A Prism for Research in Software Modularization Through Aspect Mining*

Charles Zhang and Hans-Arno Jacobsen
Department of Electrical and Computer Engineering &
Department of Computer Science
University of Toronto
{czhang,jacobsen}@eecg.toronto.edu

Abstract

The Prism project develops tools and techniques for discovering non-localized units of modularity in large software systems. A non-localized unit of modularity is manifest by code crosscutting concerns or aspects, such as logging, tracing, synchronization, and persistence. The objective is to be able to analyze legacy code bases containing well over thousands of classes with over millions of lines of code. Aspect discovery is based on aspect mining techniques developed in this research. The successful discovery of aspects will give rise to aspect oriented refactoring possibilities of the software system mined. Prism is developed as plug-in for the Eclipse integrated development environment complementing the AspectJ Development Tools efforts (AJDT) enhancing Eclipse with features for aspect oriented software development.

1 Introduction

Software systems often undergo significant evolution during their life time. This applies to, both, the requirements they must support and the target platforms they must operate on. As software architecture evolves it is often getting increasingly difficult to achieve or to maintain a high level of customizability, adaptability, and configurability. This can be attributed to the structure of the software system, which is becoming overly complicated and rigid. That problem is often manifested by the difficulties in achieving good programming abstraction and, more fundamentally, by the limitations of traditional software decomposition methods.

Aspect oriented programming, on the contrary, has introduced new design perspectives that permit the superimpositions of different abstraction models on top of one another. This is a very powerful technique for separating, modularizing, and simplifying crosscutting design concerns in (large) software systems. Aspect oriented software development is an emerging software development paradigm, which specifies abstractions, namely aspects, to modularize design and development concerns that cut across parts or all of the traditionally developed software system. Examples of such crosscutting concerns include synchronization, debugging, logging, memory management, security, and persistence concerns. The modularization of such concerns allows for higher-degrees of reuse throughout the software system and potentially other software systems and eases the maintainability of the system.

However, before aspect oriented techniques can be applied to re-engineer large software systems, the units of modularity, crosscutting concerns, and aspects have to be successfully identified in these systems. It is very hard to discern potential aspects

*Technical Communication, September 2003, Middleware Systems Research Group, University of Toronto.

of large software systems without sufficient tool support, due to their scattered and often non-systematic presence. The objective of the Prism project is to develop analysis techniques to support aspect discovery through aspect modeling, aspect mining, and modularity analysis.

This paper outlines several aspect-mining methods for discovering and locating non-localized programming concerns in (legacy) code bases. This paper also discusses the overall architecture and design of Prism, of which the purpose is to provide a platform to exercise and to evaluate the aspect-mining methods. Prism is designed as a framework with a flexible set of extension points that can be used to implement different types of aspect mining algorithms and mining tasks. The Prism Eclipse Plug-in integrates the Prism framework and aspect mining algorithms into the Eclipse platform. The Eclipse platform serves as the user interface to guide and manage the mining tasks.

2 Aspect Mining

Manually inspecting code to discern potential aspects is an arduous task. We have extensively experimented with different techniques to discover aspects in legacy code [**?**, **?**, **?**, **?**]. These investigations are driving our aspect mining algorithm and tool development efforts. The following briefly describes our mining approaches and discusses how it is supported by Prism.

The aspect mining process is guided by a human miner who knows or suspects the existence of aspects in the code. The miner begins this process with the initial description of the crosscutting structure of an aspect. This structure can be described, either in succinct lexical and type patterns, or gradually in several steps. We refer to such a description as the *characterization* of an aspect.

The use of matching in a code base based on lexical and type-based patterns works effectively with programs that have well-defined naming conventions and follow a consistent coding style. This method is appropriate for discovering crosscutting structures at the granularity of statements level inside method bodies. For example, code such as `log` or `trace` and types such as `Logger` or `Tracer` are fairly accurate in describing the crosscutting of the aspects *logging* and *tracing*. We use this method primarily to locate well-known aspects such as logging, tracing, synchronization, and others.

