Timna: A Framework for Automatically Combining Aspect Mining Analyses

David Shepherd[1], Jeffrey Palm[2], Lori Pollock[1], and Mark Chu-Carroll[3]

[1]Computer and Information Sciences, University of Delaware, Newark, DE 19716
[2]Computer and Information Sciences, Northeastern University, Boston, MA 02115
[3]IBM T. J. Watson Research Ctr, 19 Skyline Dr., Hawthorne, NY 10532

ABSTRACT

To realize the benefits of Aspect Oriented Programming (AOP), developers must refactor active and legacy code bases into an AOP language. When refactoring, developers first need to identify refactoring candidates, a process called *aspect mining*. Humans perform mining by using a variety of clues to determine which code to refactor. However, existing approaches to automating the aspect mining process focus on developing analyses of a single program characteristic. Each analysis often finds only a subset of possible refactoring candidates and is unlikely to find candidates which humans find by combining analyses. In this paper, we present Timna, a framework for enabling the automatic combination of aspect mining analyses. The key insight is the use of machine learning to learn when to refactor, from vetted examples. Experimental evaluation of the cost-effectiveness of Timna in comparison to Fan-in, a leading aspect mining analysis, indicates that such a framework for automatically combining analyses is very promising.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Restructuring, reverse engineering, and reengineering

General Terms: Design, Experimentation

Keywords: Aspect mining, Reverse engineering, Machine learning, Program analysis

1. INTRODUCTION

Aspect Oriented Programming is an important new software development paradigm, allowing programmers to modularize a concern (a high-level concept) even if that concern cuts across the dominant decomposition of the application. Due to AOP's ability to group code that represents a particular concern, AOP increases code readability and helps decrease the cost of maintenance. However, re-organizing non-AOP code into AOP style is a challenging problem that continues to hinder AOP's adoption. Furthermore, refactoring code already written in AOP also might be necessary in cases where the programmers' view of code is limited (i.e., they missed opportunities to apply AOP) and in cases where the programmer does not understand AOP well. Refactoring code already written in AOP is similar to OO refactoring, with the main goal of reversing the decay of code quality that occurs over time [6].

The Problem and Illustrated Benefit. Porting code to AOP can be viewed as two tasks, identifying candidates for refactoring into AOP, and then refactoring the "mined" candidates. Our research focuses on the process of identifying the candidates for refactoring. Automatically identifying candidates can have many benefits, such as helping novice AOP programmers recognize refactoring opportunities in their own code [15] and showing advanced programmers more subtle opportunities.

In order to illustrate the potential usefulness of our research, consider the developer in Figure 1. Here, as the developer browses source code, hints are provided in the form of highlighted lines, which indicate *method calls* our framework has marked as refactoring candidates. The developer's attention is drawn to the highlighted lines, and s/he has the option to use knowledge of the source code, or further explore the source code, to form generalizations about how highlighted source lines could be moved to an aspect. We believe that the developer is likely to deduce the implicit rule (i.e., rule used in the source code, but not necessarily documented) using the hints provided in this example. This approach uses computational power to perform analyses and identify refactoring candidates, but relies on a human's generalization expertise (since generalization of this type would be expensive for a computer [13], but is usually easy for a human).

Overview and Contribution. We separate the identification of refactoring candidates into two subtasks – (1) providing seeds and (2) expanding the seed set. Seeds are points in a program which, based on vetted examples, are better represented in AOP code. Starting from seeds, the process of discovering and including all code relevant to a particular seed is called expanding the seed set.

Figure 1: Possible Application - Using Aspect Mining to Aid the User
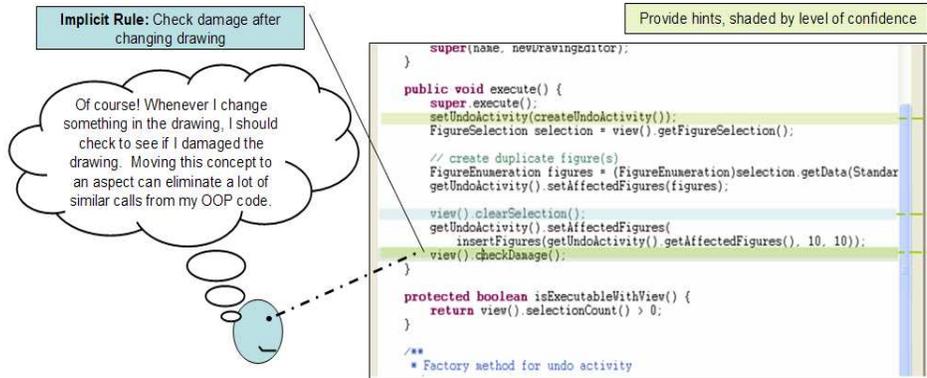
Tools exist that help users identify seeds, but these tools suffer from various drawbacks. Usually these tools are not automated, and require a lexical seed themselves [9, 10, 20]! Other tools, which are automated, may return spurious seeds and/or miss many relevant, obvious seeds, because current tools make a classification (i.e., the code is either part of a scattered concern or not) based on a single program analysis [17, 15, 2]. Tools also exist to help expand a given seed set, using program navigation. They are interactive tools, which means that identifying refactoring candidates in a large program (without a seed identifier) will require a large amount of human effort. These tools provide little to no help in making aspect mining decisions, only in navigating the code [19, 16]. However, we expect exploration tools to be very helpful at expanding a seed set, when given a seed set by a seed identifier. Exploration tools assist the programmer in locating relevant segments by allowing the programmer to easily navigate the program from each identified seed.

