Towards Supporting On-Demand Virtual Remodularization Using Program Graphs

David Shepherd, Lori Pollock, and K. Vijay-Shanker
Computer and Information Sciences
University of Delaware
Newark, Delaware 19816
shepherd, pollock, vijay@cis.udel.edu

ABSTRACT

OOP style requires programmers to organize their code according to objects (or nouns, using natural language as a metaphor), causing a program's actions (verbs) to become scattered during implementation. We define an *Action-Oriented Identifier Graph* (AOIG) to reconnect the scattered actions in an OOP system. An OOP system with an AOIG will essentially support the dynamic virtual remodularization of OOP code into an *Action-Oriented View*. We have developed an algorithm to automatically construct an AOIG, and an implementation of the construction process. To automatically construct an AOIG, we use Natural Language Processing (NLP) techniques to process the natural language clues left by programmers in source code and comments, and we connect code segments through the actions that they perform. Using a reasonably sized program, we present several applications of an AOIG (feature location, working set recovery, and aspect mining), which demonstrate how the AOIG can be used by software engineering tools to combat the tyranny of the dominant decomposition.

Categories and Subject Descriptors: D.2.3 [Software Engineering]: Coding Tools and Techniques

General Terms: Design, Experimentation, Languages

Keywords: Reverse engineering, Program analysis, Feature Location, Remodularization

1. INTRODUCTION

Regularly, for both maintenance and development tasks, developers need to locate, navigate, and understand concepts in source code. Many crosscutting concerns (CCCs) exist in modern systems, which make these tasks difficult [28, 18, 37, 19]. When given the hard choice of well modularizing an object or an action, the OOP programmer must choose the object, because objects are the *dominant* decomposition [37]. We believe that many CCCs exist due to this dominant decomposition. When expert OOP programmers were compared with expert procedural programmers while performing maintenance tasks, it took the OOP programmers significantly longer to develop a mental model of program information (e.g., control flow, data flow, etc.), suggesting that OOP obscures these action-oriented features [13].

When programmers work on CCCs, a great deal of their time is spent performing several tasks not traditionally viewed as programming tasks. Specifically, they spend much of their time (a) maintaining a working set of modules, (b) locating related modules, and (c) navigating between modules [19, 28]. Studies suggest that software has become so complex that developers spend more time looking for code than actually modifying code [17]. Developers often have to follow long and obscure structural paths in order to find code related to one action [35]. The basic building blocks of programming have become feature location, working set recovery, and program navigation. We suspect that much of the overhead of these tasks could be eliminated if the code could be re-organized to bring modules related to the task at hand into one virtual module [12]. Often, this means remodularizing code around actions, or verbs.

This paper reports on our work developing a graph that can support virtual on-demand program remodularization. The graph can be used to generate a view of the code base that presents related code segments as if they were in one file. We have built concrete tools (a feature location, a working set recovery, and an aspect mining tool) that demonstrate the utility of such a graph, and the potential to speed up the building blocks of modern programming tasks. The novel aspects of our approach are (1) using Natural Language Processing (NLP) as a program analysis tool, (2) developing an underlying representation to support both NLP and traditional program analysis, and (3) representing an Object-Oriented program in terms of its verbs.

We first discuss a small motivating example, and then the rationale for extracting more than just verbs. We then define the action-oriented identifier graph (AOIG) and how to automatically construct the AOIG from analysis of an OOP source code using our `AOIGBuilder` plugin. We present several applications of the AOIG, which demonstrate how the AOIG can be used by engi-

AOSD 06, March 20–24, 2006, Bonn, Germany

```
public Event perform(HttpServletRequest request)
   throws HTMLActionException {
   // Extract attributes we will need
   String actionType= (String)request.getParameter("action");
   HttpSession session = request.getSession();
   // get the shopping cart helper

   CartEvent event = null;
   if (actionType == null) return null;
   if (actionType.equals("purchase")) {
      String itemId = request.getParameter("itemId");
      event = new CartEvent(CartEvent.ADD_ITEM, itemId);
   }
   else if (actionType.equals("remove")) {
      String itemId = request.getParameter("itemId");
      event = new CartEvent(CartEvent.DELETE_ITEM, itemId);
   }
   ...
```

```
public EventResponse perform(Event e) throws EventException {

   CartEvent ce = (CartEvent)e;
   ShoppingClientFacadeLocal scf = null;
   scf = (ShoppingClientFacadeLocal)machine.getAttribute(
            PetstoreKeys.SHOPPING_CLIENT_FACADE);
   ShoppingCartLocal cart = scf.getShoppingCart();

   switch (ce.getActionType()) {

      case  CartEvent.ADD_ITEM  : {
         cart.addItem(ce.getItemId());
      }
      break;

      case  CartEvent.DELETE_ITEM : {
         cart.deleteItem(ce.getItemId());
      }
      break;
      ...
```

Figure 1: Implementation of the action "delete item" from cart in Petstore

Figure 2: Implementation of the action "delete item", Visualizer View

neering tools through enabling virtual remodularization of the source code.

2. MOTIVATING EXAMPLE

Consider locating the implementation of the action "delete item" in Petstore[1], which involves removing an item from a shopper's cart. The user request to remove an item is interpreted by a request handler object (shown in Figure 1, left), and later processed by another object (shown in Figure 1, right). The structural path to connect these two code segments is long and difficult to follow. The code used to implement DELETE ITEM is scattered across three other classes (for a total of 5 classes), which are connected by a complicated structural path. Figure 2 shows the entire concern in the AJDT's Visualizer View [2]. Each rectangle represents a source file, with varying lengths, and shaded lines represent the parts of each file that implement DELETE ITEM.

As a developer interested in DELETE ITEM, it is difficult to discover both segments in Figure 1, even after finding one segment, because the structural path between them is long and obscure. Finding all of the code related to DELETE ITEM is even more difficult, because of lexical differences in the action's implementation. For instance, the code on the right of Figure 3 is likely to be overlooked by a programmer who only searches for "delete", yet it is clearly relevant. In the best case, all of an action's code would be in one module, but this is often not possible in OOP, and it is not desirable as the primary decomposition (remember procedural programming?).

[1]an example J2EE application of approximately 10K LOC provided by Sun Microsystems

3. APPROACH AND RATIONALE

Since the structural dependency links linking related code are often long and obscure, we are investigating the use of NLP to connect the verbs within a program, strongly connecting the scattered pieces of an action. In a programming language, verbs correspond to actions (or operations) and nouns correspond to objects [7]. The proposed *Action-Oriented Identifier Graph* represents the actions in a program, supplemented with the direct object of each action.

