Institute of Electronics, Information, and Communication Engineers

社団法人 電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
SS2005-38 (2005-08)

アスペクトマイニングを自動的に行うツールの提案

Livieri Simone† 石尾 隆† 楠本 真二† 井上 克郎†
† 大阪大学 大学院情報科学研究科 コンピュータサイエンス専攻 〒560-8531 豊中市待兼山町 1-3
E-mail: †{simone,t-isio,kusumoto,inoue}@ist.osaka-u.ac.jp

あらまし ソフトウェアは，その規模が増大し，より複雑なものとなってきている．ソフトウェアが複雑化するに従って，保守性，モジュール性が悪化することから，これらの品質を改善するための変更作業がしばしば行われている．これに伴って，アスペクト指向プログラミングへの注目も高まっており，既存のシステムをアスペクト指向設計に改変する作業も増加している．このような問題に対して，アスペクトマイニングと呼ばれる，横断的関心事の候補をソフトウェア中から発見し，リファクタリングを支援するための手法が研究されている．本稿では，アスペクトマイニングを自動化する試験的な手法について述べる．

キーワード アスペクト, アスペクトマイニング, アスペクト指向プログラミング, 分析

Planning an Automated Aspect Mining Tool

Simone LIVIERI†, Takashi ISHIO†, Shinji KUSUMOTO†, and Katsuro INOUE†

† Graduate School of Information Science and Technology, Osaka University, 1-3, Machikaneyama-cho, Toyonaka, Osaka, 560-8531, Japan
E-mail: †{simone,t-isio,kusumoto,inoue}@ist.osaka-u.ac.jp

Abstract Software systems are growing in size and complexity. Increased complexity often leads to decresed mainainability and bad modularity, and re-engineering tasks is often performed in order to increase the key qualities of the system. With the incentive of a growing attention to aspect-oriented programming, the efforts to devise new techniques for refactoring legacy system into an aspect-oriented design have multiplied. Aspect mining tries to identify possible cross-cutting concerns in software systems and thus support this kind of refactoring. In this paper, we present a tentative approach to automated aspect mining analysis aiming at finding cross-cutting concerns that can be easily refactored to aspect-oriented artifacts.

Key words Aspect, aspect mining, aspect-oriented programming, analysis

1. Introduction

As software systems grow in size, they grow in complexity. Increasing complexity often leads to decreased readability and maintainability, and to bad modularity. Decomposing large systems into small parts, more easy to manage, comprehend and maintain, is what developers community currently do; but, while some behaviors of the systems can be easily decomposed and isolated, some other ones, by their very nature, are not: error-handling, logging, and tracing are typical examples of intrinsically *cross-cutting concerns* [1] whose implementing code exists in a redundant way, scattered across the code base.

Aspect-Oriented Programming (AOP) [2] aims at easing the modularization of such complex concerns. While retaining the benefits of the Object-Oriented approach, AOP intends to free the developer from the so-called *Tyranny of the Dominant Decomposition*: with the traditional programming languages, a program can be modularized in only one way at a time, and the many kinds of concerns that do not align with that modularization end up scattered across many modules and tangled with one another.

Since its inception in the late 90s, AOP has gained relevant support and attracted the attention of research community and a lot of different approaches that permit *aspect-oriented software development* have been developed; among them, the most mentioned are AspectJ [3], developed by the same people that coined the term Aspet-Oriented Programming [2], HyperJ [4], an offspring of subject-oriented programming [5], and AspectWerkz [6].

The increased attention also raised the question about how AOP can aid the re-engineering of legacy systems to improve their key quality attributes, like evolvability and reusability: tangled code implementing a specific concern has to be found and isolated and thereafter extracted into a modular artifact. These tasks are called *aspect-mining* and *aspect refactoring* respectively.

Our approach to automatic aspect mining exploits meth-

NII-Electronic Library Service

Institute of Electronics, Information, and Communication Engineers

ods aggregation for narrowing the search space and speed-up the mining process.

Section 2. presents an overview of the current status of aspect mining research. Section 3. describes our approach. Section 4. discusses conclusions and open issues.

2. Aspect Mining

Aspect-Mining is the task of searching for candidate aspects in existing system and isolating them into modular artifacts. To date, a number of research groups are working on this topic, and a number of tools and methods for aspect mining have been proposed.