Our framework, Timna, automates the process of combining mining analyses to find seeds in a program, with high precision. Particularly, Timna's unique contributions are:

- the ability to exploit the combination of mining results from multiple code analyses to make classification decisions for seed identification,

- the application of machine learning to learn good AOP style from canonical examples,

- the generation of human readable rules for classifying code as mining seeds, and

- the discovery of several novel mining analyses during our initial use of Timna.

The major advantage over existing automated tools is that Timna is a framework in which new and existing mining analyses can be easily added and included in the decision as to whether a code segment is a seed of a scattered concern. The key insight is the application of a machine learning algorithm so Timna is able to learn which analyses are important in which situations, and construct rules that are not based on human intuition,

but founded on hard evidence in real programs. Timna indicates which analyses are important to classification and which are irrelevant by their in/exclusion in the rules that are generated. The human readable rules output by Timna provide a quick way of classifying a code segment as a scattered concern, as well as a way of studying characteristics of scattered concerns. Based on our examination of the rules generated in our case study, we invented several new analyses which proved useful in identifying seeds.

In this paper, we present the design, use and evaluation of Timna, our framework for combining aspect mining analyses. Section 2 presents the design of Timna's two phases, and explains how its design achieves the objective of combining mining analyses. Section 3 describes our application of machine learning for learning the characteristics of concerns, and highlights Timna's ability to learn. Section 4 describes the set of new and existing analyses combined in the current instantiation of Timna. Section 5 details our evaluation questions, subjects of study, experimental setup and methodology. Section 6 describes the results and analysis of our empirical evaluation. Section 7 surveys related work. We conclude and discuss future work motivated by our results in Section 8.

2. TIMNA DESIGN

The identification of seeds can be viewed as the problem of classifying code segments as being good candidates for refactoring into AOP. Code classification can be performed at different granularities, such as statement, basic block, or method level. Our current work focuses on method level classification, similar to existing aspect mining analyses. Thus, the main objective of the automatic classification tools we build from Timna is to take a program as input, and output a classification of each of its methods as refactoring candidates, and possibly categorized by syntactic properties of the code segment when not modularized into AOP.

With the application to be refactored as input, a single aspect mining analysis typically outputs a set of method candidates for refactoring, possibly with a categorization. In order to learn the rules for classifying code as refactoring candidates, Timna provides a frame-
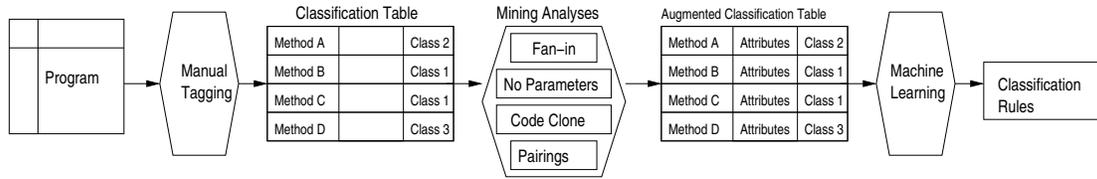
Figure 2: Training Phase
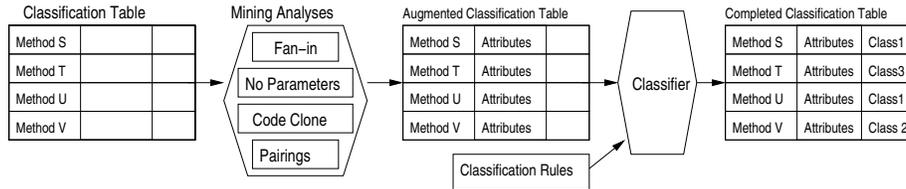

Figure 3: Classifying Phase

```
/**
 * Sets the end point.
 */
public void endPoint(int x, int y) {
    willChange();
    if(fPoints.size() < 2) {
        fPoints.addElement(new Point(x, y));
    }
    else
        fPoints.setElementAt(new Point(x, y), fPoints.size()-1);
    }
    changed();
}
```

Figure 4: Method Usage Matches Known Refactoring Candidates

```
void setP1(Point p1) {        public void setX(int x){
    this.p1 = p1;                 this.x = x;
    Display.update();             notifyObservers();
}                             }
```

Figure 5: Excerpt from AspectJ.com [1] (left) and AOP Design Patterns [11] (right)

work for building tools that consist of two phases: *training* and *classifying*.

2.1 The Off-line Training Phase

The method classifier is trained during the training phase with a set of mining analyses and a set of training applications (possibly only one application). The method classifier then executes using the training data to mine possible refactoring candidates on applications targeted for refactoring into AOP. The training phase is off-line, performed only once to learn and establish classification rules, while the classification phase is performed for each application to be mined. Figure 2 depicts the training phase of a Timna aspect mining tool.