Often verbs, such as "remove," act on many different objects in a single program. In PetStore, a lexical search for the string "remove" locates actions such as "remove attribute", "remove screen", "remove entry", and "remove template". Therefore, it is important to consider the *theme* to precisely identify a specific action. There is an especially strong relationship between verbs and their themes in English [10]. A *theme* is the object that the action (implied by the verb) acts upon, and usually appears as a direct object (DO). An example of a verb-DO relationship in natural language is (parked, car) in the sentence "The person parked the *car*."

The verb-DO relationship can be used for various purposes. For example, [29, 16] use the verb-DO relationship to classify nouns. Similar nouns can be clustered into a hierarchy by examining the verbs with which the nouns are used as DOs (e.g., consider the nouns that can be DOs of the verb "eat" versus those for the verb

```
public void deleteItem (String itemID) {
    cart.remove(itemID);
}
```

```
public void updateItemQuantity (String itemID, int newQty) {
    cart.remove(itemID);
    // remove item if it is less than or equal to 0
    if (newQty > 0) cart.put(itemID, new Integer(newQty));
}
```

Figure 3: Synonyms as clues: Delete Item, in an Expected and Unexpected Method

"drink"). We focus on DOs instead of subjects, because, in OOP code, the enclosing object is usually the subject (e.g., the `File` class has a `store` method, where the `File` stores objects that are passed in as parameters), but the DO is usually part of a crosscutting concern (e.g., the object being passed in to `store`).

In order to leverage this information about programs, we process source code to extract verb-DO pairs. Then, we can build tools which use these pairs to aid the user in navigating and viewing code in a way that crosscuts the dominant decomposition. The representation of these pairs will be similar to inverse indexing (as used in Information Retrieval), where we map verb-DO pairs to occurrences in actual code.

4. MODELING VERB-DO RELATIONS IN PROGRAMS

4.1 Definition of the AOIG

We have designed a novel program model that captures the action-oriented relations between identifiers in a source program. Specifically, the model explicitly represents the occurrences of verbs and direct objects (DOs) in a program, as implied by the usage of user-defined identifiers. We only analyze occurrences of verbs and DOs in *method declarations*, and *comments* within or referring to method declarations because method declarations are the core of concerns. For instance, if we can locate a method which implements the concern NOTIFY OBSERVERS with a particular concern mining tool, then there is no need for the concern mining tool to identify the calls to that method as well, because that can be done via traditional program navigation if necessary. The action-oriented identifier model is defined to be a graph, called the *action-oriented identifier (AOIG) graph*, which contains four kinds of nodes:

- A *verb node* exists for each distinct verb occurring in the program.
- A *direct object (DO) node* exists for each unique direct object in the program.
- A *verb-DO node* exists for each verb-DO pair identified in the program. A verb-DO pair is defined to be two colocated identifiers in which the first identifier is discovered to be an action or verb, and the second identifier is being used in the role of a direct object for the first identifier's action.
- A *use node* exists for each occurrence of a verb-DO pair in a program's comments or code.

There are two kinds of edges in the AOIG:

- A *pairing edge* has a verb node v or a DO node d as the source, and a verb-DO node as the sink, when it is determined that the verb v and DO d are used together. A given verb (or DO) node may have edges to multiple verb-DO nodes; however, a given verb-DO node has only two incoming edges, one from the verb and one from the DO node involved in the relation.
- For each occurrence (or use) of a given verb-DO pair in the program, there is a distinct *use edge* mapping the corresponding verb-DO node to the use node representing the use of that pair in the program.

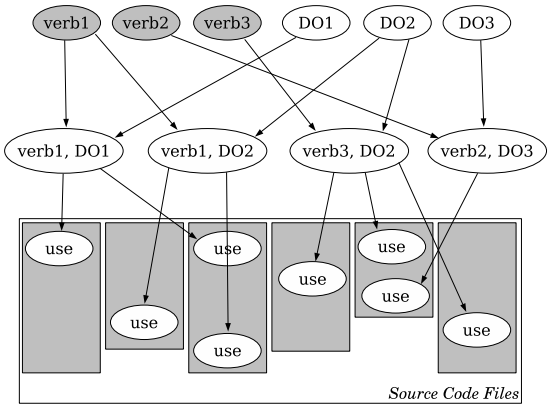

Figure 4: Example of an AOIG

Figure 4 shows the form of an AOIG. In this figure, we can see that `verb1` has two *pairing edges*, one to <`verb1, DO1`> and one to <`verb1, DO2`>, which are both *verb-DO nodes*. <`verb1, DO1`> has two *use edges*, which represent locations in the source code where this pair occurs. This representation is intentionally simple, allowing for complicated analyses to extract meaning from it efficiently.

4.2 Overview of AOIG Construction

We have built the AOIGBuilder as an Eclipse plugin. It builds a persistent AOIG automatically when the user triggers a build. Although the current implementation does not allow incremental builds, the algorithm should perform well incrementally, since each method or class can be processed independently.

Extracting an AOIG from source code involves analyzing natural language (comments) as well as program structures (declarations). We use the process illustrated in Figure 5 to construct the AOIG for an OOP program. The paths to analyze comments and source code diverge at the `Splitter-Extractor` box. The process to extract verb-DO pairs from comments is illustrated as the path that starts at `B`, and extracting pairs from method signatures is illustrated as the path that starts at `A`. Once the pairs are extracted, we use them to cre-

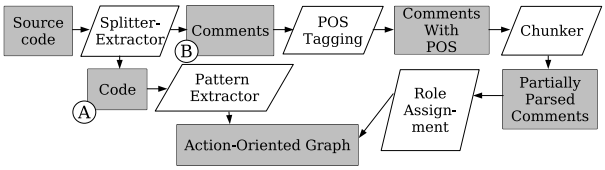

Figure 5: AOIG Construction Process

ate the appropriate nodes and edges in the AOIG. The next subsections describe paths `A` and `B` in detail.

4.3 Extracting Pairs from Comments

To extract verb-DO pairs from comments, we follow three main steps:

1. Part-of-speech (POS) tagging on each comment
2. Chunking on each POS tagged comment
3. Pattern matching (i.e., role assignment) on each chunked comment to determine verb-DO pairs

In order to create *verb-DO nodes* due to uses in comments, a POS tag (e.g., verb, adjective, noun etc.) is assigned to each word in each comment. Highly accurate (achieving precision around 97%) and efficient taggers [30, 8] are freely available. These part of speech tags are used to chunk the sentences into basic phrases such as base noun phrases, verb group sequences, and prepositional phrases. While natural languages are notorious for ambiguity, chunking (also called robust parsing or partial parsing) can be done accurately [36, 1]. Furthermore, chunking is sufficient for the purpose of the detection of verbs and their direct objects.

We detect direct objects by simple pattern matching, finding noun phrases immediately following verbs in active voice, by skipping over some possible verbal modifiers like adverbs, or scanning for a subject in the case of passive verb groups.

