedural code (size, complexity, etc.), when applied to OO systems, led to the investigation and definition of several metrics suites accounting for the specific features of OO software. However, among the available proposals, the one that is most commonly adopted and referenced is that by Chidamber and Kemerer [4]. We argue that a shift similar to the one leading to the Chidamber and Kemerer's metrics is necessary when moving from OO to AOP software.

Some notions used in the Chidamber and Kemerer's suite can be easily adapted to AOP software, by unifying classes and aspects, as well as methods and advices. Aspect introductions and static crosscutting require minor adaptations. However, novel kinds of coupling are introduced by AOP, demanding for specific measurements. For example, the possibility that a method execution is intercepted by an aspect pointcut, triggering the execution of an advice, makes the intercepted method coupled with the advice, in that its behavior is possibly altered by the advice. In the reverse direction, the aspect is affecting the module containing the intercepted operation, thus it depends on its internal properties (method names, control flow, etc.) in order to successfully redirect the operation's execution and produce the desired effects.

In the following section, the Chidamber and Kemerer's metrics suite is revised. Some of the metrics are adapted or extended, in order to make them applicable to the AOP software.

3 AOP metrics

Since the proposed metrics apply both to classes and aspects, in the following the term *module* will be used to indicate either of the two modularization units. Similarly, the term *operation* subsumes class methods and aspect advices/introductions.

WOM (Weighted Operations in Module): *Number of operations in a given module.*

Similarly to the related OO metric, WOM captures the internal complexity of a module in terms of the number of implemented functions. A more refined version of this metric can be obtained by giving different weights to operations with different internal complexity.

DIT (Depth of Inheritance Tree): *Length of the longest path from a given module to the class/aspect hierarchy root.*

Similarly to the related OO metric, DIT measures the scope of the properties. The deeper a class/aspect is in the hierarchy, the greater the number of operations it might inherit, thus making it more complex to understand and change. Since aspects can alter the inheritance relationship by means of static crosscutting, such effects of aspectization must be taken into account when computing this metric.

NOC (Number Of Children): *Number of immediate subclasses or sub-aspects of a given module.*

Similarly to DIT, NOC measures the scope of the properties, but in the reverse direction with respect to DIT. The number of children of a module indicates the proportion of modules potentially dependent on properties inherited from the given one.

CAE (Coupling on Advice Execution): *Number of aspects containing advices possibly triggered by the execution of operations in a given module.*

If the behavior of an operation can be altered by an aspect advice, due to a pointcut intercepting it, there is an (implicit) dependence of the operation from the advice. Thus, the given module is coupled with the aspect containing the advice and a change of the latter might impact the former. Such kind of coupling is absent in OO systems.

CIM (Coupling on Intercepted Modules): *Number of modules or interfaces explicitly named in the pointcuts belonging to a given aspect.*

This metric is the dual of CAE, being focused on the aspect that intercepts the operations of another module. However, CIM takes into account only those modules and interfaces an aspect is aware of – those that are explicitly mentioned in the pointcuts. Sub-modules, modules implementing named interfaces or modules referenced through wildcards are not counted in this metric, while they are in the metric CDA (see below), the rationale being that CIM (differently from CDA) captures the *direct* knowledge an aspect has of the rest of the system. High values of CIM indicate high coupling of the aspect with the given application and low generality/reusability.

CMC (Coupling on Method Call): *Number of modules or interfaces declaring methods that are possibly called by a given module.*

This metric descends from the OO metric CBO (Coupling Between Objects), which was split into two (CMC and CFA) to distinguish coupling on operations from coupling on attributes. Aspect introductions must be taken into account when the possibly invoked methods are determined. Usage of a high number of methods from many different modules indicates that the function of the given module cannot be easily isolated from the others. High coupling is associated with a high dependence from the functions in other modules.

CFA (Coupling on Field Access): *Number of modules or interfaces declaring fields that are accessed by a given module.*

Similarly to CMC, CFA measures the dependences of a given module on other modules, but in terms of accessed fields, instead of methods. In OO systems this metric is usually close to zero, but in AOP, aspects might access class

fields to perform their function, so observing the new value in aspectized software may be important to assess the coupling of an aspect with other classes/aspects.