2.1.1 Manual Tagging

In order to perform the training phase, the training program set is first manually analyzed and each method is classified, or tagged, as either being a "candidate", if it is determined to be a good candidate for refactoring,

or "not a candidate", if it is not. Currently, we tag a method as a candidate only if its use is analogous to a known, vetted example in AspectJ literature [1, 14, 8, 11]. For instance, we would tag the method `changed` (in Figure 4), because it is used much like the `update` method and the `notifyObservers` method (both in Figure 5). Method calls `update` and `notifyObservers` are both used as motivating examples of when to use AspectJ, and they are both from sources that are accepted by the AOP community.

As we were manually tagging the training program in our study, we observed that we were tagging several distinct syntactic categories of methods. We decided to label each "candidate" with a category as well, so that we could later see if these syntactic categories might aid the Timna framework in classifying methods. We distinguished eight different categories:

1. Ordered Method Calls. Often, there is a distinct relationship between methods, implicit in source code. For example, method `willChange` is often followed by a call to `changed` when occurring in the same caller in JHotDraw. In other cases, certain methods are always called from the beginning or end of callers. Calls that are always called relative to other code segments can be easily transformed into aspects.

2. Contract Enforcement. As discussed in [15], methods that perform policy checking, signaling the compliance or lack of compliance with a given policy, are better represented as aspects. These "enforcing" methods are often called near the beginning of a method, usually as a conditional.

3. Complex Getter. Usually, an adapter method is essentially a getter, in which the returned data structure undergoes some processing or transformation before returning. Adapter methods are good candidates for aspects because they all often share very similar processing (e.g., transforming a list to an iterator). This common processing should be supported in one location, not in many.

4. Event Triggering/Handling without Event Parameter. Many methods that are called relative to another method do so because one method triggers an event, while the other method handles the event. This relationship between methods is similar to Category 1, but actual method calls can appear un-related in source code because of common coding patterns (i.e., Chain of Responsibility [7]). This category includes both the event triggering and the handling methods, in which the methods do not have an Event object (or sub-class) as a parameter. We distinguish this category from the next event handling category, by the non-existence of an explicit event parameter.

5. Singletons and **6. Factories.** Both are GoF design patterns with a fairly standard syntactic representation[7]. A study has shown that Singletons are well modularized in AspectJ, and that Factories' implementation can be slightly improved [11].

7. Event Triggering/Handling with Event Parameter. This category contains event triggering and handling methods which syntactically have events (or a subclass thereof) as a parameter. This distinction makes this category of methods easier to find, and distinct from, the previous event handling category.

8. Adding/Removing Listeners. Related to events and event handling, yet syntactically and semantically different are the adding and removing of listeners. Adding and removing listeners are part of the Observer pattern, which is well modularized by AOP [11].

The output of the manual tagging is a Classification Table. The Classification Table consists of an entry for each method that comprises the training program set, and its manually determined classification.

2.1.2 Mining Analyses Component

The Classification Table is input to the mining analyses component, which consists of an implementation of each of the single analyses to be combined. Each mining analysis generates an attribute for each method. The output of the Mining Analyses step is an Augmented Classification Table, in which each method has an associated set of attributes as well as the manual classification. Each row in the Augmented Classification table (method name, attribute vector, and classification) is considered as an *example*, because it contains information about a real program's method (attributes), and that method's classification. Examples are input to the machine learning algorithm.

Attributes may be boolean, numeric, or percentage values. For instance, the Fan-in analysis produces a numeric value (i.e., the degree of the call graph fan-in of that method). Table 1 shows an Augmented Classification Table. According to the first row, the method named `findCar` is called in five different places throughout the program, does not return a variable (is a void method), and 10% of its method body is a clone. `findCar` has been tagged as a concern.

Method	Fan-in	is-Void	Code Clone	Class
findCar	5	true	10%	true
paint	1	false	0%	false
crash	5	true	0%	true
gasUp	5	false	60%	false
cruise	1	true	30%	false

Table 1: Augmented Classification Table

2.1.3 Training through Machine Learning

The machine learning step derives generalities over thousands of training examples (method, attributes, classification) from the Augmented Classification Table, and outputs rules which can be used to classify nodes in the classification phase. Machine learning enables a large training set of data to be processed in an automatic way, and allows the strengths of multiple mining analyses to be combined for aspect mining in a single application, in an automatic, yet uncomplicated way.

The machine learning step generates propositional rules over the set of attributes. An example rule is:

If $(Fan-in > 4)$ and $(is-Void = true)$,
then $(Classification=true)$
(where true denotes "is a candidate").

This rule combines the results of two analyses, Fan-in and is-Void. If this rule is used to classify `findCar` from Table 1, then `findCar` is classified as a candidate, since its value for $Fan-in > 4$ and its value for $is-Void = true$. This rule cannot classify `paint`, because paint does not meet the condition $Fan-in > 4$.

We use the Incremental Reduced Error Pruning rule-learning algorithm (RIPPER) to learn from examples. It is particularly suited for the task of generating rules because it generates human readable rules (*if(conjunction) then classification*) and because it is able to deal with disjunct rules well. Another advantage of RIPPER is that it is able to efficiently deal with large, noisy data sets (up to hundreds of thousands of examples) [5].

