Analysing source code: looking for useful verb–direct object pairs in all the right places

Z.P. Fry, D. Shepherd, E. Hill, L. Pollock and K. Vijay-Shanker

Abstract: The large time and effort devoted to software maintenance can be reduced by providing software engineers with software tools that automate tedious, error-prone tasks. However, despite the prevalence of tools such as IDEs, which automatically provide program information and automated support to the developer, there is considerable room for improvement in the existing software tools. The authors' previous work has demonstrated that using natural language information embedded in a program can significantly improve the effectiveness of various software maintenance tools. In particular, precise verb information from source code analysis is useful in improving tools for comprehension, maintenance and evolution of object-oriented code, by aiding in the discovery of scattered, action-oriented concerns. However, the precision of the extraction analysis can greatly affect the utility of the natural language information. The approach to automatically extracting precise natural language clues from source code in the form of verb–direct object (DO) pairs is described. The extraction process, the set of extraction rules and an empirical evaluation of the effectiveness of the automatic verb–DO pair extractor for Java source code are described.

1 Introduction

A large portion of the software life cycle, and thus the cost of software development, is attributed to maintenance and evolution tasks such as modifying the application to meet the new requirements or fix faults [1]. To ease the burden of software maintenance, software tools have been developed for program understanding, navigation, testing, debugging and refactoring. Many of these tools are driven by an underlying analysis of the program code, such as call graph and type analysis. Although this traditional program analysis can provide detailed information about the program, often this information is not sufficient to assist the user in answering high-level questions that software maintainers want answered (e.g. Which class represents concept X? or Where is there any code involved in the implementation of behavior Y?) [2].

Through examination of large, complex program source codes, we have observed that there are many natural language clues in program literals identifiers and comments which can be leveraged to increase the effectiveness of many software tools in answering developers' questions [3, 4]. Natural language analysis of source code complements traditional program analysis because it analyses different program information than traditional program analysis. We call this kind of analysis Natural Language Program Analysis (NLPA), since it combines natural language processing techniques with traditional program analysis to extract natural language information from a program.

© The Institution of Engineering and Technology 2008
doi:10.1049/iet-sen:20070112
Paper first received 26th September 2007

The authors are with Computer and Information Sciences, University of Delaware, Newark, DE 19716, USA
E-mail: pollock@cis.udel.edu

Using NLPA, we have developed integrated tools that assist in performing software maintenance tasks, including tools for program understanding, navigation, debugging and aspect mining [3–5]. Thus far, we have focused on NLPA tools that identify scattered code segments that are somehow related: whether it be to search through the code to understand a particular concern implementation, to mine aspects or to isolate the location of a bug. Our existing NLPA tools combine program structure information such as calling relationships and code clone analysis with the natural language of comments, identifiers and maintenance requests. In order to leverage this information about programs, the main challenge is to precisely extract verb–direct object (i.e. verb–DO) pairs from appropriate locations in source code to build more effective software tools.

This paper presents the challenges of extracting useful natural language clues from source code and our approach for automatically discovering useful, precise verb–DO pairs from source code (The particular extraction rules are Java-specific, but the overall approach is general.). Specifically, this paper makes the following contributions:

- algorithms for automatically extracting verb information from program source code comments;
- a set of rules for extracting verb–DO pairs from program method signatures, with motivating examples and discussion;
- description of our implementation of the extraction process;
- empirical evaluation of the effectiveness of the automatic technique for precisely extracting verb-direct object pairs from Java source code.

The paper is organised as follows. Sections 2 and 3 introduce and motivate the extraction and the use of verb–DO pair information from program source code to improve the effectiveness of software tools. Section 4 describes the extraction process and individual rules. The evaluation methodology and results are presented in Section 5. We put our work into perspective with related work in Section 6. We summarise and discuss the future research plans in Section 7.

2 Verb–DO information in programs

There are many natural language clues that can be extracted from programs. For software maintenance tasks, action words are central, because most maintenance tasks involve modifying, perfecting or adapting the existing actions. Actions tend to be scattered in object-oriented programs, because the organisation of actions is secondary to the organisation of objects. Thus, to improve the tools used in software maintenance, we currently focus on extracting action words from source code.

