Case Study: Supplementing Program Analysis with Natural Language Analysis to Improve a Reverse Engineering Task

David Shepherd, Lori Pollock, and K. Vijay-Shanker

Computer and Information Sciences
University of Delaware
Newark, Delaware 19716
{shepherd, pollock, vijay}@cis.udel.edu

Abstract

Software maintainers often use reverse engineering tools to aid in the extremely difficult task of understanding unfamiliar code, especially within large, complex software systems. While traditional program analysis can provide detailed information for reverse engineering, often this information is not sufficient to assist the user with high-level program understanding tasks. To bridge the gap between current reverse engineering tools and the high-level questions that software maintainers want answered, we propose supplementing traditional program analysis with natural language analysis of program source code. This paper presents a case study where we have augmented an existing reverse engineering tool, an aspect miner, to complement the existing traditional program-analysis-based miner with natural language analysis of method names, class names, and comments. Our quantitative and qualitative results strongly suggest that supplementing traditional program analysis with natural language analysis is a promising approach to raising the level of effectiveness of reverse engineering tools.

Categories and Subject Descriptors D.2.6 [*Software Engineering*]: Programming Environments: Programmer workbench; D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement: Restructuring, Reverse engineering, and Reengineering

General Terms Languages, Algorithms, Reliability

Keywords Program Analysis, Natural Language, Software Tools, Aspect Mining

1. Introduction

A program's comprehensibility often decays over time, due to software maintenance and evolution [13]. In a large project, with millions of lines of code, even maintainers familiar with the software have to comprehend or re-comprehend an extensive amount of unfamiliar source code daily to fix bugs and add features. Unless developers maintain the code's comprehensibility by refactoring, daily tasks become increasingly difficult [7].

Maintainers use many reverse engineering tools to quickly understand a software system so they can make appropriate changes or refactorings [6]. During a maintenance task, developers often ask high-level questions such as *Which class represents concept X?* or *Where is there any code involved in the implementation of behavior Y?* These questions can be difficult to answer using traditional program-analysis-driven reverse engineering tools [23]. A recent survey concludes:

> ...[traditional program analyses] have been successful at treating the software at the syntactic level to address specific information needs and to span relatively narrow information gaps." [18]

This judgment may be overly harsh, yet correctly identifies an issue that researchers should address: spanning the gap between users' high-level tasks and program analyses' detailed information.

We propose using natural language information to bridge this gap. We supplement traditional program analyses (**TPA**) (e.g., call graph analysis, control flow analysis, data flow analysis, type analysis, etc.) with natural language analyses (**NLA**) applied to a program's source code. We hypothesize that NLA complements TPA because NLA analyzes different program information than TPA.

In this paper, we investigate our hypothesis through a case study. In our case study, we add NLA to an existing TPA-based-framework, Timna, to improve its effectiveness at a reverse engineering task, aspect mining. Aspect mining is the process of identifying code segments in an object-oriented program that developers could better modularize into an aspect-oriented program (AOP) using an aspect-oriented language. Aspect mining is the first step towards refactoring a system from OOP to AOP. We selected aspect mining as our representative reverse engineering task because (1) there is no sufficiently effective solution for aspect mining and (2) researchers have only used TPA for aspect mining so we are able to investigate how much NLA can assist in this task. In this paper, we make the following contributions:

- A case study of supplementing TPA with NLA to improve a reverse engineering task
- Development of NLA targeted toward aspect mining
- Implementation of an extended aspect mining framework, iTimna, to exploit the integration of NLA and TPA
- An initial quantitative and qualitative evaluation of combining TPA and NLA for aspect mining

Section 2 describes the necessary background on aspect mining. In Section 3, we present the NLA-based features we developed for aspect mining. Section 4 presents our evaluation methodology. Section 5 presents both quantitative and qualitative results. Section 6 describes related work, and Section 7 presents our conclusions.

PASTE'07 June 13–14, 2007, San Diego, California, USA.