For aspects that have no statement-level crosscutting structures but are manifest by domain-specific semantics, it is difficult to characterize the aspect without a good understanding of the semantics of the program. We use a feedback-directed approach to find a good characterization of these type of aspects in several steps of refinements. That is, a lexical pattern can be used to describe the miner's intuition of what the crosscutting structure might look like. Matching results discovered in this step serve as feedback and assist the miner to refine the characterization of the perspective aspect. We use this approach to locate system specific aspects, not, generally, manifest in other software systems. This mining stage is assisted by a *type ranking* feature, which allows to rank the usage popularities of all class types in the system. Types that are used relatively widely in the code space provide good hints of potential aspects. For example, the tool would indicate that the class type `Assert` is the most widely used type in a particular code base. We then use this class type and quantify its presence.

Another type of crosscutting structure is the additional control flow incurred in the code through the coordination with an orthogonal concern. We can inspect the values involved in conditional branches and trace all the code involved in accessing these values through assignments, method parameter passing, and accessor methods. We essentially compute code slices based on the conditional statements. If these slices are not localized, we have identified very good candidates of crosscutting concerns. A tool can use this technique and evaluate all conditional statements in the system to find non-localized slices. Currently, we perform this technique manually by browsing through the source code with the help of Eclipse and Feat [**?**]. The objective is to support this functionality in the Prism platform eventually.

3 Prism Architecture

The architecture of Prism is based on the assumption that aspects can be defined in terms of the structure of the source code. We term these structures *aspect fingerprints*. Therefore, the functionality of the framework is primarily to provide effective means to define the following key elements in analyzing aspects in large systems: source code representations, aspect representation, mining algorithms and mining execution control. The rest of the section provides the functional descriptions of these elements.

3.1 Source code representation

The main purpose of the source code representation is to provide a sufficient abstraction of the information in a source-code unit. The abstraction serves as a model that contains all necessary information required for further analysis. The Prism source code representation consists of a set of classes aiming at providing a uniform abstraction of different types of decomposition units in order to support projects implemented in different languages. Figure **??** illustrates the type hierarchy of the source code representation classes together with all other key entities of the architecture. A *Decomposition Unit* (DU) represents the decomposition unit in the source program. This is the base class for extensions for all types of decomposition units. The *Object oriented decomposition unit* (OODU) captures the characteristics that are common to all object oriented decomposition units. And the *Java decomposition unit* (JDU) represents a Java class definition. It includes Java specific features, such as information regarding interfaces implemented by a class.

3.2 Aspect Representation

Prism uses two extensible concepts to represent aspects in software systems. The first concept is the *aspect fingerprint*. An aspect fingerprint abstracts a pattern, which can be used to locate aspects in the source code. The second concept is an *aspect footprint*. An aspect footprint is an abstraction of the location of a particular aspect fingerprint. Each aspect

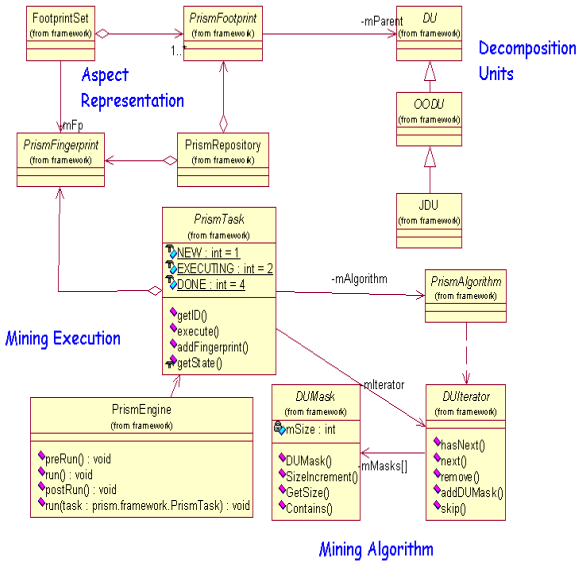

Figure 1: Prism Architecture

footprint has a parent, which is the *DU* where the footprint is located. Normally, an aspect fingerprint will find many instances of footprints across the code base. To better organize these footprints, the class `FootprintSet` encapsulates all footprints related to one particular aspect fingerprint in one entity. It also contains the reference to the corresponding aspect fingerprint instance.