One of the first effort in identifying cross-cutting concerns was the *Aspect Mining Tool* (AMT) of Hannemann and Kiczales [7]. AMT supports both text and type-based analysis and can be extended with other types of analysis. Because each of these analyzes has benefits and drawbacks, AMT has been set as a *multi-modal* analysis tool. The tool allows user defined queries based on type usage and regular expressions, displaying matching lines in specific colours in the source code.

FEAT [8] utilizes concern graphs to represent and document cross-cutting concerns. They are based on localizing an abstracted representation of the program elements contributing to the implementation of the concern. The structure of a concern is stored in a concern graph and, at the same time, the relationships between the concern's elements, such as classes, methods and fields are documented. FEAT supports the analysis of the dependencies between a concern and the rest of the program and allows the viewing of the source code, in a Java system, associated to a concern graph element. For concern identification, FEAT supports the use of structural queries and integrated lexical searches. The tool displays a concern graph as a collection of trees with respect to certain convention, e.g. the root of each tree is a class that contributes to the implementation of the concern. FEAT is also implemented as an Eclipse plug-in.

PRISM [9] is a framework for aspect mining based on the assumption that aspects can be defined in terms of structures in the source code. It uses two extensible concepts to represent aspects in software systems. The first concept is the *aspect fingerprint*: an aspect fingerprint abstracts a pattern, a structure in the source code, which can be used to locate aspects. The second concept is an *aspect footprint*: an aspect footprint is an abstraction of the location of a particular aspect fingerprint. The aspect mining is performed selecting a collection of aspect fingerprints and walking through the source code decomposition units generating a collection of aspect footprint. PRISM exists also as a plug-in for Eclipse: this specific extension of the framework implements a decomposition based on AMTEX an extension to AMT.

DynAMiT [10] is a dynamic aspect mining approach based on program traces that are generated during the execution of a program. These traces are investigated for recurring execution relations that describes certain classes of aspects of the software. The main classes considered are *inside-aspects*, when the call of a method m_1 occurs always inside a call of a method m_2, and *outside-aspects*, when the call of a method m_1 occurs always before (or after) the call of a method m_2. Furthermore they distinguish subclasses such as *inside-methods* that can be *first-in* or *last-in* inside a specific method call, and *outside-method* than can be *after* or *before* a specific method call.

Ophir [11] identifies initial refactoring candidates using a control-based comparison, followed by a filtering based on data dependence information. The initial identifications phase uses Program Dependence Graphs (PDG) to detect code clones that are successively filtered in order to eliminate undesirable refactoring candidates. The output of the filter phase is a set of candidate pairs, where each member of a pair is a from a different method, that are coalesced into sets of similar candidates.

While the tools currently available for aspect mining provide support for the process of identification of aspects in existing software, each approach suffers from one or more of the following drawbacks:

- users are required to have considerable amount of knowledge about the program being analyzed;
- search seeds need to be specified as an explicit input to the analysis;
- the identification and filtering phase is not fully automatic;
- the identification analysis can miss desiderable aspects;
- the analysis can take a large amount of time;

The main downfall of lexical searches is the requirement of a user to input a seed. Formulating a significative seed is a non-trivial task that requires the user to have great understanding of the code base. Even is a good seed is formed, the fragility of the lexical search limits its effectiveness. Lexical search is just a look for copy of the seed, while often duplicate code means, methods, statements or group of statements with the same semantics.

Exploratory tools can be of great help in aiding the identification of the aspects candidate and system comprehension, but the main drawback is that they require a lot of time to complete the identification due to the necessary interaction with the user.

Exhaustive approaches like [11] have the drawback or requiring a lot of time to complete.

In this paper we present an aspect mining analysis with the following properties:

- desiderable candidates for refactoring are identified automatically
- no input from the user is required

Our approach to automatic aspect mining exploits code clone analysis of portion of the code base that could contain aspect candidates. The next section presents some insights of the proposed technique.

3. Automated Aspect Mining

Detection of aspect candidates can be conducted in two ways: in a top-down approach, where the code that implements well-known aspects is searched, and in a bottom-up way, where *symptoms* of the lack of proper aspect support in the language used are searched. More specifically, these symptoms are *code scattering* and *code tangling*. Suitable techniques for identifying such symptoms may be clone detection (for scattering) or slicing (for untangling functionality).