During development, programmers attempt to realise their thoughts as source code. However, because they must operate within strict confines (i.e. writing code that compiles and executes correctly), programmers' ability to write human-readable code is limited. However, many developers attempt to make their code readable and follow similar patterns in their naming conventions. Our work leverages these common patterns and naming conventions to automatically extract useful natural language information from the source code.

Because we seek to identify and extract actions from source code, we search for verbs in source code. In a programming language, verbs usually appear in the identifiers of method names, possibly in part because the Java Language Specification recommends that 'method names should be verbs or verb phrases...' [6]. Therefore we have focused our initial extraction efforts on method names and the surrounding information (i.e. method signatures and comments).

Identifying the verb in a phrase does not always fully describe the phrase's action. To fully describe a specific action, it is important to consider the theme. A theme is the object that the action (implied by the verb) acts upon and usually appears as a DO. There is an especially strong relationship between verbs and their themes in English [7]. An example is (parked car) in the sentence 'The person parked the *car*.' Similarly, in the context of a single program code, often verbs such as 'remove,' act on many different objects, such as 'remove attribute', 'remove screen', 'remove entry', and 'remove template'. Therefore to identify specific actions in a program, we examine the DOs of each verb (e.g. the direct object of the phrase 'remove the attribute' is 'attribute').

We leverage natural language information for software maintenance by analysing the source code to extract action words, in the form of verb–DO pairs. A verb–DO pair is defined to be two terms in which the first term is an action or verb and the second term is a DO for the first term's action. We currently only analyse the occurrences of verbs and DOs in method declarations and comments, string literals and local variable names within or referring to method declarations. This paper focuses on the extraction of verb–DO pairs from method signatures, as our exploratory studies have shown that method declarations usually hold the most relevant action information, and because we can use traditional NLP to analyse variable names and especially comments. We now describe how we represent the extracted verb–DO pairs and some maintenance tools we have built based on this information.

In previous work, we designed a novel program model, the action-oriented identifier graph (AOIG), to explicitly represent the occurrences of verbs and DOs that we extract from a program, as implied by the usage of user-defined identifiers [4]. An AOIG representation of a program contains four kinds of nodes: a verb node for each distinct verb in the program, a DO node for each unique direct object in the program, a verb–DO node for each verb–DO pair identified in the program, and a use node for each occurrence of a verb–DO pair in a program's comment or code.

An AOIG has two kinds of edges: pairing edges and use edges. There exists a pairing edge from a verb *v* or DO *do* node to a verb–DO node when *v* and *do* are used together, that is, appear in the same sentence or phrase and interact, as (jump, hurdle) do in the phrase 'jump the hurdle'. For each use of a verb–DO pair in the program, there exists a use edge in the AOIG mapping the verb–DO node to the corresponding use node. Although a verb (or DO) node may have edges to multiple verb–DO nodes, a verb–DO node has only two incoming edges: one from the verb and one from the DO node involved in the relation.

Fig. 1 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 the locations in the source code where this pair occurs. We described how to build the AOIG with reasonable time and space costs using the open-source NLP components and open-source Java analysis components to extract the verb–DO information from both comments and method signatures, in a previous paper [4].

3 Using verb–DO information to improve software tools

We have used the actions (i.e. verb–DO pairs) found in source code to drive several maintenance tools. Below we describe the motivation for each tool and our strategy, as well as how our approach employs the extracted natural language information. On the basis of our experience, using natural language to improve maintenance tools is quite promising.

3.1 Concern location

The particularities of source code structure can frustrate a search engine's ability to find relevant code segments. For instance, a direct search over source code, such as the lexical search 'add auction', will not find the method add_entry (which actually adds an auction to a list). In our approach, we extract and build the AOIG and then search over this natural language representation of source code, which leads to more accurate search results. We use NLPA to determine that the method add_entry actually