3.3 Mining Algorithm

While we anticipate a semantic driven algorithm to tackle logic tangling caused by aspects, we currently rely on the repetitions of program structures to closely approximate their crosscutting structures. The architecture for mining algorithms is thus kept at its minimum. It assumes that a mining algorithm is simply a task that, given a collection of aspect fingerprints, walks through a collection of decomposition units in an arbitrary order and produces a collection of aspect footprints for each aspect fingerprint. The collection of decomposition units is accessed through

a *DUIterator* and its iterating sequence can be further influenced by a *DUMask*. A *PrismAlgorithm* defines a strategy for evaluating a collection of aspect fingerprints against a collection of DUs represented by a concrete *DUIterator*.

3.4 Mining Execution

This functional area includes classes that are responsible for executing and managing aspect mining tasks. The focus of the design of the Prism execution classes is to, firstly, support flexible assembly of aspect fingerprints and yet maintain simplicity in terms of the management of mining tasks. Each mining task is represented by the class *PrismTask*, which contains a collection of aspect fingerprints. A *PrismRepository* is a central storage place. Its primary purpose is to store the associations between aspect fingerprints and their corresponding sets of aspect footprints. The mining tasks are processed by the *PrismEngine*. The *PrismEngine* execute mining tasks either in batch style or individually.

Due to space limitations we cannot discuss the detailed algorithms, which will be described in a follow up paper.

4 Prism Eclipse Plug-in

The current Prism eclipse plug-in is built on top of an extension of the Prism mining framework presented in the previous section. This extension implements the functionality of AMTEX [**?**], which provides aspect discovering capabilites based on lexical patterns and type patterns for Java systems. In the Prism mining environment, the mining activities are represented by mining tasks and managed through the mining management console. The console also provides a consoliated view of all *fingerprints* defined in the systems, the summary of each mining task, and the *footprint*s of the *fingerprint*s contained in each mining task. The editor shows the actual places in the code that are represened by the *footprint*s. Figure **??** is a snapshot of the layout of the Prism management console. A Prism task can be defined in the task configuration view. This view allows the miner

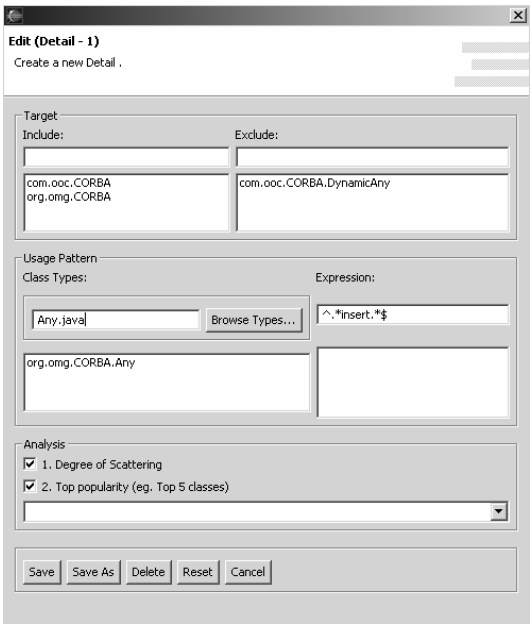

Figure 3: Prism Task Configuration View

to define classes that are to be analyzed by using the inclusion/exclusion functionality. The aspect fingerprint can be defined by specifying a collection of types and regular expressions. The miner can also specify the types of data that are collected in this view. Figure **??** shows a screen shot of the Prism task configuration view.

