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.

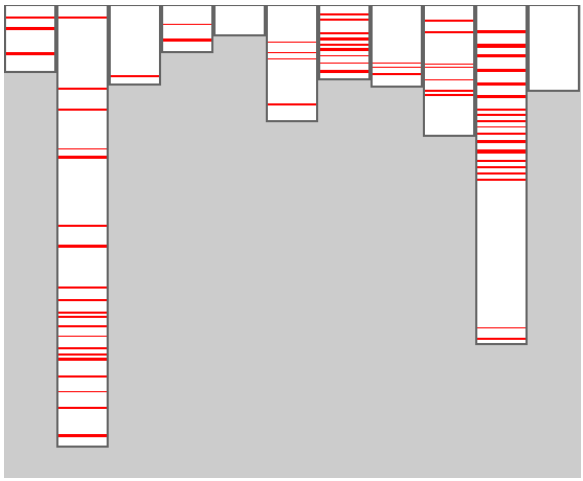

Figure 1. Scattering of the parameter checking concern.

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:

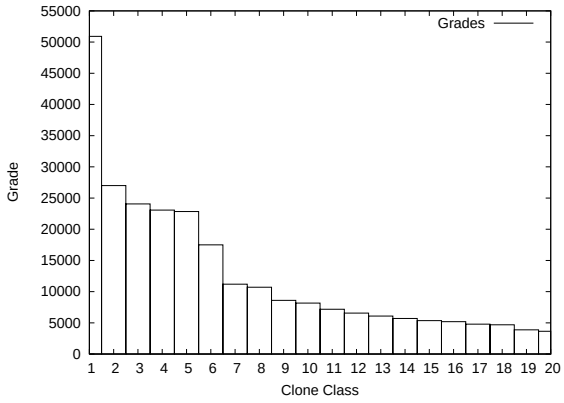

Figure 2. First 20 highest graded clone classes.

- **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.