RFM (Response For a Module): *Methods and advices potentially executed in response to a message received by a given module.*

Similarly to the related OO metric, RFM measures the potential communication between the given module and the other ones. The main adaptation necessary to apply it to AOP software is associated with the *implicit* responses that are triggered whenever a pointcut intercepts an operation of the given module.

LCO (Lack of Cohesion in Operations): *Pairs of operations working on different class fields minus pairs of operations working on common fields (zero if negative).*

Similarly to the LCOM (Lack of Cohesion in Methods) OO metric, LCO is associated with the pairwise dissimilarity between different operations belonging to the same module. Operations working on separate subsets of the module fields are considered dissimilar and contribute to the increase of the metric's value. LCO will be low if all operations in a class or an aspect share a common data structure being manipulated or accessed.

CDA (Crosscutting Degree of an Aspect): *Number of modules affected by the pointcuts and by the introductions in a given aspect.*

This is a brand new metric, specific to AOP software, that must be introduced as a completion of the CIM metric. While CIM considers only explicitly named modules, CDA measures all modules possibly affected by an aspect. This gives an idea of the overall impact an aspect has on the other modules. Moreover, the difference between CDA and CIM gives the number of modules that are affected by an aspect without being referenced explicitly by the aspect, which might indicate the degree of generality of an aspect, in terms of its independence from specific classes/aspects. High values of CDA and low values of CIM are usually desirable.

The proposed metric suite has no completeness claim and needs to be adapted for specific measurement goals (e.g., following the GQM approach [1]). While all the proposed metrics can be used to compare alternative AOP implementations, not all of them can be applied when an OOP program is migrated to AOP. CAE and CIM do not make sense in OOP, thus an overall TC (Total Coupling) metric should be used instead, counting the total number of coupling relationships between modules (either of type CAE, CIM, CMC or CFA). Of course, this is not the sum of the four metrics. Individual coupling metrics are still of interest to understand where a given TC value originates from. Similarly to CAE and CIM, CDA does not apply to OOP. However, its value for the migrated AOP program is interesting when compared to CIM, as explained above.

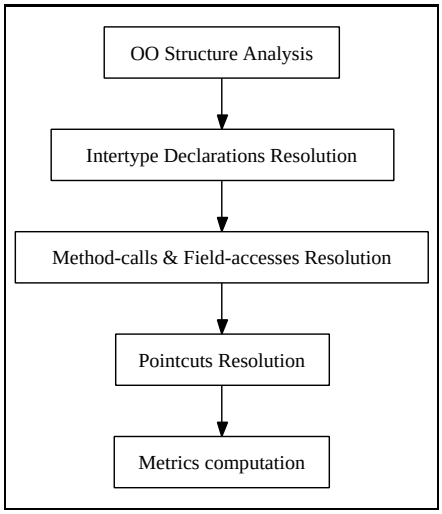


Figure 1. *Metrics tool.*

4 Metrics tool

To assess the proposed metrics suite, we developed an AOP metrics tool that computes all the proposed measures for code written in the AspectJ [7] language. The tool exploits a static analyzer developed in TXL [5]. Figure 1 shows the internal organization of the tool, focusing on the modules required to compute the AOP metric values.

The first module of the tool takes as input all the source classes, interfaces and aspects and performs some standard static OO code analysis, to detect the structure of the modules, in terms of their fields, operations and inheritance relationships. Such information is stored in a data base.

After the first module, the second one can be run, performing more accurate analysis. Each aspect is processed for a second time in order to detect the inter-type declarations, in terms of field introductions, method introductions and changes of class/interface inheritance relationships. The resulting data are stored in the same data base, being associated to the target class as if the information came from the analysis of the class itself. The name of the aspect responsible for such introductions is however recorded. In this way the first step of the weaving algorithm is realized.

The next module of the tool detects the method-call relationship. Moreover, it discovers the field-access relationship between operations and fields (both belonging to the same module or to other modules). For such an analysis a

symbol table, mapping the variables to the respective type, is maintained and pushed onto the stack whenever a new scope is opened. The symbol table is required to know the type of each method invocation target, return value and accessed field. Such type information is stored in the database under construction. Polymorphic calls are resolved conservatively with all methods that possibly override the invoked one.