5 Related Work

Since aspect orientation is a relatively new software development paradigm, research addressing aspect mining is in its infancy. We are aware of only a few aspect mining approaches discussed to date [**?**, **?**, **?**]. All these approaches suffer from the fact that the user has to be intimately familiar with the code base analyzed and none of these approaches really discover aspects in a fully automated fashion. Clearly, this is a severe limitation of these approaches, should they be applied to analyzing large legacy software systems, where neither original developers, nor documentation

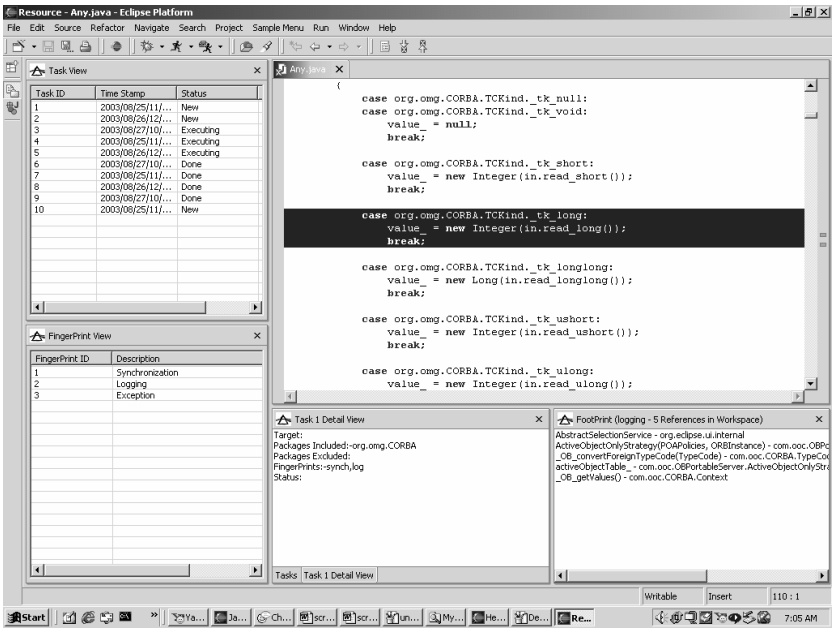

Figure 2: Prism Management Console

may be at hand. We briefly discuss these approaches.

The aspect mining tool (AMT) [**?**] is designed to capture the code scattering problem on the source code level. It is developed at the University of British Columbia and based on the AspectJ compiler. AMT emphasizes on visualizing perspective aspects in program code and is therefore well suited to study aspects in small software projects. AMT allows one to inspect the code scattering phenomenon by performing type and textual analysis in building a collection of all the types used in the program in combination with the entire source code space. We have built the extended aspect mining tool (AMTEX) [**?**] to overcome the limitation of visualization-based mining and scale the mining process to software systems with thousands of classes. The extended mining tool provides much larger flexibility in terms of composing mining activities, managing mining tasks and cross analyzing mining results. It is designed to fit the needs of mining very large software systems that consist of thousands of classes with millions of lines of code. AMTEX has been successfully applied to the aspect analysis of three open source Java implementations of the CORBA middleware platform, namely JacORB, ORBacus, and OpenORB (for details see [**?**, **?**, **?**].)

We think that there are a number of techniques, not directly related to aspect oriented software development, but helpful in solving the aspect mining problem. We briefly discuss these techniques next. Program slicing is a family of program decomposition techniques based on selecting statements relevant to a computation, even if they are scattered throughout a program [**?**]. It was originally designed for debugging where the statements not in the code of interest would be sliced away. Recent extensions to program slicing allow it to be used efficiently to extract reusable functions in existing code. Since locating and identifying aspects is a form of extracting reusable code, it is very likely that aspect mining can be based on program slicing technique for locating aspects. Other techniques, such as clone detection [**?**], design pattern detection [**?**], and code smell detection [**?**] may also inspire techniques for solving the

aspect mining problem.