2. Background

To modify an application, developers must identify the high-level idea, or ***concept***, to be changed and then locate, comprehend, and modify the concept's ***concern***, or implementation, in the code [15]. The research community agrees that object-oriented programming causes certain concerns to become scattered [11, 24]. These concerns are known as crosscutting concerns. *Aspect-Oriented Programming*, or AOP, is a language paradigm created to better modularize crosscutting concerns [11]. Legacy codes are refactored into AOP by first identifying the *refactoring candidates* (code segments developers could better modularize in an AOP language) through the process of aspect mining and then performing the actual program modularization into AOP style. For this paper, we assume that refactoring candidates are at the method granularity, although, in general, any code snippet can be a refactoring candidate.

A good refactoring candidate is a method that is part of a crosscutting concern. For instance, a tracing function that is called by many different methods in many different classes is part of a crosscutting concern, called the tracing concern. The source code would be more modular if the tracing information was all in one module, i.e., an aspect, instead of scattering calls to the tracing function throughout the program. Researchers have identified several categories of methods that are good refactoring candidates, such as methods in the observer design pattern, methods that are called side-by-side several times, event triggering methods, contract enforcement methods, factory and singleton methods, complex getter methods, and persistence methods (see Section 4.2) [20].

There are many straightforward TPA clues that researchers have used to perform aspect mining (see Section 6 for a thorough discussion). Because aspects were made to remodularize concerns that are not well modularized by objects, certain code smells can be used to find these concerns. For instance, researchers have realized that if a method is called by many different methods, it is often a good refactoring candidate [17]. TPA can determine if a method is called by many different methods, by analyzing the call graph. Researchers have also used code clone detection, which is based on the control flow graph, to identify refactoring candidates [21].

Prior to our work on Timna [20], existing approaches to automating the aspect mining process focused on developing aspect mining analyses based on a single TPA characteristic. Each analysis typically found only a subset of possible refactoring candidates and was unlikely to find candidates which humans find by combining analyses. We designed Timna as a framework for enabling the automatic combination of aspect mining analyses [20]. The key insight was the use of machine learning to learn when to refactor, from vetted examples (i.e., supervised learning). The results of our experimental comparison of Timna to Fan-in, a leading aspect mining analysis [17], indicated that such a framework for automatically combining analyses is very promising and extensible.

In addition to call graph and code clone analysis, Timna uses TPA information that can be extracted from the abstract syntax tree of a program, such as whether a given method returns void, or whether it has parameters [20]. Because Timna is a machine learning framework each TPA used in Timna must output its result as a feature. Timna uses TPA to create a feature vector for each method. For example, an abbreviated vector (missing a few TPA features) for method AbstractTool.isEnabled() looks like:

Method Name	Fan-In	Returns Void	Is Constructor	Is Refactoring Candidate
isEnabled	4	F	F	T

where *Method* is a label, {*Fan-In*, *Returns Void*, *Is Constructor*} are input features, and *Is Refactoring Candidate* is the output attribute. Timna combines these TPA clues to accurately identify good refactoring candidates. For example, Timna previously discovered that if a method returns a value, has no parameters, is not a constructor, and is called from more than three methods, then it is likely a good refactoring candidate [20]. This rule is able to find enforcement methods such as public boolean AbstractTool.isEnabled() in JHotDraw by combining several TPA.

While Timna is a state-of-the-art aspect mining framework, it was only able to identify refactoring candidates with 67% accuracy on the training data in our experiments. After analyzing the methods that Timna was having difficulty identifying, we realized that there were a large number of natural language clues in the program code that we were not leveraging. The remainder of this paper describes how we improved both the precision and recall of Timna by developing NLA-based features and adding them to Timna to create integrated NLA-TPA Timna, iTimna.

3. Integrating Natural Language Analyses

In previous work, we have investigated how to extract useful natural language clues from a program's source code, developed a program model to represent the natural language relationships, and designed and implemented a search tool based on our natural language representation [19, 22]. This paper goes beyond this work by investigating the combined use of NLA and TPA within a single tool, for a new application. When using natural language clues extracted from a program to help with searching for crosscutting concerns, we discovered that *verbs* are extremely important [19]. We believe that this is because in *object*-oriented programs, the nouns (i.e., objects) are well modularized, causing some verbs to become scattered. Therefore, in an OOP system, the verbs sometimes represent crosscutting concerns. We have previously created techniques for extracting verbs from method names and comments which we use in several NLA-based tools as well as the following analyses [22]. This section describes the five distinct NLA that we added to Timna to create iTimna. For each NLA, we present motivation, supporting examples, and how the analysis is performed. Each NLA returns a boolean value which iTimna uses as a feature input during its machine learning phase.