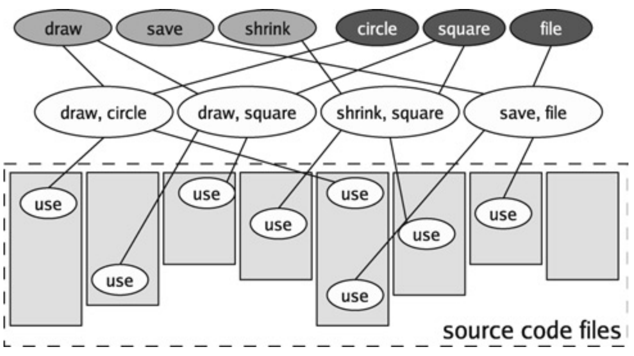

Fig. 1 *Example of an AOIG program representation*

'adds an auction' by analysing its parameter types (type AuctionInfo). If a user searches for the verb–DO pair 'add-auction' using our tool based on the AOIG, the NLP-based tool will only return methods using the verb–DO pair in the signature or comments, thus yielding a smaller, more precise result set than that provided by most other search techniques.

We have created a prototype code search tool, Find-Concept, that searches the AOIG instead of the unprocessed source code, often leading to more effective searches [3]. Find-Concept leverages the AOIG's structure and NLP techniques to assist the users in expanding their initial query into a more effective query (e.g. by using synonyms). This interactive Eclipse plug-in was designed to facilitate easier location of concerns within a code base inside the Eclipse environment. Find-Concept takes as the input a target concept in the form of a verb and an object and, after interaction with the user to expand the query, outputs a search result graph for the concept. In an experimental evaluation, Find-Concept performed more consistently and more effectively than a state-of-the-art competitor (a plug-in to Google Desktop) [3].

3.2 Aspect mining

To realise the benefits of aspect-oriented programming, developers must refactor active and legacy code into an aspect-oriented programming language. When refactoring, developers first need to identify refactoring candidates, a process called aspect mining. Humans mine using a variety of program-analysis-based features, such as call graph degree. However, the existing automatic aspect mining approaches only use single program-analysis-based features [8]. Thus, each approach finds only a specific subset of refactoring candidates and is unlikely to find candidates which humans find by combining characteristics. We have created a framework, called Timna, that uses machine learning to combine program analysis characteristics and identify refactoring candidates. Previously, we evaluated Timna and found that it was an effective framework for combining aspect mining analyses [5].

Most previous works in automatic aspect mining, including Timna, leveraged traditional program analyses exclusively. To evaluate Timna's effectiveness, we annotated every refactoring candidate in a substantial Java program. We began identifying candidates using only traditional program analyses' clues, yet we found that often natural language clues reinforced the traditional program analyses' clues. For instance, two methods that are often called in the same method body are often refactoring candidates. These methods typically performed actions (indicated by the verb–DO pairs extracted from the method) which were opposites of each other, such as 'open' and 'close'. iTimna uses these traditional program analyses and natural language clues together to more effectively identify refactoring candidates. In our initial evaluation, iTimna (i.e. Timna with NLPA features added) reduced the misclassification rate by 38%, when compared with Timna [9].

4 Extraction of verb–DO pairs from code

Given the intended use of extracted verb–DO pairs for maintenance tool building, the goal of verb–DO pair extraction is to determine verb–DO pairs based on the identifier names and literals in method signatures and the corresponding method comments such that the extracted verb–DO pairs accurately represent the behaviour of the corresponding method's body. Our initial AOIG construction and the use of the AOIG for software tool improvement was based on simple extraction rules for verb–DO pairs [3, 4]. Although our evaluation studies showed improvements using NLPA with these simple rules, careful analysis of our experimental results and systematic inspection of several large software systems revealed that there were many more complex natural language relations embedded in the method signatures. This section begins by describing the challenges in extracting verb–DO pairs that accurately represent the intended behaviour of a method. We then present our algorithms for extracting pairs from comments and method signatures.

4.1 Challenges in extracting useful pairs

Extracting representative verb–DO pairs from method signatures and their comments is not an easily generalisable task. Through many developmental iterations, we found that by generalising our analysis methods too much, we failed to extract adequate pairs for several specific methods. Although our approach generally starts with the method header, it is often necessary to examine the parameters and class name. Furthermore, after partitioning the method header into separate words, it is sometimes necessary to use stemming and other natural language techniques to obtain the most useful forms of these words.