Detecting aspect candidates can mean detecting very short code clones: for example, the code implementing the logging

NII-Electronic Library Service

concern is often a simple line of code inserted in many different locations of the source code. Different techniques for code clone detections have been proposed, each with its advantages and disadvantages: if applied to large code bases, more precise detection methods, like AST or PDG based ones, can be slow, and fast detection methods, like token based ones [12], can miss significative clones(注1). It is natural to think that devising a way for narrowing the search space will be of sure benefit to the code clone analysis. In our approach we narrow the search space through the indentification of so-called *cross-cut unit set* that are a subsets of the set containing all the methods of the code base containing cross-cutting code.

The proposed automated aspect mining analysis is based on AspectJ. In AspectJ there are several type of *advice*, such as: *before* and *after*. This advice can be execute at a specified joint-point. *Joint-points* are the points that one can specify within a program to execute a code segment, points such as the beginning of a method or before an access to a field.

What we pursue is to effectively discover aspect candidates that can be easily refactored into *before* aspects at the beginning of a method.

Our algorithm consists of four phases:

(1) *Construct* cross-cut unit sets.
(2) *Identify* code clones (token based).
(3) *Prune* the set of code clones (constraint based).
(4) *Classify* the aspect candidates (dependence based).

3.1 Constructing cross-cut unit sets

We call *cross-cut unit* a method or a class containing code implementing some cross-cutting concern. Scattered code is called from different places throughout the software system, and it is likely to have a very high value of the *fan-in metric*, defined in [13] as:

> "[...]the number of distinct method bodies that can invoke [a method] m[...]"

The analysis is made in three consecutive steps (the first two steps are taken from [13]):

Step 1 Automatic computation of the fan-in metric for all the methods in the target code base.

Step 2 Filtering of the result of the first step:

- Restrict the set of methods to those having a fan-in above a certain threshold.
- Remove accessor methods.
- Remove known utility methods

Step 3 For each method m in the restricted set compute the cross-cut units set C_m^X whose elements are all the methods containing a call to m. A method can belong to more than a cross-cut unit: this is absolutely not a problem because, as we will see soon, it just means that that method may contain more than one aspect candidate.

Figure 1 outlines a small example to illustrate this step. The methods m_1, m_2 invoke method m_1^X, method m_3 invokes methods m_1^X and m_2^X, and method m_4 invokes method m_2^X. In this scenario the methods m_1, m_2 and m_3 belongs to the same cross-cut unit set, similarly the methods m_3 and m_4 are in the same set.

(注1): Token based methods for code detection, detect clones with a minimum length of a given number of tokens: a small length value will result in too many reported clones, making them useless.

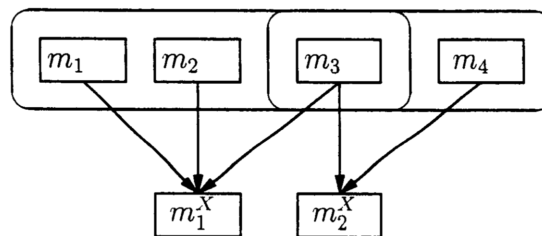

図 1 Cross-cut unit sets

3.2 Identifying code clones

Scattered code can span more than the call site of a high fan-in method, thus it is reasonable to thoroughly conduct a code clone analysis on each of the set $C_{m^X}^X$. As stated before, code clone analysis can be performed in several way. For our approach we decided that, for comparison purpose, two techniques will be evaluated:

- the first will be token based and will rely on the use of the tool CCFinder;
- the second will use a Program Dependence Graph and will borrow, and eventually tailor to our specific needs, the algorithm outlined in [14].

The identification process can be split in 2 steps:

Step 1 Let n be the total number of cross-cut unit sets. For each of the n cross-cut unit set $C_{m_i^X}^X$ $(i = 0, 1, \ldots, n)$, perform a code clone analysis with targets all the methods $m \in C_{m_i^X}^X$. The result of this step is the set $C_{m,m_i^X}^{CLONE}$ of code clones contained in the body of the method m;

Step 2 Compute the complete set of code clones for each method m' as:

$$C_m^{CLONE} = \bigcup_{i=0}^{n} C_{m,m_i^X}^{CLONE}$$

3.3 Pruning the clone sets

The result of the preceding step are n_m code clone sets C_m^{CLONE} with n_m being the number of methods in the target code base. We can expect that a code clone set may contain code clones that are not cross-cut by any of the m_i^X and, for this reason, of no interest to us. Hence, from each of the n_m code clone set C_m^{CLONE} we remove those code clones that don't comprise a call of a method m_i^X for some i in $\{0, 1, \ldots, n\}$.