Each of the components in Figure 5 (POS tagger, noun phrase chunker, role assignment, etc.) can be done with accuracy over 90% even in complex texts [20, 30, 10]. We have used these components for other tasks and have noticed these freely available components port well (requiring minor modifications) to other domains such as scientific literature [38, 31]. In our implementation, we used the components from OpenNLP [27]. Once the verb and the DO are both detected, we construct a *use node* to represent the comment and connect it to the corresponding *verb-DO node*, creating one if necessary, which could cause cascading creates of a *verb node* or a *DO node*, if either has not been constructed yet.

As an example, consider the comment "Removes an item from a cart". We would tag the comment (Remove$_{<v>}$ an$_{<dt>}$ item$_{<n>}$ from$_{<p>}$ a$_{<dt>}$ cart$_{<n>}$), partially parse the comment (Remove$_{<v>}$ [an item]$_{<NounPhrase>}$ [from [a cart]$_{<NounPhrase>}$]$_{<PrepositionalPhrase>}$), and use pattern extraction to determine the verb-DO pair (Remove, item). We then construct a *use node* at that comment, and attach it to the (Remove, item) *verb-DO node*, creating one if necessary.

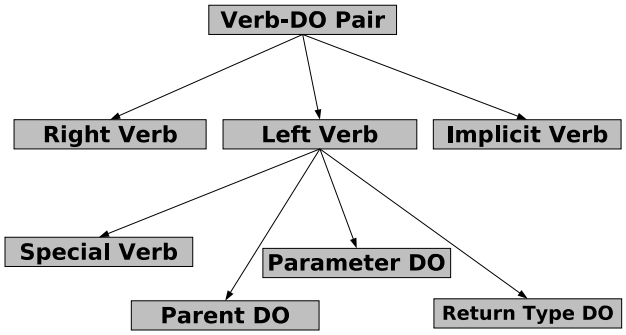

Figure 6: Extraction Classification Hierarchy

4.4 Extraction from Method Signatures

To extract *verb-DO pairs* due to uses in method signatures, we developed a classification system for the different method signature forms that we encountered, and a set of rules to identify each class. The extraction class of the method signature determines where to extract the verb and DO (see Table 1). In order to determine the class of a previously unclassified method signature, we merge the rule sets from all classes into one ordered ruleset, optimized for precision. Thus, we perform the following steps:

1. Process the method name
   (a) Split method name (e.g. writeToFile → Write To File [39, 35])
   (b) Perform POS tagging on split method name
   (c) Perform chunking on tagged method name
2. Use classifying rules to determine the extraction class of the method signature. If a rule requires program analysis information, perform appropriate analysis.
3. Given the method signature's classification, extract the verb, DO, and verb-DO pair from the appropriate positions in the method signature

4.4.1 Extraction Classification Hierarchy

We present our classification system as a hierarchy in Figure 6. At the first level, we categorize every method signature by where the verb appears. For instance, method signatures are classified as `Left Verb` when the verb appears as the leftmost word in the method name (e.g., public File *convert***File()**). The `Right Verb` class corresponds to method signatures such as public void **mouse***Dragged()*, where the verb is the rightmost word in the method name. The class `Implicit Verb` consists of method signatures where no verb appears (e.g., public Handle **handles()**, which is essentially a getter method).

The class `Left Verb` can be further subclassed into four specializations based on the location of the DO — including `Special Verb, Parent DO, Parameter DO`, and `Return Type DO`. Table 1 presents the current extraction classification for method signatures. For each extraction class, the locations of the verb and DO are

Class	Verb	DO	Example
Parent DO	Leftmost word in method name	Parent class	public void *start*() in class **RequestHandler**
Return Type DO	Leftmost word in method name	Return type	public **Factory** *create*(Properties p)
Parameter DO	Leftmost word in method name	Parameter type	public boolean *process*(**Request** r)
Special Verb	Leftmost word in method name	Specific to Verb	public void *get***Name**()
Right Verb	Rightmost word in method name	Leftmost word in method name	public void **mouse***Dragged*()
Implicit Verb	get	Method name	public List **handles**()
Left Verb	Leftmost word in method name	Rightmost word in method name	public URL *parse***Url**()

Table 1: Method Signature Classification for Verb and DO Identification

Class	Rule Set
Parent DO	1. POS analysis determines a verb is the leftmost word in the method name 2. If the method has a non-void return type and parameters, but the parameters are more specific than the return type, classify as Parent DO
Parameter DO	1. POS analysis determines a verb is the leftmost word in the method name 2. If the method has a void return type and parameters, classify as Parameter DO
Return Type DO	1. POS analysis determines a verb is the leftmost word in the method name 2. If the method uses a creational verb and has a non-void return type, classify as Return Type DO 3. If the method has a void return type and parameters and the return type is more specific than the parameters, classify as Return Type DO
Special Verb	1. POS analysis determines a verb is the leftmost word in the method name 2. If the method uses a special verb, such as "get", "set", "to", etc., classify as Special Verb
Right Verb	1. POS analysis determines a passive verb is the rightmost word in the method name. 2. If that verb ends in "ing" or "ed", classify as Right Verb
Implicit Verb	1. POS analysis determines a verb does not exist in the method name. 2. If the class has a field whose identifier or type closely resembles the method name, classify as Implicit Verb

Table 2: Partial Rule Set (add Table 3 for Complete Rule Set)

described, and then an example method signature with the DO in bold is presented.

4.4.2 Rule Sets for Classification

Based on manually examining many method signatures from several programs, we developed rule sets for each extraction class. These rule sets use both NLP and program analysis information to test for a particular class. To use NLP information, we first preprocess a method signature, breaking its method name into words via known heuristics [39, 35], and using a POS tagger and a chunker to process it. We also effectively preprocess a method signature for program analysis information, by using Eclipse's JDT API. An example rule set that leverages both NLP and program analysis information is `Implicit Verb`'s set in Table 2. Rule 1 determines that no verb exists in the method name via NLP, and rule 2 determines if the method name is similar to a field in the method's class through simple program analysis.

The complete rule list used to classify an unknown method signature is the merged set of all rule sets from every extraction class. We present the merging of only two rule sets here, to demonstrate the rule set integration. Table 3 presents a rule set for both the `Parent DO` and `Left Verb` classes with example code fragments. The rest of the rule sets are shown in Table 2. Consider constructing the ordered rule list from the two rule sets in Table 3. Since rule 1 is the same for both classes, this rule would appear first. However, since rules 2a and 2b are different, we must decide which rule will be applied first for class integration. In this case, we would execute 2b before 2a, because if a method name has a noun phrase after its verb, it is highly likely that this noun phrase is the direct object, even if the method has a void return type and no parameters. From our experience with classifying method signatures by hand, we were able to determine an order for all the rules that would classify method signatures well.