2.2 Code Classifying Phase

Figure 3 shows the online code classification phase of aspect mining. A Classification Table for the application to be mined is initialized with each method name, and input to the analyses component. The analyses component performs each individual mining analysis, creates and augments the Classification Table with attributes, but with no classifications as yet. The classifier then takes the Augmented Classification Table and the set of classification rules generated by the training phase as input, and generates the Completed Classification Table for the application being mined. The classifier applies rules to the Augmented Classification Table in order. It classifies all examples it can with the first rule, and removes classified examples, then it repeats with the next rule, until all examples are classified or there are no more rules to apply.

When Table 1 was used as input to the machine learning step, it learned the rules:

1. if$(Fan-in >= 5)$ and (is-Void=true)
then $Classification = true$,

2. if(), then $Classification = false$

In order to classify Table 1 with this rule-set, we start with the first rule, and the working set initialized to all examples. The first rule classifies examples one and three in Table 1, so we note their classification (in this case true) and we remove them from the working set, leaving examples two, four, and five. These examples are classified by the second rule (all are classified as false), and the classifying phase is complete.

2.3 Time and Space Cost

Time. The training phase consists of the time to perform each single mining analysis and the machine learning time. While the analyses phase is dependent on the kinds of analyses, the machine learning time is dependent on the number of examples. RIPPER, the machine learning algorithm we used, has a faster running time than most rule-based learners, and while formal analysis of RIPPER is not straight-forward, it has processed 500,000 examples in 61 CPU minutes [5]. Our training program offers approximately 3,000 examples.

The classifying time is dominated by the analyses phase, which is dependent on the kinds of mining analyses applied.

Space. The space requirements of the training phase are dominated by the Augmented Classification Table. RIPPER does not require any data structures larger than this table. The Augmented Classification Table's size is determined by the number of methods in the training program set and the number of mining analyses. The space required is the same during both the classifying phase and the training phase.

2.4 Discussion of Tradeoffs

The Timna approach has the following advantages:

- Timna uses the results of several analyses to make a classification. This makes it more general, and helps avoid the mistakes of any single analysis.

- Timna learns off-line, once, and then applies its knowledge multiple times without expending any more time learning. This leads to a quick classification.

- Timna generates rules by analyzing real programs, so its rules are likely to be practically applicable.

- Timna can classify a program with partially generated attributes (rules often only test a subset of the overall attribute set), allowing for a more fault-tolerant, on-demand (before all analyses have completed) classification.

Timna is designed to provide a more precise classification of methods than a single analysis, but there are tradeoffs for this improvement. Running several analyses is always more expensive than running only one analysis. Timna also requires, during its training phase, a tagged input program set, which is expensive to construct. However, if good training data is provided once, Timna can then perform classification many times.

3. MINING ANALYSES

The Timna framework is extensible such that new and existing mining analyses can be incorporated without changing the current implementation. Our current implementation of a Timna mining tool includes both analyses that have been developed previously (by us and others) as well as new and extended analyses that we discovered in our initial experiences with Timna and during manual tagging of our subject training program.

3.1 Existing Analyses

Fan-in. Marin, et al. [15] developed a technique for identifying aspects based on the fan-in property of a method in a program's call graph. With each node in a program's call graph representing a method, the edges directed from a given node n representing method M lead to the nodes representing each possible callee from any call-site in M. Similarly, the node n will have edges leading into n from every possible direct caller of method M. The fan-in of a node n in a call graph is the number of these incoming edges of n. Marin et.al. showed that high fan-in values for a given node n represented by method M can mean refactoring of method M is needed [15]. Their intuition in creating this analysis is that cross-cutting functionality, when located in a method, will have a high fan-in, because it is used (called from locations) across the code base. In Timna, the attribute returned from fan-in analysis for a given method M is the fan-in of the node representing M.

Code-Clone. The automatic identification of code clones has been investigated by several researchers as a promising aspect mining analysis [17, 3]. For aspect mining, code clones are defined to be lines of source files that are identical (or extremely similar) to lines of code in a different location within the same application. In the Timna aspect mining prototype, we use CCFinder[12], a token-based tool, to identify code clones. The Code-Clone analysis returns the percentage of a method M's body that is part of a clone as the attribute value for the method M. This percentage is not affected if a code segment is part of multiple clones or is part of a group of clones, but it is a simple measure, where each statement is either part of a clone or not, and the percentage of the statements in a method that are clones is reported.

3.2 New Analyses

Unique-Class-Fan-In. During manual tagging, we observed that the number of unique classes that call a given method can be useful in determining candidates. Some candidates identified manually as good candidates had only a few incoming edges (i.e., static calls to the method), but the incoming edges were from methods in several different classes. The method does not have enough incoming edges to reach a high fan-in value, so fan-in analysis would miss this candidate. However, the fact that it is called by several different classes makes it likely to be a candidate for refactoring. The attribute returned from Unique-Class-Fan-In analysis for a given method M is the the number of unique classes from

which method M is called.