3.4 Classifying the aspect candidates

Aspect candidates, that is, code clones that don't present data dependencies with the surrounding code are obviously more easy to refactor than code that has data dependencies. Candidates can then be *ranked* proportionally to the number of dependencies, with rank 0 being the highest rank.

More specifically, data dependencies can be of two kinds:

- dependencies *to* some data outside the clone (*outgoing-dependencies*);
- dependencies *from* some data outside the clone (*incoming-dependencies*).

Outgoing dependencies are less strict than incoming dependencies, and this can be expressed assigning a weight w_O to each outgoing dependency, and a weight w_I to each incoming dependency, with $w_I \gg w_O$. Let n_c^I and n_c^O be the number of incoming dependencies and the number of outgoing dependencies respectively of a code clone c, the rank r of

NII-Electronic Library Service

c can be computed as:

$$r(c) = n_c^I w_I + n_c^O w_O$$

The result of this step is an ordered list of code clones $c_0, c_1, \ldots c_n$ with $r(c_0) \leqq r(c_1) \leqq \ldots \leqq r(c_n)$

Clones at the head of the list are considered better candidates for an aspect-oriented refactoring.

4. Conclusion

This short paper presented a tentative approach at automating the mining of aspects in existing software systems. While we are confident of the soundness of the approach, at the time of writing there are no experimental data for validating it. Some questions can be answered only after thoroughly experimentation:

- what is the percentage of false positives?
- what is the percentage of false negatives?
- what are the performance of the approach in terms of time and memory usage?

文 献

[1] P.L. Tarr, H. Ossher, W.H. Harrison, and S.M.S. Jr., "N degrees of separation: Multi-dimensional separation of concerns," Proceedings of the International Conference on Software Engineering, pp.107–119, 1999.

[2] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin, "Aspect-oriented programming," Proceedings of the European Conference on Object-Oriented Programming, ed. M. Akşit and S. Matsuoka, Berlin, Heidelberg, and New York, pp.220–242, Springer-Verlag, 1997.

[3] AspectJ-Team, "The aspectj programming guide."

[4] P. Tarr, H. Ossher, V. Kruskal, and M. Kaplan, "The hyperj homepage." http://www.alphaworks.ibm.com/tech/hyperj.

[5] W. Harrison and H. Ossher, "Subject-oriented programming: a critique of pure objects," Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications, pp.411–428, 1993.

[6] J. Bonzr and A. Vasseur, "The aspectwerkz homepage." http://aspectwerkz.codehaus.org.

[7] J. Hannemann and G. Kiczales, "Overcoming the prevalent decomposition of legacy code," Proceedings of the Workshop on Advanced Separation of Concerns, 2001.

[8] M.P. Robillard and G.C. Murphy, "Concern graphs: finding and describing concerns using structural program dependencies," Proceedings of the International Conference on Software Engineering.

[9] C. Zhang and H.A. Jacobsen, "A prism for research in software modularization through aspect mining." Technical Communications, Middleware System Research Group, University of Toronto, September 2003.

[10] S. Breu and J. Krinke, "Aspect mining using event traces," Proceedings of the 19th Conference on Automated Software Engineering, IEEE Press, 2004.

[11] D. Shepherd, L. Pollock, and E. Gibson, "Design and evaluation of an automated aspect mining tool," Proceeding of the International Conference on Software Engineering Research and Practice, 2004.

[12] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multi-linguistic token-based code clone detection system for large scale source code," IEEE Transactions on Software Engineering, vol.28, pp.654–670, July 2002.

[13] M. Marin, A.v. Deursen, and L. Moonen, "Identifying aspects using fan-in analysis," Proceedings of the 11th Working Conference on Reverse Engineering (WCRE2004)., pp.132–141, IEEE Computer Society, 2004.

[14] J. Krinke, "Identifying similar code with program dependence graphs," Proc. Eigth Working Conference on Reverse Engineering, pp.301–309, 2001.

NII-Electronic Library Service