4.5 Time and Space Costs

4.5.1 Space Costs

The number of *use nodes* is, in most cases, the largest cost of building the AOIG. This is because there are usually several *use nodes* for each *verb-DO pair node*, and therefore several *use nodes* for each *verb* and *direct object* node. Since the number of *use nodes* is, in practice, easily larger than the other types, we focus our analysis on use nodes. One *use node* is built for every method signature, causing O(m) nodes in the graph, where m is the number of method signatures in a program. Up to C *use nodes* may be built for each comment, where C is the largest number of sentences in a comment, because each sentence in a comment could provide a verb-DO pair. Since each method could have up to N comments, where N is the largest number of comments in a method, there could be O(N*C*m) *use nodes* built. However, since N and C are usually small constants (less than 10), this reduces to O(m) space.

4.5.2 Time Costs

Creating the AOIG for a program requires scanning every method once (O(m)), examining its method sig-

Parent DO Extraction Class	Example
1. POS analysis determines a verb is the leftmost word in method name	public void *initialize*() in class **Container**
2a. If method has void return type and no parameters, then classify as Parent DO	{initialize, container}
Left Verb Extraction Class	Example
1. POS analysis determines a verb is the leftmost word in method name	public void *show***Help**() in class DrawingApp
2b. If the remaining text of the method name is a noun phrase, then classify as Left Verb	{show, Help}

Table 3: Example Rules for Extraction Classification

nature and any comments associated with the method. Within this scan, the processing of comments and method signatures each require time. As above, let N be the largest number of comments in a method. Comments require slightly more time to process than method signatures, because of the use of NLP, but this process can be bounded by a small constant L. Method signatures are quicker to process, and their processing time can be bounded by a small constant P. Since both processes only require a small, constant time, the time costs can be reduced from $O((L*N + P)*m)$ to $O(m)$.

From our implementation of the AOIG, we found that the space and time costs were very reasonable. The AOIG required only a fraction of the space that the corresponding source code required, and AOIG construction time was approximately 10 seconds per source file for a file with about 10 methods and 10 multi-sentence comments. The AOIG construction process spent about 1 second analyzing the method signatures and about 9 seconds analyzing the comments. Since this process can be done incrementally (the AOIGBuilder can process each new method signature and comment added to code, as they are added), these times are reasonable for a prototype implementation (hardware: Pentium 4 CPU 2.40GHz).

4.6 Precision and Recall

To evaluate the precision and recall of `AOIGBuilder`, we manually examined an AOIG generated by `AOIGBuilder`. Specifically, we manually inspected a set of 57 randomly selected method declarations and the comments associated with these method declarations, marking a use as *correct* only if the `AOIGBuilder` had extracted the desired verb-DO pair (see Table 4 for results). We calculated *precision* as $correct/(correct + incorrect)$, and *recall* as $correct/(correct + incorrect + omitted)$, where *correct* (*incorrect*) is the number of correctly (falsely) identified verb-DO pairs, and *omitted* are verb-DO pairs not found and thus not included in the AOIG generated by `AOIGBuilder`. We found that both precision and recall were high for method declarations, with 92.5% precision, and 87.7% recall in the surveyed sample. We also found that precision was high for comments (97.7%), but the recall was slightly lower (78%), due to the simplicity of our pattern extractor (which could be improved in a production version). These results were consistent with our broader experience with `AOIGBuilder`, where we found that a high percentage of the AOIG was generated accurately.

4.7 Potential Uses of AOIG

Category	**No. of Verb-DO Prs**	**No. of Verb-DO Prs**
	Declarations	**Comments**
correct	50	43
incorrect	4	1
omitted	2	11
total	57	55

Table 4: Results of Survey

The AOIG is intended to allow software engineers to construct tools that help overcome the tyranny of the dominant decomposition. We foresee the AOIG being useful in powering tools that perform feature location, working set discovery/recovery, concern/aspect mining, and other maintenance tasks. The next three sections describe potential uses we have studied.

5. FEATURE LOCATION

5.1 Problem and Background

Biggerstaff et. al. [6] was one of the first researchers to articulate the *concept assignment* problem in program understanding, as the problem of discovering individual human-oriented concepts and assigning them to their code implementation. Our work is on feature location, which we define as a subproblem of concept assignment, in which the concept can be expressed as an action-object pair, such as (LOAD FILE). Most features programmers wish to discover involve action (such as tasks during development or debugging), and the only concepts not found with our technique are those expressed without action (such as only an object, which would already be well modularized in an OOP program).

Dynamic approaches to the general problem of concept assignment use program traces [21, 24, 11]. This approach requires a set of test cases that exercise targeted concepts, and a mapping of tests to concepts; our approach requires neither. Dynamic approaches then use deductive reasoning to eliminate methods that are not central to a specific feature. It can be difficult to construct test cases that exercise some human concepts, but this approach works well with GUI-based programs [24, 11]. Information retrieval (IR) approaches map queries to the most related source code [22]. However, this approach suffers from the imprecision associated with IR, which was initially designed to handle much larger document sets. Tools have been designed which leverage both IR and dynamic analysis [39]. By using NLP, our approach is able to extract more precise information from documents (e.g., we can identify syn-

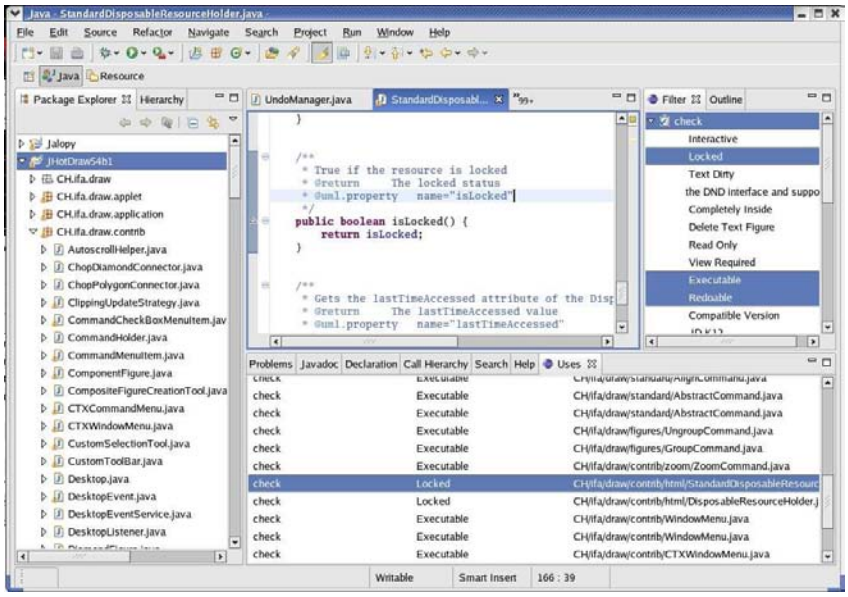

Figure 7: `ViRMoVis` in Eclipse

onyms as such, use context to deduce the meaning of a word, etc.).