Calls-In-Clones. If the calls to a method M are often part of clones, then it is likely that M is always called before or after a certain segment of code or particular method call. Manual tagging found that method M with this property was often a good candidate. The Calls-In-Clones analysis for a given method M returns the percentage of calls to M that are identified as part of any clone. Calls-In-Clones analysis will be more effective when a PDG-based clone detector is used, because it can detect intertwined clones. For our initial prototype, we did not have a PDG-based clone detector that could handle the size of our training program. However, CCFinder, which uses source code tokens to determine clones, provides a robust implementation and good approximation. We expect our results from Calls-In-Clones analysis to improve if CCFinder is replaced by a PDG-based tool.

Method-Call-Pairing. Manual tagging revealed that a method that is always called by the same method as its "opposite" method is often a good candidate as well as its "opposite" method. For example, if both `willChange()` and `changed()`, or `start()` and `stop()` are called within the same method, the opposite pairs of methods would be good candidates. Thus, Method-Call-Pairing analysis for a given method M reports the percentage of the callers of M that pair with, or are called from the same method as, the calls to M's known "opposite" method. If a method pairs with several other methods, the analysis reports the highest percentage of pairing.

Call-Placement. Calls at the entry or exit of a method, especially if they always occur there, are often candidates. Consistently placing a method call either at the entry or exit of a method means the programmer is probably trying to express a concern that is not modularized well in OOP. Call-Placement analysis for a given method M reports the percentage of calls to M that are located at method entry or exit points.

Presence-of-Qualifiers. Often qualifiers (such as static, public, or interface), although not useful alone, can help make a case for a candidate. If a method M is a "getter" (known through another analysis), M's classification is still unknown, but if it is also public and static, then M is most likely an instance of a singleton, which is a good candidate for refactoring. Presence-of-Qualifiers analysis for a given method M reports whether M has a particular qualifier or not.

Parameters. While mining manually, we observed that methods with no parameters are more likely to be candidates. Similarly, when a method M has an event as a parameter, M is likely to be a candidate, since this usually means the method handles an event. Thus, No-Parameter analysis for a given method M reports whether M has any parameters. Event-Parameter analysis for a given method M reports whether M contains an event as a parameter.

Program	NCLOC	Methods	Classes
JHotDraw	11,689	2,739	296
PetStore	9,437	1,448	244

Table 2: Characteristics of Subject Programs

Special-Method. Special kinds of methods, such as getters/setters, utility methods, toString methods, and constructors offer some insight into a method's candidacy. For instance, most researchers do not consider toString methods to be good candidates for refactoring. Special-Method analysis for a given method M reports whether M is in the set of methods considered to be special.

4. EXPERIMENTAL STUDY

The goal of our experimental study was to evaluate the cost-effectiveness of the Timna framework for aspect mining. Unlike other empirical case studies of aspect mining techniques, which describe in detail the kinds of candidates discovered by their tools, we focus here on the precision, recall, and costs of mining by our machine learning approach with automatically combining multiple analyses. Our hypothesis is that combining analyses through the Timna framework will achieve high precision and recall on code segments that are considered through laborious manual tagging to be good candidates for refactoring.

To this end, we compare the effectiveness of the rules generated by Timna against rules suggested by another strong, but singular, aspect mining technique. We apply the classifying phase to both the tagged training program and an untagged test program, in order to evaluate the relevancy of rules generated from a training program to other programs. We begin with the research questions this evaluation is designed to answer.

4.1 Research Questions

Question 1. Does the combination of analyses increase precision and/or recall beyond a single analysis?
Question 2. How effective are Timna's rules generated by learning from combined mining analyses on a training program when applied to other programs?
Question 3. Does categorizing candidates syntactically when tagging produce rules that perform better than tagging them with a boolean (candidate or not)?
Question 4. What is the associated time overhead in using the Timna framework?
Question 5. How can the rules that Timna generates direct further research and provide quick evaluation of an individual analysis?

4.2 Subject Programs

We chose JHotDraw 5.4b1[1] as our training program. JHotDraw is an open-source framework for building drawing programs. In particular, we used a large, sample application, built using the framework and included in its distribution. JHotDraw is an especially good candidate for a training program because it was designed and implemented intentionally to be an example of good

---
[1] http://www.jhotdraw.org/

6

object-oriented code, using the latest design patterns and coding standards. Therefore, there should not be many cases where code needs to be refactored in an object-oriented manner, but instead only cases where object-oriented solutions did not succeed.

For our alternate (non-training) application, we used Java PetStore Demo [2], a J2EE sample application developed by Sun Microsystems. PetStore is an application that allows customers to purchase goods via a browser. It is a good candidate not only because it is open-source and of a reasonable size, but also because other groups performing aspect mining have studied it, which allows us to compare our results. Both programs' size characteristics are detailed in Table 2.

4.3 Variables and Measures

The independent variables in our study are the applied mining techniques and the tagging method. The mining technique is either Timna or an approximation of fan-in[15] by Timna. When using Timna, we tag the training data in two different ways, called "All" and "Two". When tagging in the "All" style, we tag each method with an aspect category (defined earlier) or with null (i.e., not a candidate). When tagging in the "Two" style, we tag each method with a boolean, true if it is a candidate, and false if it is not.