Opposite Verbs When two methods in the same class perform opposite functions, they are often good refactoring candidates. Consider the methods lock() and unlock(), called before and after an operation occurs. We could refactor lock() and unlock() into before and after advice. Our experience and other aspect mining research [1, 3, 16] supports our hypothesis that methods with opposite verbs are good refactoring candidates. It is often difficult for TPA to detect opposite methods because they can be far apart in both the call graph and the stack trace.

The opposite verb analysis must determine, given a single method m in class c, if method o (m's opposite) occurs in c. The iTimna framework uses NLA [22] to find and extract the verb v from m's identifier. For lockFile(), iTimna would extract *lock*. Similarly, iTimna extracts the verbs from all other methods in c. If any method besides m uses v's antonym (e.g., the verbs *lock* and *unlock*), the analysis returns true. Similarly, if any method besides m uses v with the opposite preceding word (e.g., beginEdit() vs. endEdit()), then the analysis returns true.

Past-Tense Verb The Java Language Specification states "method names should be verbs or verb phrases...". Thus, developers often use verbs in method names. A developer who wants to create a method that "opens a file" might call it openFile(). That developer will not call the method openedFile(), because the verb's tense conflicts with his intent. Using this same principle, when a developer uses a past-tense verb in a method name, he is declaring that the method is a reactionary method. For instance, a method called figureSelectionChanged() does not itself *change the selection*; it only reacts, as another method has already *changed the se-*

lection. Similarly, the method mousePressed() reacts to the mouse button being pressed. These reactionary methods (similar to "update" methods in the observer pattern) are good refactoring candidates [10, 20].

The past-tense verb analysis must determine, given a single method m, if m uses a past-tense verb. Similar to opposite verb analysis, iTimna uses NLA to find and extract the verb v from m's identifier. For figureSelectionChanged(), iTimna would extract *changed*. Then iTimna uses simple morphological analysis to determine v's tense. If v is past-tense, the analysis returns true.

Observer Verbs Researchers agree that an implementation of the observer design pattern contains several methods that are good refactoring candidates, including the *notification*, *update*, *attach*, and *detach* methods [10, 20]. Developers use a limited vocabulary when implementing the observer design pattern. In method fireViewSelectionChangedEvent() of JHotDraw, the developer used several different words to note that fireViewSelectionChangedEvent() is a notification method, including the verb *fire* in the method name and the verbs *notified* and *changed* in the comments. Similarly, the developer who authored keyPressed() used clear comments to note that keyPressed() is an update (i.e., handler) method, with the inserted comment, "Handles key down events". Finally, attaching and detaching methods, such as method addListeners(), are also good refactoring candidates. We have created an NLA-based analysis that detects these observer-pattern verbs and their synonyms in code.

The observer-pattern verb analysis must determine, given a single method m, if m or m's set of comments com use any of the observer-pattern verbs. iTimna uses the techniques in [22] to find and extract the verb v from m's identifier. For keyPressed(), iTimna would extract *pressed*. iTimna also uses part-of-speech tagging to identify and extract all verbs in com. iTimna checks if any verbs that it has found either match or are synonyms of verbs that describe the observer design pattern. If so, then the observer-pattern verb analysis returns true. When identifying attaching and detaching verbs, iTimna must also ensure that the direct object of the verb is a *listener* or an *observer*, to avoid methods that use the verb *add* with an unrelated direct object (e.g., addPopupMenuItems()).

Constraint Verbs Researchers agree that constraints, or contract enforcement methods, are good refactoring candidates [16, 20]. A constraint method checks a condition to determine whether to execute an operation. For instance, AbstractTool.isEnabled() is checked before a user can select a tool for use. This constraint method uses the verb *is*, and other constraint methods use verbs such as *check*Enabled() or *can*Connect().