5.2 Requirements

A feature location tool should assist developers in finding features during common programming tasks. It should find features that are directly relevant to debugging, development, and maintenance. Most tasks that a programmer must do, especailly late in the lifecycle of a program, relate to an action, so the tool should be able to search for these types of features. The user should be able to add to his search a specification of which objects the action interacts with, in order to find more specific features.

5.3 Implementation

We implemented `ViRMoVis` a simple virtual remodularization visualization tool, as an Eclipse plugin. It is intended to expose the AOIG's natural remodularization ability in order to demonstrate the AOIG's utility. We envision, in the future, tools that go beyond exposing the AOIG and use the AOIG to build more complex remodularization functionality.

`ViRMoVis` consists of two Eclipse views, shown in Figure 8, and, within the context of Eclipse in Figure 7. It provides the following features:

- A user can define a filter in the *Filter View* to restrict the *use nodes* displayed in the *Uses View*.
- A user can add verbs to the *Filter View*.
  - A user can browse code, then right click on code, which will display verbs that are used nearby. The user can then select a verb.
  - A user can manually enter any verb.
  - A user can browse the *Filter View*. When the user right clicks on a verb, semantically related verbs (via Wordnet [25]) are shown, and the user can select one.
- A user can browse the *Uses View* to follow links to code related to a particular verb or verb-DO pair.

The *Filter View*'s selected members affect the *Uses View*. If verbs or DOs are selected in the *Filter View*, then only uses that match at least one selected verb and one selected DO are shown.

5.4 Feature Location with `ViRMoVis`

A user has to follow the following steps in order to find a feature with `ViRMoVis` :
1. Input a verb into the *Filter View*. `ViRMoVis` responds to right-clicks by displaying a set of related verbs.
2. Add related verbs to the *Filter View*. `ViRMoVis` displays all DOs for each selected verb.
3. Choose subset of provided DOs related to a feature. `ViRMoVis` displays all uses of the selected verb-DO pairs.
4. Inspect the results

We used `AOIGBuilder` to construct an AOIG for JHotDraw, a Java drawing application implemented as an exercise in good design. As a test for `ViRMoVis` , we used it to locate the feature ACTIVATE TOOL. This feature corresponds to a user choosing a tool (such as a pen or an eraser) and the application activating that tool for use on the drawing pane.

To find ACTIVATE TOOLS we first input the verb "activate" into the *Filter View*. Then, we can right click on "activate", and a menu will provide related verbs that appear in the program, such as "Activated", "activating", and even synonyms such as "start". The user decides to just add verbs that are variations of the stem "activate". Under each verb that is added to the *Filter View*, a list of all the DOs that it acts upon are provided. We select a subset of the DOs that are related to "tool"

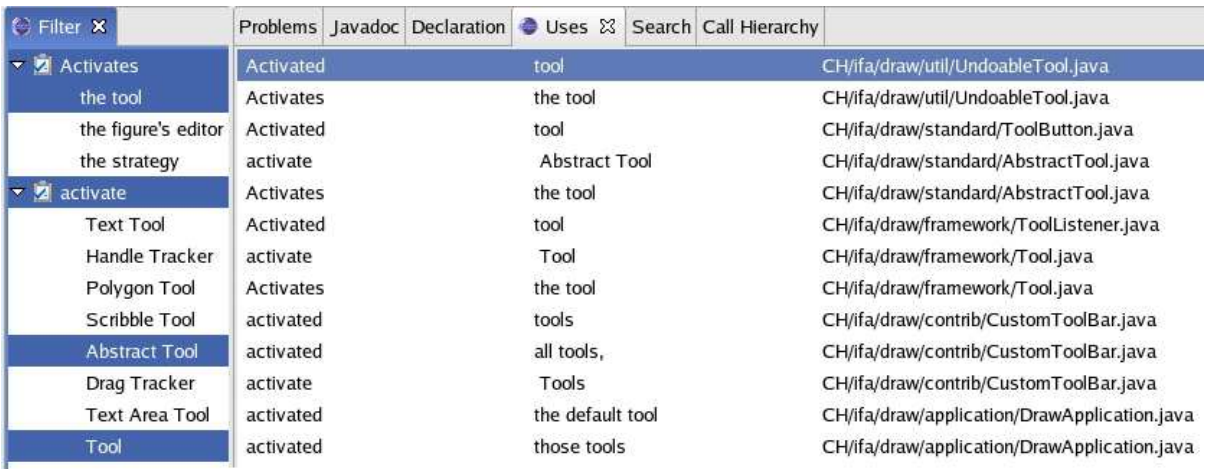

Figure 8: Partial Filter and All Returned Nodes

as shown in Figure 8. We do not select DOs that refer to specific types of tools (such as `Text Area Tool`), because we are only interested in the generic tool activating framework, not each specific tool. Upon defining the filter, the *Uses View* is updated, returning all uses in the program that match a selected verb and a selected DO in the filter. A partial view of the filter and a complete view of the results are shown in Figure 8. The *Uses View* returns 7 files, 6 of which we consider to be the core of the ACTIVATE TOOL feature.

Of course, now the user still has to understand how this set of modules interacts to form the feature, yet the task has become much easier, because the user only has to analyze an extremely small subset of methods and classes. In fact, give this subset, the user can use commercially available reverse engineering tools to create UML diagrams which show the interactions between the modules. This can enable the user to quickly find and then understand a feature which was previously hidden.

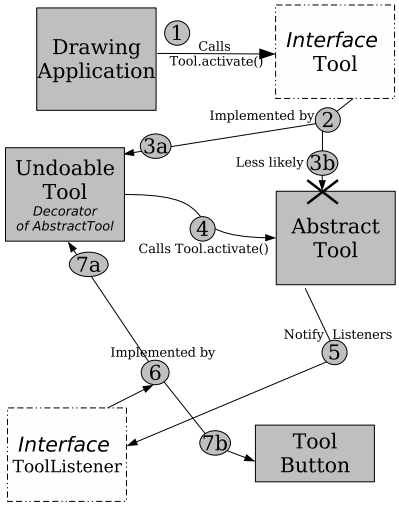

Figure 9: Feature Location in a Modern IDE

5.5 Contributions to Feature Location

Feature location is a building block for many programming activities, and `ViRMoVis` speeds feature location. Although it is difficult to speculate exactly how a user might locate ACTIVATE TOOL without our tool, we can inspect the relationships between all files in the feature, generating an approximation of how difficult this task might be. In Figure 9, we present these relationships, and an ordered path of how a user could traverse these relationships to find these files. Each box represents a class, each white box represents an interface, and each circled number indicates the order of events. Given the number of relationships the user could potentially follow and the complexity of the methods involved in this feature, we believe that using the `ViRMoVis` will lead to faster feature location for most users. (The figure only shows the correct path. At each method, for instance, the user could potentially follow an irrelevant method call.) *`ViRMoVis` allows users to avoid these long and obscure structural paths.* The user without the `ViRMoVis` is at a further disadvantage, because even finding a starting part of the feature could be difficult, as lexical searches break easily (e.g., searching for `delete` when the user used `deleting` in code will return nothing). *`ViRMoVis` provides a way of including semantically related words (i.e., synonyms) and morphologically related words (e.g., `delete` and `deleting`) in a search.*