The fourth step is the most complex one. It completes weaving by resolving all the pointcuts in the aspect code, thus producing the corresponding join points in the intercepted code. The algorithm for this phase starts coping with the primitive pointcuts, which are resolved using the inheritance, invocation and access information collected so far. Then, it composes the join points according to the union, intersection and negation operators used in the pointcut specifications. When all the pointcuts are resolved, the advices can be associated to the advised methods, storing this relationship in the available data base

The final step concerns the computation of the metrics. The value of a metric for a given module is obtained just by running a query on the database. The overall value of a metric for the whole system is computed as the median of the values computed for all the modules in the system.

5 Example

The proposed metrics have been computed for a small example, taken from the implementation of some design patterns [6] provided by Jan Hannemann [1] both in Java and in AspectJ.

Our test is the implementation of the *Observer* design pattern [6], in which there are two distinct roles, the *Subject* and the *Observer*. The Subject is an entity that can be in several different states. Some of the state changes are of interest to the Observer, which may take some actions in response to the change.

The Observer pattern requires that the Observer registers itself on those Subjects it intends to observe. The Subject maintains a list of the Observers registered so far. When the Subject changes its state, it notifies the Observers of the change, so that the Observers can take the appropriate actions.

In the OO implementation by Jan Hannemann, this design pattern consists of two interfaces, `ChangeSubject` and `ChangeObserver`, with the abstract definitions of the Subject and Observer roles. Moreover, the implementation contains the `Point` and the `Screen` classes, the first playing the role of Subject whereas the second plays both roles in two different instances of the pattern. The `Main` class contains the code to set up the two different pattern instances and run them. In the first pattern instance `Point` acts as the Subject and `Screen` as the Observer. In the second case, an instance of the class `Screen` is the Subject, while other instances of the same class are its Observers.

The AOP implementation contains a different version of the classes `Point` and `Screen`, with no code regarding the Subject/Observer roles. `ObserverProtocol` is an abstract aspect defining the general structure of the aspects that implement the Observer pattern. This abstract aspect is extended by `ScreenObserver`, `ColorObserver` and `CoordinateObserver`. These concrete aspects contain the actual implementation of the protocol. By means of inter-type declarations, they impose roles onto the involved classes and by means of appropriate pointcuts they specify the Subject actions to be observed. Moreover, these aspects contain the mapping that connects a Subject to its Observers. The class `Main` runs the code for the initialization of the patterns an for their execution.

version	**WOM**	**DIT**	**NOC**	*TC*	**RFM**	**LCO**	**CDA**
java	3	1	0	*2*	7	1-12	N.A.
aspectj	1	2	0	*3*	2	0	3

Table 1. *Metrics for the Observer design pattern.*

version	**CAE**	**CIM**	**CMC**	**CFA**	**TC**
java	0	0	2	0	**2**
aspectj	0	2	1	0	**3**

Table 2. *Coupling Metrics for the Observer design pattern.*

We applied our metric suite to the two implementations of the Observer pattern. The median values produced by the tool are shown in Table 1. The value of LCO for the OO code is indicated as 1-12, since these two values are adjacent to the median point. The *TC* column contains the value for total coupling. Detailed values for all different coupling kinds are shown in Table 2.

We can notice a general improvement of some metrics (WOM, LCO, and RFM), no change a metric (NOC) and a worse value of DIT (due to the super-aspect `ObserverProtocol`) and of TC. While in general the values change only a little bit, for RFM the change is relatively high, passing from 7 to 2. LCO is also affected positively, going from 1-12 to 0. The cost to be paid for such improvements is an increase of the coupling metric TC as expected. Looking at Table 2, we can have a detailed insight on the reasons for the coupling increase. Even if

[1] http://www.cs.ubc.ca/~jan/AODPs

there is a decrease of the method coupling (CMC) there is a much bigger increasing of the coupling regarding the aspects which intercept method executions (CIM). However, the fact that the value of CAE is higher than that of CIM indicates that the aspects have only a partial knowledge of the classes they are affecting and contain quite generic, independent pointcut definitions.