The dependent variables are the precision, recall, categorization breakdown of mined candidates, and time costs. We measure precision as *precision for technique T = (number of good candidates identified by T) / (total number of candidates identified by T)*. Precision is measured for both the training program and the test program. If the number of candidates that should be identified is known, *recall for T = (number of good candidates identified by T) / (total known good candidates)*. Recall is much harder to measure when performing aspect mining, because, in order to calculate recall, we must know the number of candidates that should be identified in the test program. Thus, we measured recall for the tagged training program only. Categorization of mined candidates was measured as a percentage of the total mined candidates by *T* which were in a given category described in Section 2.

4.4 Framework and Methodology

Timna was built as an Eclipse plug-in in order to leverage the analysis tools that Eclipse provides, as well as to provide a framework that was easy to extend. We manually tagged the training program, JHotDraw, at the method level, using the categorization in Section 2.1.1. These categories were defined during tagging with JHotDraw. The generality of these categories for other applications has not yet been investigated. Training Timna with JHotDraw, we generated two Augmented Classification Tables: a table with boolean classifications, and a table with categorical classifications. Feeding these two classification tables separately into the machine learning algorithm, we generated two sets of classification rules.

To simulate fan-in analysis, the most promising exist-

[2]http://java.sun.com/blueprints/code/

ing individual analysis for comparison, we first manually constructed a rule, based on Marin's [15] fan-in mining analysis. The fan-in rule is: *If a method* M *is not a getter/setter, not* **toString**, *not a utility function (part of a utility class), and the fan-in > 10, then Classification = true*. The simulation of fan-in analysis involved manually examining all of the candidates that this rule returned, using only this rule to identify candidates.

We performed the code classification phase with all three sets of rules on JHotDraw (our training program) and then on PetStore. We recorded the number of candidates identified in each category.

4.5 Threats to Validity

The size of our training set and our test set are the largest threats to the validity of our experiment. The size of the sets are: training set size = 2,739 examples, test set size = 4,187 examples. We believe these set sizes are large enough to provide feedback on the effectiveness of our approach (while small enough to be feasible for an academic study). Another possible threat is that we currently only gather training examples from one program, and only two programs (one being the training program itself) are used to gather testing examples. We chose the training program because of its specific qualities (see Section 4.2), and we specifically chose the testing program because it is drastically different from the training program. In effect, we have made it harder for Timna to perform well, because it is trained on one domain, and tested on a different domain. If Timna is able to perform well under these conditions, it is likely that Timna's principles are sound.

5. RESULTS

5.1 Effectiveness - Precision and Recall

Table 3 presents the mined candidate set sizes, precision and recall for Fan-in and two variations of Timna mining analysis. Fan-in returned, by far, the smallest candidate set. Furthermore, the precision of the two variations of Timna was also better than Fan-in, almost twice as precise as Fan-in in both cases. The coupling of larger candidate set size and higher precision naturally leads to higher recall for Timna, and in this case, significantly higher recall. Thus, the precision and recall for at least the training program supports our hypothesis with regard to Question 1, that is, combining analyses in a rational manner can significantly improve aspect mining effectiveness.

Since PetStore has not been manually tagged, we focused on returned candidate set size and precision for this program, which we used as our non-training program. As indicated in Table 3, Fan-in was not able to identify any candidates in PetStore, while both versions of Timna found over 200 candidates, with very high precision in both cases. It was surprising to observe that the precision was in fact higher for the non-training program than the training program in both Timna variations. These findings indicate a very promising positive answer to Question 2, suggesting that the rules generated by Timna with the training program are quite

7

Program	Technique	Total Ret'd	Precision	Recall	Category Recall							
					1	2	3	4	5	6	7	8
JhotDraw	Tagged	966	–	–	351	129	27	107	7	31	248	66
	Fan-in	38	36.84%	1.45%	2.0%	1.6%	11.1%	0%	14.3%	0%	0%	1.5%
	Timna:All	327	67.28%	22.77%	42.7%	4.7%	0%	14.1%	0%	0%	16.5%	12.1%
	Timna:Two	940	62.02%	60.35%	59.1%	1.6%	0%	73.8%	85.7%	0%	93.5%	86.4%
PetStore	Fan-in	0	0%									
	Timna:All	258	93.80%									
	Timna:Two	464	81.03%									

Table 3: Precision and Recall
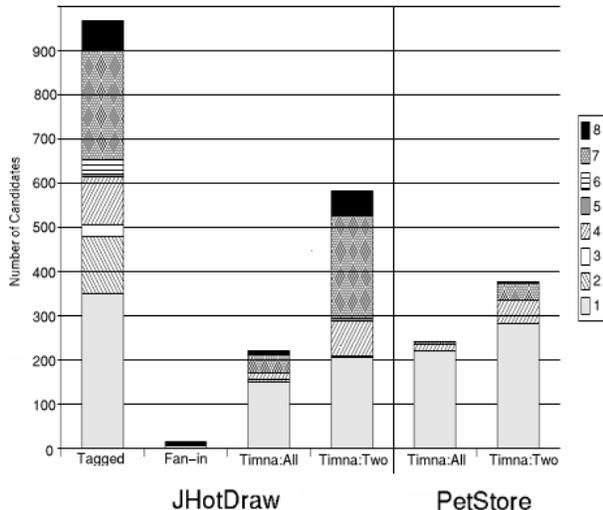
Figure 6: Category Distribution of Mined Candidates

Program	Technique	Time Costs		
		Analyze	Learn	Classify
JHotDraw	Timna:All	5m28s	6.24s	0.45s
	Timna:Two	5m28s	1.88s	0.65s
PetStore	Timna:All	2m04s	–	0.32s
	Timna:Two	2m04s	–	0.47s

Table 4: Timna Time Cost Measurements

relevant and useful across programs, not just on the training program.