6. WORKING SET RECOVERY

6.1 Problem and Background

Consider another common programming task, the recovery of a working set of modules. We define working sets as the union of several features, because, with the use of the AOIG, working set recovery reduces to several feature recoveries. Consider the following hypothetical situation. If a programmer is working on a specific task on Friday, yet does not finish before the end of the day, and he attempts to return to that task on Monday he will often have to recover a working set, in order to

understand and manipulate that code. The programmer in our example was working on the feature LOAD DRAWING because `Image` objects were not being loaded correctly. In order to perform this task the developer has to understand both the way a drawing is loaded and specifically how images are handled within that system. We consider this to be the union of two features, LOAD DRAWING and LOAD IMAGE.

Researchers have also done work on discovering and storing working sets [32, 17]. Working sets can be deduced by tracking the files a programmer inspects. Then, a working set aware view of a program's files can greatly reduce the complexity of maintaining a working set [17]. Developers can also explicitly specify and save a working set, for working on the same feature in the future [32]. This is similar (but more robust) than the Eclipse's notion of a working set, which allows for a subset of the overall workspace to be specified as a working set. The AOIG is meant to be used in conjunction with these kinds of tools, and it supplements their advantages with its ability to find modules in working sets prior to a developer visiting or specifying that module. If no developer has previously visited or specified a particular working set, our tool can still be used to find it, whereas the previous tools do not ease this task. Developers can use querying systems to find working sets, and save these queries to quickly virtually remodularize code when returning to that task [4, 32]. These tools are similar because they query programs to find a concern, but they query more traditional program structures (call graph, dataflow information, etc.), whereas we query the AOIG.

6.2 Requirements

A working set of program modules corresponds to some task that the developer is completing. The developer must have a high-level idea of what he is working on. We propose that these high-level ideas can be broken down into action-object pairs. A particular task can involve several different action-object pairs. We propose then, that working set recovery *reduces to* one or more instances of the feature location problem. Therefore, a working set recovery tool must locate several, possibly independent, features.

6.3 Finding Working Sets with `ViRMoVis`

We can use `ViRMoVis` to easily represent a particular working set, because it deals with only one set of verbs. If a working set involves more than one set of verbs, it would be better to have several panes like the one in Figure 8, and we plan this addition in a future version. If we were searching for code related to the groups {LOAD, RESTORE} and {SAVE, RESTORE} we could use one pane for each group.

The user begins by inputting the verb "load" into the *Filter View*, and adding "restore","resurrects", and variations of those words via the related words menu. The user looks at a few of the returned results by double clicking on the files in the *Uses View* (displaying the source code in the editor). He realizes that in almost every file the word read is used as a synonym to load, which is logical in this domain, so he adds read to the *Filter View*. He is then able to select all verbs and all DOs in the *Filter View* that correlate with what he is looking for. He selects all of the verbs in the *Filter View*, but only a subset of the DOs (i.e., drawing, image, drawings, images, default drawing, the Image, etc.). This causes the *Uses View* to return a set that is almost identical to the desired working set.

6.4 Contributions

We have generalized the task of recovering a working set, stating that it reduces to several instances of feature location. This unifies previously separate research efforts.

As noted previously, it is difficult to mimic a potential user's process, so we discuss inherent areas of difficulty associated with this task. A major difficulty of this task is the large number of times that the word `load` appears in the program. It appears in 124 files. *With `ViRMoVis` we can avoid large search results by specifying which verb-DO pair we are interested in* (and ignoring uses of `load` with other DOs). Investigating many of these files, such as the `TextFigure` file, will reveal only how that class is stored; a user must follow references from the `TextFigure` class (or the many other implementors of the `Storable` interface) to understand how the persistence system is implemented. *`ViRMoVis` allows the developer to avoid following long structural paths by connecting files through the uses of verb-DO pairs.*

Another difficulty is locating only the files that are related to loading an image. Since a lexical search for `load` returns so many files, a user might also search for `image`. However, this search returns many results as well. `ViRMoVis` allows the user to specify a search for both IMAGE and LOAD, returning only the points where LOAD interacts with IMAGE (avoiding chance colocations of LOAD and IMAGE).

7. ASPECT MINING AND NAVIGATION

7.1 Problem and Background

Researchers have investigated several ways to find opportunities for refactoring into an aspect-oriented program. The most promising techniques for semi-automated aspect mining are from lexical-based tools and individual program analyses. Lexical search-based tools, such as AMT [15] and the Aspect Browser [14], 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. These tools rely on lexical searches, which are known to be fragile, and do not provide a method for systematically mining aspects in favor of a more on-demand approach. The user must have text to search for, instead of the tool providing the most scattered feature. Individual aspect mining analyses, which automatically identify a set of refactoring candidates, have also shown promise [23, 9]. These types of analyses have done better when combined to form a more general framework [34]. These types of analyses can give the user line numbers of code to refactor, but they do

not group line numbers into coherent concerns (such as all line numbers related to caching), so manual processing is required afterwards.

7.2 Requirements of a Mining Tool

A mining tool should be able to help a developer mine concerns he knows exist as well as concerns which he is unaware of, but are poorly modularized. The second is particularly valuable for a developer who wishes to increase the readability, maintainability, and usability of his code in a systematic manner. A mining tool should rank concerns from worse modularized to best, so that the developer can concentrate his refactoring efforts on concerns that are more likely to slow the development process. We focus on the process of identifying seeds, or starting points, for refactoring. Given a set of seeds related to a specific concern, a developer must manually expand this set to include all of the concern's code by following structural links from the initial seeds. This is consistent with the state of the art in aspect mining.

7.3 Implementation of `CCVerbFinder`

In order to validate the usefulness for aspect mining, we have constructed a simple mining tool which uses the AOIG. As with other tools, this tool represents a first step in using the AOIG for this purpose, and we expect to build on it and extend its functionality in the future. We built the `CCVerbFinder` to find verbs that crosscut the system, and are therefore good candidates for refactoring. The `CCVerbFinder` finds these crosscutting verbs by traversing the AOIG, starting at each individual verb node in the verb node set of the AOIG, (or verb-DO node if the desired data is with respect to verb-DO pairs). For each verb node v, the `CCVerbFinder` traverses the pairing edges from v to the reachable verb-DO nodes, and then traverses the use edges from the reachable verb-DO nodes, ending at the *use* nodes reachable from the starting verb node v. The `CCVerbFinder` counts the number of unique files that contain reachable *use* nodes, and reports this *crosscutcount* for the verb v.