The constraint verb analysis must determine, given a single method m, if m or m's comment set com use any of the constraint verbs. iTimna uses the techniques in [22] to find and extract the verb v from m's identifier. For isEnabled(), iTimna would extract *is*. iTimna also uses part-of-speech tagging to identify and extract all verbs in com. iTimna checks if any verbs that it has found either match or are synonyms of constraint verbs, such as *check*, *is*, or *can*. If so, then the constraint verb analysis returns true.

Factory Terms Factories are one of the design patterns that researchers have shown to be better implemented as an aspect [10]. Therefore, we attempt to identify refactoring candidates such as createUndoActivity() which has the comment "Factory method for undo activity". We currently search for the term *factory* in the comments and identifiers of the code.

The factory term analysis must determine, given a single method m, if m or m's comment set com use any factory terms. iTimna uses previous techniques [22] to find and extract all the terms v from m's identifier and comments. For createUndoActivity(), iTimna would extract *create*, *undo*, and *activity*. iTimna would also extract each word in the comment "Factory method for undo activity". iTimna checks if any terms that it has found either match or are synonyms of factory terms. If so, then the factory term analysis returns true.

4. Case Study Evaluation

We conducted a case study to evaluate iTimna's improvement over Timna due to the combination of NLA and TPA. In particular, we designed the case study to provide some insight into the question: *What effect does integrating NLA with the TPA of our state-of-the-art aspect mining framework, Timna, have on the aspect miner's effectiveness in terms of precision and recall?*

4.1 Subject Program

We chose JHotDraw 5.4b1[1] as our subject application. JHotDraw is an open-source framework for building drawing programs with approximately 22,000 non-commented lines of code and is relatively well-documented. JHotDraw is an especially good candidate for a training program because it was designed and implemented intentionally to be an example of good 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.

4.2 Training Data

Our machine learning approach requires training data, so we manually marked refactoring candidates in JHotDraw. We marked a method as a refactoring candidate only if it is similar to a well-known example in the AspectJ literature [10, 11, 12]. For instance, we tagged the method changed() in Figure 1 because it is much like the notify() method in the observer design pattern, which researchers have shown are good refactoring candidates [10]. As we manually tagged the program in our study, we observed that we were tagging several distinct syntactic categories of methods, which we tagged differently so we could study the ability of Timna and iTimna to identify different categories of refactoring candidates. We distinguished eight categories, which we briefly describe below (with a more thorough description in [20]).

1. Ordered Method Calls. Often developers call methods, such as willChange() and changed(), from the same caller method, always in the same order.

2. Contract Enforcement. These methods perform policy checking, such as isEnabled(), often at the entry point of a method.

3. Complex Getter. Usually, an adapter method is essentially a getter, in which the returned data structure undergoes some processing or transformation before returning.

4. Event Triggering/Handling without Event Parameter. These methods either trigger or handle an event, yet do not have a parameter that is an event.

5. Singletons and **6. Factories.** Both are GoF design patterns with a fairly standard syntactic representation[8].

7. Event Triggering/Handling with Event Parameter. Similar to Category 4, except these methods have an event 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.

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

```
1 /** Sets the arc's width and height. */
2 public void setArc(int width, int height)
3 {       willChange();
4         fArcWidth = width; fArcHeight = height;
5         changed();      }
```

Figure 1. willChange() and changed()'s consistent usage pattern.

4.3 Variables and Measures

The independent variables in our study are the applied mining techniques and the tagging strategy. The mining technique is either Timna or iTimna (i.e., Timna with NLA). When using Timna or iTimna, we tag the training data in two different ways, called "All" and "Boolean". 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 "Boolean" 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 overall precision, overall recall, and recall of refactoring candidate categories. Precision provides intuition about the quality of a result set. We measure precision as *precision for technique T = (number of good candidates identified by T) / (total number of candidates identified by T)*. Recall provides intuition about the completeness of the result set. We measure recall as *recall for T = (number of good candidates identified by T) / (total known good candidates)*. Categorization of mined candidates was measured as a percentage of the total candidates mined by *T* which were in a given category described in Section 4.2.