The comparison of the results for precision and recall using Timna:All versus Timna:Two shows that Timna:All provides at least 5% higher precision than Timna:Two for both the training program and the non-training program. However, Timna:All returned a smaller candidate set than Timna:Two for both programs, and provided much lower recall for the training program. Thus, our experiments do not yet provide a clear answer to Question 3. More results on recall are needed to understand the discrepancy between the higher precision with lower sized sets, against the lower recall.

5.2 Mining Distribution over Categories

Table 3 indicates the recall within each of the eight categories of candidates described earlier, for our training program, JHotDraw. The table shows that Timna:Two has the highest recall for five of the eight categories, and that Timna:Two is able to achieve over 85% recall for three categories.

Figure 6 graphically depicts the distribution among the categories that were manually tagged and candidates that were identified in different categories by each mining analysis on the JHotDraw and PetStore programs. The graph clearly shows that category 1 is the most frequently occurring mined candidate and that both Timna variations perform relatively well at identifying candidates in this category. Timna does not perform well at detecting candidates in categories two, three, four, and six. Our data shows that Fan-in performs poorly in most categories, considering that it finds only 14 candidates in total for JHotDraw and no candidates in PetStore.

5.3 Time Costs for Timna

As shown in Table 4, analyzing the programs is the most expensive task when running Timna. Most of these analyses can be performed incrementally, and a production implementation of Timna would perform these analyses during idle processor time. With our prototype implementation and 18 analyses, the reported times are not unreasonable for the size of our subject programs. The learning and classification times are both small. We did not measure space costs, as we do not believe our prototype could provide a fair assessment of these costs. We presented a theoretical analysis of costs in Section 2.3.

5.4 Analysis of the Generated Rules

We examined the rules generated by Timna to get a better understanding of the relationship between each mining analysis and a method's classification. Intermediate sets of rules, generated as we were deciding which analyses to include, also assisted us in understanding which analyses complemented or subsumed other analyses. Here, we present a brief overview of some of our observations, targeted at answering Question 5.

5.4.1 Categorical Rules

By separating all aspects into syntactic categories, the sequential learning algorithm was able to generate rules specific to the identification of the different categories. We believe that these rules should have high precision across domains, because they are specifically tuned to discover a syntactic category. We present the categorical rules here:

$(No-Parameters = n)$ and $(Num-Of-Classes-Called-$ $From >= 2)$ and $(Percent-Calls-At-Beginning-Or-$ $End >= 100)$ and $(Is-Setter = n) \Rightarrow Classification = 8$
$(Is-Void = y)$ and $(Is-Constructor = n)$ and $(Is-$ $Setter = n)$ and $(Percent-Calls-At-Beginning-$ $Or-End >= 50)$ and $(Is-Public = y)$ and $(Highest-$ $Pairing >= 50)$ and $(Num-Of-Classes-Called-$ $From >= 2)$ $\Rightarrow Classification = 4$
$(Is-Void = y)$ and $(Event-Parameter = y)$ and $(Highest-Pairing <= 0)$ $\Rightarrow Classification = 7$
$(Is-Void = y)$ and $(No-Parameters = y)$ and $(Is-$ $Constructor = n)$ and $(Fan-In >= 3)$ $\Rightarrow$ Classification=1
$(Is-Void = y)$ and $(No-Parameters = y)$ and $(Is-$ $Constructor = n)$ and $(Percent-Of-Method-Body-$ $That-Is-Clone <= 0)$ $\Rightarrow Classification = 1$
$\Rightarrow Classification = -1$

The most interesting trait about these rules is the grouping of different analyses. In the first rule, we see that it is important for this kind of aspect to have parameters, be called from several places, not be a setter, and have all of its calls at the beginning or end of a method. Intuitively, adding and removing listeners (category 8) often have these traits. However, if we had used intuition to generate these rules, we might have included other conditions, such as the method cannot be a Getter. We would also have difficulty producing the constants (i.e., $AnalysisResult < 50$), because of the volume of data. Finally, these rules are ordered to maximize performance. The ordering of rules gives us insight into each rule's importance.

5.4.2 Boolean Rules

Boolean rules, as we expected, are more general than the rules generated by the categorized input. While slightly less precise, they provide significantly higher recall, and perhaps give us insight into constructs that we should eliminate from the language. Below we present the boolean rules:

$(Is-Void = y)$ and $(Is-Constructor = n)$ and $(Is-$ $Setter = n)$ and $(Fan-In <= 0)$ and $(Percent-Of-$ $Method-Body-That-Is-Clone <= 22)$ and $(Event-$ $Parameter = y)$ $\Rightarrow Classification = 1$
$(Is-Void = y)$ and $(Is-Constructor = n)$ and $(Is-$ $Setter = n)$ and $(Is-Public = y)$ and $(Percent-Of-$ $Method-Body-That-Is-Clone <= 14)$ $\Rightarrow Classification = 1$
$\Rightarrow Classification = -1$

The above rules both require that the percent of a method's body that is a clone be low, in order to classify the method as a scattered concern. Our initial intuition was that if a method included cloned code, then it was more likely to be a part of a scattered concern. However, by learning from actual code, we see that in this context (where the other conditions of the rules hold), more cloned code actually represents a decline in the likelihood of being a scattered concern. This demonstrates a case where an individual analysis's results would produce an incorrect classification when taken alone, but are part of a correct decision when combined with others. We suspect that many code clones are comprised of method calls to methods that need refactoring into

the AOP paradigm. Clones made of calls can create a method that clones, but does not need refactoring.