In Figure 4, if the `CCVerbFinder` was counting for the verb node labeled `verb1`, it would traverse the AOIG to `<verb1, DO1>` and `<verb1, DO2>`. From each of these verb-DO nodes, it would traverse the use edges to find two uses for each verb-DO pair, for a total of 4 uses of `verb1`, which occur in only three different files. In the end, the verbs, or verb-DO pairs, which generate large file counts are the most likely to implement crosscutting concerns, because their implementation is scattered into a number of files. In the future, this could be done at a more fine-grained level.

7.4 A Case Study with `CCVerbFinder`

In Figure 10, we present the results of running the `CCVerbFinder` on JHotDraw. The graph presents the number of verb-DO pairs that are used in each specified number of files. For instance, there are 5 verb-DO pairs in the system that are used in 4 different files, 14 verb-DO pairs used in 3 different files, and one verb-DO pair that is used in 15 different files. This data shows that there exists a significant number of verb-DO

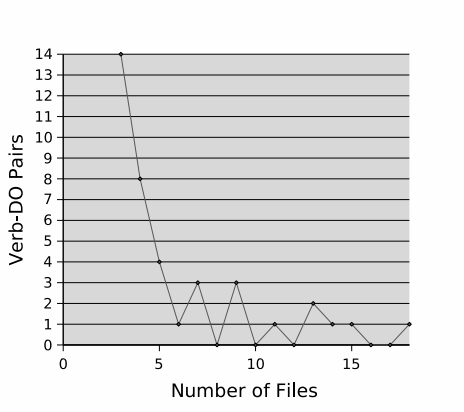

Figure 10: Crosscut Counts of Verb-DO Pairs

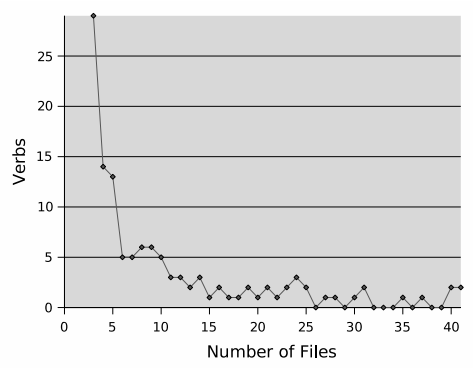

Figure 11: Crosscut Counts of Verbs

pairs (134 total) that are not well modularized (assuming most verb-DO pairs implemented in 3 or more files are likely to be not well-modularized). A few examples of verb-DO pairs that were not well-modularized are (`check, undoable`), (`deactivate, strategy`), and (`disable, tool`). We believe that refactoring these verb-DO pairs into an AOP language could dramatically decrease the time spent on development and maintenance. For many tasks, understanding the surrounding software is simple, because of good OOP style. However, in the cases where the task involves a scattered verb-DO pair, locating all related code can take an arbitrarily large amount of time, because there is often only an obscure structural link between the two (or more) segments of related code. Therefore, if many scattered verb-DO pairs exist in a system, a given task is likely to involve one, and could require an arbitrarily large amount of time. If most pairs were refactored into a better modularization, the developer could access program features without such a burden.

When viewing verb-DO information, we are considering a fine-grained view of the system. `CCVerbFinder` also presents a courser-grained view. In Figure 11, we present how verbs are used in JHotDraw. Figure 11 shows that there are 28 verbs that are used in 3 files, 13 verbs used in 5 files, and 3 in 24 files. A large num-

ber of verbs (39 total) are not well-modularized. A few examples of these verbs are `invalidate`, `deactivate`, and `redo`. Refactoring these verbs could dramatically increase the readability of the system. Refactoring also should improve the consistency of the code, since a verb that is implemented in many different places tends to achieve poorer consistency than one implemented in a single module.

7.5 Contributions to Aspect Mining

A key contribution of our tool is that it automatically finds sets of code to refactor, and it can present these sets in order of severity. This provides the developer with a systematic way of decreasing the amount of crosscutting in his system. Without the `CCVerbFinder` the developer is limited to using either tools that require input, such as a search string, or tools that operate on a low-level representation of the program. Tools that require search strings are not helpful for systematically eliminating crosscutting concerns from a system, because you first have to discover that a particular string is part of a CCC before searching for it. Program analyses that operate on low level representations of a program can find some aspects accurately, but they do not group code together into a coherent concern. The *`CCVerbFinder` presents, as a result, a coherent concern related to only one concept, which can be refactored into an AOP module that represents that concept.*

In its current implementation, the `CCVerbFinder` can present the verbs that crosscut the highest number of files first. In the future, we plan to add other heuristics, such as counting a parent and child class as only one file, to rank these verbs in a more appropriate order. The `CCVerbFinder` provides a systematic way to find and eliminate crosscutting concerns from your system.

8. OTHER RELATED WORK

Researchers have investigated using NLP to understand source code. Specifically, they have investigated how to create links from design level documents to the corresponding design patterns in code by using semi-automated NLP methods [5]. Our work does not seek to link code to design, only to facilitate browsing the existing features, and our technique is highly automated, whereas this work was semi-automated. Researchers have also used *conceptual graphs* to represent programs, for use in code retrieval [26]. A conceptual graph contains structural program information and some higher level information that a reasoner can operate on logically to deduce new information. The conceptual graph is not as focused on the high level, natural language clues that a programmer leaves in code.

Researchers have used language clues to mine aspects in requirements [33, 3]. Some of this work has automated the NLP part [33], but others remain slightly less automated [3]. Both of these works focus on the requirements level, where we focus on the actual code. A focus of one tool is to identify stakeholders, in order to generate viewpoints of the system. This approach also identifies action words which can then be used to link associated words and also identifies a set of words associated with known aspects [33].

9. CONCLUSIONS

In this paper, we have introduced the Action-Oriented Identifier Graph, which helps to re-connect the scattered actions throughout a program. We have shown that NLP technology is sufficiently robust to automatically construct the AOIG graph for a given program to support the construction of useful software engineering tools. We have also demonstrated how the AOIG could be used to help with common programming tasks that are otherwise time-consuming in a modern programming environment. We anticipate a useful future for the AOIG, and believe it can assist developers in feature location, working set recovery/discovery, and general program navigation.