4.4 Framework and Methodology

Timna and iTimna were built as an Eclipse plug-in in order to leverage Eclipse's program analysis APIs, as well as to provide a framework that is easy to extend. First, we manually tagged the subject application, then we used Timna and iTimna to generate a feature-vector for each method, where each item in a vector is the result for a single analysis. Timna and iTimna used the Weka Machine Learning Toolkit to train and test on this data, using ten-fold cross-validation [26]. Within Weka, we used a rule-learning algorithm, which creates a set of human-readable classifying rules as part of its output. After calculating the overall precision and recall, we also calculated the recall for individual refactoring candidate categories.

4.5 Threats to Validity

We used 2,739 refactoring candidate examples (i.e., method declarations) to evaluate iTimna; we believe this is a sufficient amount of examples for a machine learning case study, but recognize that a full evaluation should include more examples. To reduce the risk of overfitting and to have enough examples to evaluate our, framework we perform ten-fold cross-validation. We also include qualitative evidence (examples) that iTimna discovered so that the reader can decide whether iTimna is finding reasonable refactoring candidates.

The first author tagged the training data used in this experiment during a previous experiment. We believe that the threat of bias is reduced because the author tagged the data two years prior to developing the current approach. We plan to further evaluate the framework with data tagged by another annotator.

Naming Conventions Our approach relies on reasonably standard naming conventions, assuming, for instance, that most method identifiers either contain a verb or strongly imply a particular verb. While no empirical studies exist that definitively support our assumptions, many naming convention studies, naming convention guidelines, and even source code examples suggest that our basic assumptions hold for most systems. In an empirical study of naming conventions researchers introduce a commonly used metaphorical convention, the METHODS ARE ACTIONS convention, where method identifiers are verb phrases [14]. In this study, researchers also introduced the DATA ARE THINGS and the TRUE/FALSE DATA ARE FACTUAL ASSERTIONS conventions. Both of these method naming conventions, while not explicitly requiring a verb, usually imply a verb (e.g., List.isEmpty → "*check* if the list is empty"). Another group has created a regular grammar to parse method identifiers where one major category is "Actions"; each example in this category contains a verb [4]. Many naming convention guidelines support our assumptions, as they encourage developers to use verbs or verb phrases for method identifiers [9]. Finally, almost all of the large number of source code examples in aspect mining literature are consistent with our assumptions [16, 17, 2, 1]

5. Results

5.1 Quantitative Results

Effectiveness - Precision and Recall Table 1 presents the precision and recall for Timna and iTimna where each was evaluated using all of the categories of aspects (Timna:All and iTimna:All) and then using only a boolean classification (Timna:Boolean and iTimna:Boolean). iTimna:All was dramatically more effective than Timna:All with 15.9% higher precision and 40.9% higher recall. However, because the Timna framework generally performs better in the boolean classification mode, the precision and recall values for Timna:Boolean and iTimna:Boolean represent the real improvement in adding NLA. In boolean mode, iTimna's precision is 19.2% higher and its recall is 12.2% higher than Timna. iTimna's increase in both values is significant, representing a 50.5% error-rate reduction and a 30.6% error-rate reduction, respectively. Timna:Boolean returned 940 candidates, of which only 583 were valid, whereas iTimna:Boolean returned 855 candidates, with 694 being valid.

Effectiveness by Categories In Table 1, we also report the recall for each aspect category. We do not report precision because it is impossible to calculate precision for Timna:Boolean and iTimna:Boolean (i.e., a boolean classification does not determine the category of refactoring candidates). By examining the categorical data, it can be seen that iTimna:All achieves dramatic overall increase in recall (40.5%) over Timna:All because *(1)* iTimna:All more effectively mines categories one, two, three, four, seven, and eight, and *(2)* iTimna:All improves recall on categories that appear often in JHotDraw (one, two, four, and seven). iTimna:Boolean also improves recall over Timna:Boolean in categories one, two, six, and seven, leading to better recall overall. iTimna:Boolean is less effective in categories four, five, and eight, but because these categories are either very small or the recall only drops slightly, iTimna:Boolean's recall still improves overall. For larger categories (one, two, and seven), iTimna:Boolean achieves improved recall.