6 Related work

The cohesion measure called *Module-Attribute Cohesion* in [13] is based on the same dependences between operations and fields that we consider in our LCO metric, but, differently from our metric, it is not an extension of the LCOM metric proposed in [4]. As regards the proposed coupling metrics, while CIM, CMC and CAE correspond to the *Pointcut-class*, *Method-method* and *Pointcut-method* dependence measures presented in [12], CDA has no counterpart in [12].

Similarly to us, the authors of [9, 10] considered the Chidamber and Kemerer's metric suite, properly adapted to AOP. However, they do not recognize the different nature of the various kinds of coupling introduced by the aspects. The authors of [8] added a few metrics to capture the level of scattering of the application concerns. However, the definition of such metrics (*SoC metrics*) is not operational, thus making it difficult to compute them automatically. The expected effects of AOP on the Chidamber and Kemerer's metrics are analyzed in [11].

The indications in [2, 3] on the definition of cohesion and coupling metrics for OO systems will be considered in our future work, in order to possibly refine the proposed AOP metrics for such attributes.

7 Conclusions

Most research in AOP is focused on new design processes, languages and frameworks to support the new paradigm. However, no strong empirical evaluation was conducted to assess the effects of AOP adoption. The first step in this direction consists of defining a metrics suite for AOP software, designed so as to capture the novel features introduced by this programming style. We contributed to the ongoing discussion on such metrics by distinguishing among the different kinds of coupling relationships that may exist between modules and by proposing a new metric for the crosscutting degree of an aspect (CDA). Moreover, we conducted a small case study to evaluate the information carried by the proposed metrics when applied to an OO system and to the same system migrated to AOP. Results indicate that meaningful properties, such as the proportion of the system impacted by an aspect and the amount of knowledge an aspect has of the modules it crosscuts, are captured by the proposed metrics (CDA and CIM repsectively). We envisage the definition of a common set of AOP metrics, to be adopted by the AOP community, in order to simplify the comparison of the results obtained by different research teams and to have a standard evaluation method.

References

[1] V. Basili, G. Caldiera, and D. H. Rombach. *The Goal Question Metric Paradigm, Encyclopedia of Software Engineering*. John Wiley and Sons, 1994.

[2] L. Briand, J. Daly, and J. Wuest. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3(1):65–117, 1998.

[3] L. Briand, J. Daly, and J. Wuest. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999.

[4] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[5] J. Cordy, T. Dean, A. Malton, and K. Schneider. Source transformation in software engineering using the TXL transformation system. *Information and Software Technology*, 44(13):827–837, 2002.

[6] E. Gamma, R. Helm, R.Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley Publishing Company, Reading, MA, 1995.

[7] I. Kiselev. *Aspect-Oriented Programming with AspectJ*. Sams Publishing, Indianapolis, Indiana, USA, 2002.

[8] C. Sant'Anna, A. Garcia, C. Chavez, A. von Staa, and C. Lucena. On the reuse and maintenance of aspect-oriented software: An evaluation framework. In *17o. Simpsio Brasileiro de Engenharia de Software*, pages 19–34, 2003.

[9] S. L. Tsang, S. Clarke, and E. Baniassad. An evaluation of aspect-oriented programming for java-based real-time systems development. In *The 7th IEEE International Symposium on Object-oriented Real-time distributed Computing, ISORC*, 2004.

[10] S. L. Tsang, S. Clarke, and E. L. A. Baniassad. Object metrics for aspect systems: Limiting empirical inference based on modularity. 2000.

[11] A. A. Zakaria and H. Hosny. Metrics for aspect-oriented software design. In *AOM: Aspect-Oriented Modeling with UML, AOSD*, March 2003.

[12] J. Zhao. Measuring coupling in aspect-oriented systems. In *Proc. of the 10th International Software Metrics Symposium (METRICS)*, Chicago, Illinois, USA, September 2004.

[13] J. Zhao and B. Xu. Measuring aspect cohesion. In *Proc. International Conference on Fundamental Approaches to Software Engineering (FASE), LNCS 2984*, pages 54–68, Barcelona, Spain, March 2004. Springer-Verlag.