This paper focused on the NLP analysis and tools that utilize it. We believe that additional benefits can be derived by integrating such NLP analysis with methods being used currently for these applications. For instance, it should be straightforward to extend our tools to obtain the functionality of full lexical search. Further, we believe that there is considerable scope to integrate NLP analysis with both dynamic and static program analysis and that such integration will lead to tools that add to the capabilities derived from the individual components.

10. ACKNOWLEDGMENTS

We would like to give a special thanks to Dr. Gail Murphy for her insightful comments on drafts of this paper, and we would also like to thank the members of Hiperspace Lab for their comments on drafts of this paper.

11. REFERENCES

[1] S. Abney. Partial parsing via finite-state cascades. In *Workshop on Robust Parsing, 8th European Summer School in Logic, Language and Information*, 1996.

[2] AJDT Homepage. http://www.eclipse.org/ajdt/. 2005. (January 17, 2005).

[3] Elisa Baniassad and Siobhan Clarke. Theme: An approach for aspect-oriented analysis and design. In *ICSE '04: 26th International Conference on Software Engineering*, pages 158–167. IEEE Computer Society, 2004.

[4] Elisa L. A. Baniassad and Gail C. Murphy. Conceptual module querying for software reengineering. In *ICSE '98: Proceedings of the 20th international conference on Software engineering*, pages 64–73, Washington, DC, USA, 1998. IEEE Computer Society.

[5] Elisa L. A. Baniassad, Gail C. Murphy, and Christa Schwanninger. Design pattern rationale graphs: linking design to source. In *ICSE '03: 25th International Conference on Software Engineering*, pages 352–362. IEEE Computer Society, 2003.

[6] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas Webster. The concept assignment problem in program understanding. In *ICSE '93: 15th International Conference on Software*

Engineering, pages 482–498, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[7] Grady Booch. Object-oriented design. *Ada Lett.*, I(3):64–76, 1982.

[8] Thorsten Brants. Tnt: a statistical part-of-speech tagger. In *Sixth Conference on Applied Natural Language Processing*, pages 224–231, 2000.

[9] Silvia Breu and Jens Krinke. Aspect mining using event traces. In *Auto. Soft. Eng. Conf.*, 2004.

[10] John Carroll and Ted Briscoe. High precision extraction of grammatical relations. In *7th International Workshop on Parsing Technologies*, Beijing, 2001.

[11] Keith Chan, Zhi Cong Leo Liang, and Amir Michail. Design recovery of interactive graphical applications. In *ICSE '03: 25th International Conference on Software Engineering*, pages 114–124. IEEE Computer Society, 2003.

[12] Mark C. Chu-Carroll, James Wright, and Annie T. T. Ying. Visual separation of concerns through multidimensional program storage. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 188–197, New York, NY, USA, 2003. ACM Press.

[13] Cynthia L. Corritore and Susan Wiedenbeck. Mental representations of expert procedural and object-oriented programmers in a software maintenance task. *Int. J. Hum.-Comput. Stud.*, 50(1):61–83, 1999.

[14] William G. Griswold, Yoshikiyo Kato, and Jimmy J. Yuan. Aspect browser: Tool support for managing dispersed aspects. In *Workshop on Multi-Dimensional Separation of Concerns*, 2000.

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

[16] Donald Hindle. Noun classification from predicate-argument structures. In *Meeting of the Association for Computational Linguistics*, pages 268–275, 1990.

[17] Mik Kersten and Gail C. Murphy. Mylar: a degree-of-interest model for ides. In *AOSD '05: 4th International Conference on Aspect-oriented Software Development*, pages 159–168. ACM Press, 2005.

[18] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP*, 2001.

[19] Andrew J. Ko, Htet Aung, and Brad A. Myers. Eliciting design requirments for maintenance-oriented ides: a detailed study of corrective and perfective maintenance tasks. In *Int. Conf. of Soft. Eng.*, 2005.

[20] Taku Kudo and Yuji Matsumoto. Chunking with support vector machines. In *NAACL 2001*, 2001.

[21] Kazimiras Lukoit, Norman Wilde, Scott Stowell, and Tim Hennessey. Tracegraph: Immediate visual location of software features. In *ICSM '00: International Conference on Software Maintenance*, page 33. IEEE Computer Society, 2000.

[22] Andrian Marcus. *Semantic-driven program analysis*. PhD thesis, Kent State University, 2003. Director-Jonathan I. Maletic.

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

[24] Amir Michail. Browsing and searching source code of applications written using a gui framework. In *ICSE '02: 24th International Conference on Software Engineering*, pages 327–337. ACM Press, 2002.

[25] George Miller. Wordnet: a lexical database for english. In *Communications of the ACM*, pages 39–41, 1995.

[26] Gilad Mishne. Source code retrieval using conceptual graphs. In *Proceedings RIAO*, 2004.

[27] Jason Baldridge Tom Morton and Gann Bierner. OpenNLP maxent package in Java. *http://maxent.sf.net*.

[28] Gail Murphy, Mik Kersten, Martin Robilland, and Davor Cubranic. The emergent structure of development tasks. In *ECOOP*, 2005.

[29] Fernando C. N. Pereira, Naftali Tishby, and Lillian Lee. Distributional Clustering of English Words. In *Meeting of the Association for Computational Linguistics*, pages 183–190, 1993.

[30] Adwait Ratnaparkhi. A maximum entropy part-of-speech tagger. In *Proc. of the Empirical Methods in Natural Language Processing Conference*, 1996.

[31] M. Narayanaswamy K. E. Ravikumar and K. Vijay-Shanker. Beyond the clause: Extraction of phosphorylation interactions from medline abstracts. *Bioinformatics, Suppl 1*, 21(1):i319–i328, 2005.

[32] Martin P. Robillard and Gail C. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *International Conference on Software Engineering*, 2002.

[33] A. Sampaio, N.Loughran, A. Rashid, and P. Rayson. Mining aspects in requirements. In *Workshop on Early Aspects*, 2005.

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

[35] David Shepherd, Tom Tourwe, and Lori Pollock. Using language clues to discover crosscutting concerns. In *The 1st International Workshop on Modeling and Analysis of Concerns in Software at ICSE*, May 2005.

[36] Wojciech Skut and Thorsten Brants. A maximum-entropy partial parser for unrestricted text. In *Sixth Workshop on Very Large Corpora*, 1998.

[37] Peri Tarr, Harold Ossher, William Harrison, and Jr. Stanley M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *ICSE*, 1999.

[38] M. Torii and K. Vijay-Shanker. Using machine learning for anaphora resolution in medline abstracts. In *Proc. of Pacific Symposium on Computational Linguistics*, 2005.

[39] Wei Zhao, Lu Zhang, Yin Liu, Jiasu Sun, and Fuqing Yang. Sniafl: Towards a static non-interactive approach to feature location. In *ICSE '04: 26th International Conference on Software Engineering*, pages 293–303. IEEE Computer Society, 2004.