Discussion It is usually difficult to improve the performance of a previously tuned machine learning system unless a researcher can add new, innovative features. We believe that the dramatic improvement that iTimna shows over Timna supports our hypothesis that NLA combined with TPA can significantly assist the reverse engineering task of aspect mining. Furthermore, the precision and recall results are now strong enough to warrant iTimna's use as a backend to an interactive aspect-refactoring tool.

5.2 Qualitative Results

Because NLA is often complementary to TPA, iTimna can identify refactoring candidates even when TPA provides virtually no clues or TPA provides only weak clues. Here we present a few examples where NLA was necessary to identify a refactoring candidate. iTimna uses a machine learning algorithm that outputs a ruleset, which iTimna then uses to classify new methods. For each example, we include the rule that iTimna used to identify it as a refactor-

Technique	Total Methods Returned	True Positives	Precision	Recall	Category Recall 1	2	3	4	5	6	7	8
Tagged	966	–	–	–	351	129	27	107	7	31	248	66
Timna:All	327	220	67.3%	22.8%	42.7%	4.7%	0.0%	14.1%	0.0%	0.0%	16.5%	12.1%
iTimna:All	713	593	83.2%	63.3%	49.9%	84.3%	42.9%	22.1%	0.0%	0.0%	93.9%	80.3%
Timna:Boolean	940	583	*62.0%*	*60.4%*	59.1%	1.6%	0.0%	73.8%	85.7%	0.0%	93.5%	86.4%
iTimna:Boolean	855	694	***81.2%***	***72.5%***	67.3%	85.0%	0.0%	50.0%	0.0%	35.5%	93.9%	81.8%

Table 1. Precision and Recall

```
1  public class TextTool...{
2  protected void beginEdit(TextHolder figure) {...}
3  protected void endEdit() {...}
4  /** If the pressed figure is a TextHolder it can be
5   * edited otherwise a new text figure is created. */
6  public void mouseDown(MouseEvent e, int x...){...
7      if ((textHolder != null)&&textHolder.acceptsTyping()){
8        // don't create new TextFigure, edit existing one
9        beginEdit(textHolder);
10 }...}...
11 /** Terminates the editing of a text figure. */
12 public void deactivate() {
13     endEdit();
14     super.deactivate();
15 }... }//end class TextTool
```

Figure 2. beginEdit() and endEdit()'s usage makes it difficult for TPA alone to detect their relationship.

ing candidate to illustrate how the TPA and NLA were often used in combination.

In JHotDraw, the class TextTool has two methods that occur before and after an edit occurs, beginEdit() and endEdit(). They are similar to the well-studied refactoring candidates willChange() and changed(), two methods that are always called before and after changing a Figure (shown in Figure 1). iTimna uses the following rule to identify endEdit() as a refactoring candidate:

$(\mathbf{Opposite-Verbs} = T)$ and $(Is-Void = T)$ and $(No-Parameter = T) \Rightarrow Classification = T$

iTimna used two TPA and one NLA to identify this candidate. It would have been difficult for Timna to identify beginEdit() without NLA because of the lack of conclusive TPA clues. For example, calls to beginEdit() and endEdit() do not ever occur from the same caller (see Figure 2 lines 9 and 13), and the methods appear approximately 240 methods apart in the stack trace. However, the natural language clues are strong—two methods in the same class are named begin$< verb >$ and end$< verb >$—and TPA with NLA provide enough evidence to correctly identify this candidate.

Constraints are known to be good refactoring candidates [17, 20]. It is often difficult to identify constraint methods using only TPA because the constraint method must be used many times in order to exhibit typical code smells, such as being called from many methods. However, NLA and TPA together can identify constraint methods regardless of their usage. iTimna created the rule:

$(\mathbf{Constraint-Verb} = T)$ and $(Is-Void = F) \Rightarrow Classification = T$

This rule uses one NLA and one TPA. In JHotDraw, even though isEnabled() (called in line 2 of Figure 3) is called only four times, iTimna correctly identifies it as a refactoring candidate.