6 Conclusion

Prism is an effort of providing the tool support for analyzing aspects in very large (legacy) code bases. It is based on an abstraction of aspect mining activities. These activities include the representation of a crosscutting concern (Prism aspect fingerprint), the representation of the location corresponding to one or a group of crosscutting concerns (Prism aspect footprint), and the algorithm (Prism aspect mining algorithm) used to take fingerprints as input and produce footprints as output. Since the Eclipse integrated development environment provides a natural platform for aspect mining, Prism is developed as an Eclipse plug-in and provides a set of GUI elements to allow the miner to define, modify, and execute mining tasks. The current version supports aspect mining based on lexical patterns and type patterns. We are currently undertaking extensive development towards more sophisticated extensions of the Prism framework as well as the Eclipse user interface. Future goals include extensions to allow the miner to work with sophisticated syntactical patterns to better capture crosscutting structure and to further refine the aspect mining methodology and algorithms to minimize the amount of knowledge a miner has to have about the code base mined.

Acknowledgments

We would like to thank Helen Shi for her contributions to the implementation of the Prism aspect mining plug-in. We are also very grateful to Harold Ossher for his help and support with this project. This project was supported by an IBM eIG 2003 grant.

References

[1] I. Baxter, A. Yahin, L. L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *nternational Conference on Software Maintenance*. IEEE, 1998.

[2] Jan Hannemann. The aspect mining tool. URL `http://www.cs.ubc.ca/~jan/amt/`.

[3] Dirk Heuzeroth, Thomas Holl, Gustav Högström, and Welf Löwe. Automatic design pattern detection. In *11th Internatinal Workshop on Program Comprehension, co-located with 25th International Conference on Software Engineering, Portland*. IEEE, May 2003.

[4] Filippo Lanubile and Giuseppe Visaggio. Extracting reusable functions by flow graph-based program slicing. *IEEE Trans. on Software Engineering*, 23(4):4–18, April 1987.

[5] N. Loughran and A. Rashid. Mining aspects. In *Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design (in conjunction with AOSD 2002)*, Enschede, Netherlands, April 2002.

[6] Martin Robillard. Feat: A tool for locating, describing, and analyzing concerns in source code. In *Eclipse Technology eXchange (ETX) at International Conference on Software Engineering (ICSE)*, Portland, Oregon, May 2003. URL, `http://www.cas.ibm.ca/conferences/etx/`.

[7] Eva van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Proceedings of the 9th Working Conference on Reverse Engineering*. IEEE Computer Society Press, October 2002.

[8] Charles Zhang, Gao Dapeng, and Hans-Arno Jacobsen. Extended aspect mining tool. URL `http://www.eecg.utoronto.ca/~czhang/amtex`, August 2002.

[9] Charles Zhang and Hans-Arno Jacobsen. Modularity analyzer and aspect extractor. In *Eclipse Technology eXchange (ETX) at International Conference on Software Engineering (ICSE)*, Portland, Og, May 2003. URL, `http://www.cas.ibm.ca/conferences/etx/`.

[10] Charles Zhang and Hans-Arno Jacobsen. Quantifying aspects in middleware platforms. In

2nd International Conference on Aspect Oriented Systems and Design, Boston, MA, March 2003.

[11] Charles Zhang and Hans-Arno Jacobsen. Quantifying aspects in middleware platforms. In *2nd International Conference on Aspect Oriented Systems and Design*, Boston, MA, March 2003.

[12] Charles Zhang and Hans-Arno Jacobsen. Refactoring middleware systems: A case study. In *International Symposium on Distributed Objects and Applications*, Catania, Sicily, Italy, 3-7 November 2003 2003. Lecture Notes In Computer Science, Springer Verlag.

[13] Charles Zhang and Hans-Arno Jacobsen. Refactoring middleware with aspects. *IEEE Trans. on Parallel and Distributed Systems*, 2003. (accepted for publication).