5.5 Summary of Results

Our results show that Timna performs much better than Fan-in, a leading (individual) aspect mining analysis, according to precision, for both the training program and the test program, and in recall for the training program. Tagging the training data with categories generates rules that perform with slightly higher precision, but less recall. This increased effectiveness comes with very little time overhead. The rules of both the categorized training data and the boolean training data help the researcher understand which characteristics cause a method to be a candidate.

6. RELATED WORK

In early aspect mining work, two types of tools were developed - lexical tools and exploratory tools. Both help the user find candidates, but insist that the user make the final decision over classification, and so we call them non-classifying tools.

6.1 Non-Classifying Tools

Lexical search-based tools, such as AMT [10] and the Aspect Browser [9], were designed to leverage the power of a lexical search. They are particularly good at visualizing the scattering of certain types of concerns, as they can display the results of their string searches as highlighted lines on a source-code model. Every line that contains the string in question is highlighted on rectangles that represent source files. If a particular search highlights lines in many different places, it may indicate a scattered concern. AMT also includes searching by type, where type-based searching can be used to increase confidence in conclusions drawn from a lexical search [20].

Exploratory tools are less automatic than lexical search tools. They allow the user, starting with a seed, to navigate a program via structural queries. These navigation tools, like JQuery [19] and FEAT [16], help the user navigate the program, but the user has to make his own conclusions about the code. Since programs often have large call graphs, sometimes a structural exploration path can become quite long. Exploration tools are most useful when used in conjunction with a seed identifier, like Timna, because seed identifiers provide a starting set of seeds. A robust starting set can help the user avoid long exploration paths. Most relevant code should be only a step or two from any given node in a robust starting set.

6.2 Single Classifying Analyses

Researchers have investigated several individual mining analyses. These analyses are largely focused on the identification of seeds. 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.

[17] used PDG-based clone detection to discover seeds with very high precision. Magiel and van Deursen [15] 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 [3]. Silvia et al. [2] 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. [18] 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. [15] introduces a strong analysis. Their automated tool produces reasonably precise results, which they then refined with manual filtering. They provide a thorough discussion of the candidates that they found.

6.3 Comparison of Analyses

Researchers have also done work on the comparison of different mining techniques. In this work, they suggest different methods for combining techniques, after using three different mining techniques on the same program. Their findings support our use of machine learning to discover which analyses are important when classifying different types of candidates [4].

7. CONCLUSIONS AND FUTURE WORK

Our study suggests that, by combining analyses, we can identify seeds with increased precision and recall over a single analysis used in isolation. Timna generates rules by analyzing real programs, so its rules are likely to be practically applicable. Timna generates a human-readable set of rules that help us reason about seeds and understand which analyses are important. These rules appear to be applicable across programs, as they were generated from a training program, then applied to the training program and another program with approximately equal success. Because the recall of this method was much higher than from a single analysis, less exploratory work will need to be performed to find all of the concern.

In the future, we plan to refine the tagging of training programs, and to provide other training programs. These are important to have in order to evaluate any aspect mining analysis, and to further evaluate the Timna framework. We intend to continue developing individual analyses, because the quality of individual analyses influences the results of our overall framework.

Acknowledgments

8. REFERENCES

[1] AspectJ Homepage, <http://www.aspectj.org>. 2005. (May 26, 2005).

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

[3] 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.

[4] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwe. A qualitative comparison of three aspect mining techniques. In *Int. Wkshp. on Program Comprehension*, 2005.

[5] W. W. Cohen. Fast effective rule induction. In *Int. Conf. on Machine Learning*, 1995.

[6] M. Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley Longman Publishing, 1999.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[8] J. D. Gradecki, N. Lesiecki, and J. Gradecki. *Mastering AspectJ.* John Wiley & Sons, Inc., New York, NY, USA, 2003.

[9] 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.

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

[11] J. Hannemann and G. Kiczales. Design pattern implementation in java and aspectj. In *Object Oriented Programming, Systems, Languages and Applications*, 2002.

[12] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 2002.

[13] A. Kellens and K. Gybels. Issues in performing and automating the extract method calls refactoring. In *Software Eng. Properties of Lang. for Aspect Technology, Wkshp. at AOSD*, 2005.

[14] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming.* Manning Publications Co., Greenwich, CT, USA, 2003.

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

[16] M. P. Robillard and G. C. Murphy. Concern graphs. In *Int. Conf. on Software Eng.*, 2002.

[17] 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.

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

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

[20] C. Zhang, G. Gao, and A. Jacobsen. Amtex, <www.eecg.utoronto.ca/ czhang/amtex>. (October 18, 2003).