Several methods in the Observer design pattern are good refactoring candidates [10, 20]. Among these are the *attach* and *detach* methods, which the Observer-Verbs NLA helps iTimna to detect. iTimna uses this rule to correctly identify many attach methods:

$(\mathbf{Observer-Verb} = T)$ and $(Is-Void = T)$ and $(\mathbf{Opposite-Verb} = T) \Rightarrow Classification = T$

```
1  public void mouseReleased(MouseEvent e) {
2      if (isEnabled()) {
3          fState = fOldState;
4          repaint();...
5  }   }
```

Figure 3. The method call to isEnabled() on line 2 is a constraint.

This rule uses two NLA and one TPA to identify refactoring candidates. Figure 4 is an example of a method that this rule identifies. Because the method removeListener() is in the same class as addListener(), the Opposite Verb NLA returns true, and because the method uses the verb *add* in conjunction with *listener*, the Observer-Verb NLA returns true.

Also within the observer pattern, the notification method is a good refactoring candidate [10, 20]. Unless the notification method is called from many places or happens to exhibit another code smell, it is difficult to identify using TPA alone. iTimna uses the following rule of four TPA and one NLA to correctly classify many methods in JHotDraw.

$(Is-Getter = F)$ and $(Is-Constructor = F)$ and $(Is-Setter = F)$ and $(Is-Void = T)$ and $(\mathbf{Observer-Verb} = T) \Rightarrow Classification = T$

The TPA characteristics are not discriminatory by themselves; many methods exist that return void and are not a getter, a setter, or a constructor. The NLA helps iTimna identify fireViewSelectionChangedEvent() (shown in Figure 5) as a refactoring candidate.

These examples demonstrate how iTimna uses TPA and NLA together to identify refactoring candidates. Many simple TPA, not very helpful by themselves, combine with NLA and other TPA to create an accurate classification rule. All of the TPA from the original Timna were used in iTimna, yet fewer TPA appear in the ruleset of iTimna. We believe that this is partially due to adding more analyses (features) to the machine learning problem, but also due to the stronger classifying ability of the NLA, which causes the NLA to be used instead of TPA.

6. Related Work

Researchers have investigated several individual aspect mining analyses. Generally, they either employ static or dynamic techniques. iTimna goes beyond the individual static and dynamic analyses presented below because iTimna *(a)* uses NLA and *(b)* combines many analyses (both TPA and NLA).

The call graph fan-in analysis is a promising static analysis approach [17]. An automated tool based on this research produces reasonably precise results. Two groups have investigated the use of code clone detection tools for the discovery of refactoring candidates. Shepherd et al. used PDG-based clone detection to discover candidates with very high precision [21]. Magiel and van Deursen compared token-based and AST-based clone detection techniques for candidate discovery, finding no clear winner between these methods, but confirming that cross-cutting functionality is often implemented using code clones [2].

```
1  /** Registers the listeners for this window */
2  protected void addListeners() {
3      addWindowListener(new WindowAdapter() {
4              public void windowClosing(WindowEvent event){
5                  exit();
6              }});  }
```

Figure 4. addListeners(), an attaching method.

```
1  /** An appropriate event is triggered and all registered
2   * observers are notified if the drawing view has been
3   * changed, ...*/
4  protected void fireViewSelectionChangedEvent(...) {
5      Object[] listeners = listenerList.getListenerList();..
6      for (int i = listeners.length-2; i>=0 ; i-=2) {...
7              vsl = (ViewChangeListener)listeners[i+1];
8              vsl.viewSelectionChanged(oldView, newView);...
9  }   }
```

Figure 5. A notification method with strong NL clues

Silvia et al. [1] performed several experiments to use program traces to identify candidates. They search for specific patterns in a trace, identifying these methods as candidates. This technique appears very promising; we hope to integrate it into our framework. Tonella et al. [25] used formal concept analysis to analyze program traces. They examined the generated concept lattice and used it to assist in making a classification.

After our original work on combining TPA or aspect mining, Ceccato et al. [5] suggested different methods for combining techniques, based on their experience using three different mining techniques on the same program. Their findings support Timna's use of machine learning to discover which analyses are important when classifying different types of candidates. Researchers have also introduced "sorts", or categories, of aspects, which helps researchers compare different mining techniques. Marius et. al. have also introduced simple approaches for combining mining analyses, such as unioning or intersecting their results [16]. iTimna, in contrast, uses empirical data and machine learning to decide how to combine mining analyses to identify refactoring candidates.

7. Conclusion

We have demonstrated how supplementing traditional program analyses with natural language analyses can substantially improve the effectiveness of an aspect mining system called iTimna. We believe that our initial results strongly warrant further evaluation of iTimna as well as the use of natural language analyses to improve tools that perform other software engineering tasks.

References

[1] Silvia Breu and Jens Krinke. Aspect mining using event traces. In *Automated Software Engineering Conference*, 2004.

[2] Magiel Bruntink, Arie van Deursen, Remco van Engelen, and Tom Tourwe. An evaluation of clone detection techniques for identifying cross-cutting concerns. In *Int. Conf. on Softw. Maint.*, 2004.

[3] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. On the use of line co-change for identifying crosscutting concern code. In *Int. Conf. on Software Maintenance*, 2006.

[4] Bruno Caprile and Paolo Tonella. Nomen est omen: Analyzing the language of function identifiers. *Working Conf. on Reverse Eng.*, 1999.

[5] 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.

[6] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1), 1990.

[7] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

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

[9] James Gosling, Bill Joy, and Guy Steele. Java Language Specification. online, September 2006. http://java.sun.com/docs/books/jls/second_edition/html/names.doc.html.

[10] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In *Int. Conf. on Object-oriented programming, systems, languages, and applications*, 2002.

[11] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *European Conf. on Object-Oriented Programming*, 2001.

[12] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., 2003.

[13] M. M. Lehman and L. A. Belady, editors. *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[14] Ben Liblit, Andrew Begel, and Eve Sweeser. Cognitive perspectives on the role of naming in computer programs. In *Annual Psychology of Programming Workshop*, Sussex, United Kingdom, 2006.

[15] Andrian Marcus, Rainer Koschke, Arie van Deursen, Vclav Rajlich, Paolo Tonella, and Harry Sneed. Identification of concepts, features, and concerns in source code. Panel Discussion at the International Conference on Software Maintenance, 2005.

[16] Marius Marin, Leon Moonen, and Arie van Deursen. A common framework for aspect mining based on crosscutting concern sorts. In *Working Conf. on Reverse Engineering*, 2006.

[17] Marius Marin, Arie van Deursen, and Leon Moonen. Identifying aspects using fan-in analysis. In *Working Conf. on Reverse Eng.*, 2004.

[18] Hausi A. Muller, Jens H. Jahnke, Dennis B. Smith, Margaret-Anne D. Storey, Scott R. Tilley, and Kenny Wong. Reverse engineering: a roadmap. In *ICSE — Future of SE Track*, pages 47–60, 2000.

[19] David Shepherd, Zachary P. Fry, Emily Hill, Lori Pollock, and K. Vijay-Shanker. Using natural language program analysis to find and understand action-oriented concerns. In *Int. Conf. on Aspect-oriented Software Development*, 2007.

[20] David Shepherd, Jeffery Palm, Lori Pollock, and Mark Chu-Carroll. Timna: A framework for combining aspect mining analyses. In *International Conference on Automated Software Engineering*, 2005.

[21] David Shepherd, Lori Pollock, and Emily Gibson. Design and evaluation of an automated aspect mining tool. In *International Conference on Software Engineering Research and Practice*, 2004.

[22] David Shepherd, Lori Pollock, and K. Vijay-Shanker. Towards supporting on-demand virtual remodularization using program graphs. In *Proc. of Int. Conf. on Aspect-oriented Software Development*, 2006.

[23] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Questions programmers ask during software evolution tasks. In *Int. Symposium on Foundations of Software Engineering*, 2006.

[24] Peri Tarr, Harold Ossher, William Harrison, and Jr. Stanley M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Int. Conf. on Softw. Eng.*, 1999.

[25] Paolo Tonella and Mariano Ceccato. Aspect mining through the formal concept analysis of execution traces. In *Working Conference on Reverse Engineering*, 2004.

[26] Ian H. Witten and Eibe Frank. *Data mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.