The most representative verb and DO for a given method can appear in any number of locations within a given method signature, including various positions within the method name, the class name or the formal parameter list. For instance, consider the methods `Parser.parse (String document)` and `Document.parse(int numLines)`. In both cases, the verb 'parse' most accurately describes the method's primary action. However, in the first example, the most accurate DO would be the parameter: 'document', and in the second example, the most accurate DO would be the class name: 'Document'. In the extraction process, these two methods would be analysed very similarly as they both have a one-word method signature and one parameter. This example demonstrates the kinds of issues that we encountered when trying to generalise a rule set to extract appropriate verb–DO pairs from methods.

4.2 Overall extraction process

Extracting verb–DO pairs and building an AOIG representation from source code involves analysing natural language (comments) as well as program structures (method signatures). We use the overall process illustrated in Fig. 2 to extract the verb–DO pairs for an object-oriented program. The paths to analyse the 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`, including the POS tagging, chunker and role assignment. Extracting pairs from method signatures is illustrated as the path that starts at `A` and invokes the pattern extractor. Once the pairs are

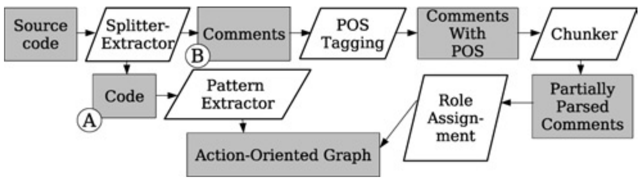

Fig. 2 *Overall verb–DO pair extraction process*

extracted, we use them to create the appropriate nodes and edges in the AOIG. The next subsections describe the paths A and B in detail.

4.3 Extraction 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 from comments, a POS tag (e.g. verb, adjective, noun etc.) is assigned to each word in each comment. Highly accurate (achieving precision ~97%) and efficient taggers [10, 11] 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. Although natural languages are notoriously ambiguous, chunking (also called robust parsing or partial parsing) can be done accurately [12, 13].

We detect DOs 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.

As an example, consider the comment 'Removes an item from a cart'. We would tag the comment (Remove$_{\langle v \rangle}$ an$_{\langle dt \rangle}$ item$_{\langle n \rangle}$ from$_{\langle p \rangle}$ a$_{\langle dt \rangle}$ cart$_{\langle n \rangle}$), partially parse the comment (Remove$_{\langle v \rangle}$ [an item]$_{\langle \text{Noun Phrase} \rangle}$ [from [a cart]$_{\langle \text{Noun Phrase} \rangle}$]$_{\langle \text{Prepositional Phrase} \rangle}$) and use pattern extraction to determine the verb-DO pair (Remove, item). To build the AOIG, we then construct a use node at that comment and attach it to the (Remove, item) verb–DO node, creating a node if necessary.

Each of the components in Fig. 2 (POS tagger, noun phrase chunker, role assignment etc.) can be done with accuracy over 90% even in complex texts [7, 14, 15]. 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 [15, 16]. In our implementation, we used the components from OpenNLP [17]. Although we are able to obtain the verb–DO pairs from comments, the usefulness of this information is an open question and depends on the client tool. We believe that method signatures will provide better information due to missing, outdated and incomplete comments in code.

4.4 Extraction from method signatures

To extract verb–DO pairs from the names appearing in method signatures, we perform the following analysis steps.

1. Identify individual words used in the method signature and note their position and role within the method signature:
(a) split method name (e.g. writeToFile → Write To File [18, 19]);
(b) perform POS tagging on split method name;
(c) perform chunking on tagged method name.
2. Locate and extract the set of verbs from the method signature.
3. Given the set of verbs and their identified position in the method signature, locate possible DOs for each verb and extract a set of verb–DO pairs to represent the behaviour of the method body.

To analyse the method signatures for verb–DO pairs, we developed a set of extraction rules, based on identifying the verb first, and then forming the verb–DO pairs by locating the possible DOs based on the position of the verb in the method signature. This two-step extraction process allows for a comprehensive rule set that attempts to avoid errors from over-generalisation.

In the extraction process, we do not apply the rules exclusively. If more than one extraction rule applies, we apply each rule that fits the method signature to extract a verb–DO pair, and thus our analysis returns a set of verb–DO pairs for that method. We believe that it is important to ensure that we do not miss any appropriate verb–DO pairs for a given method signature. Thus, we conservatively extract a set of verb–DO pairs when a signature fits more than one rule. The client application (e.g. concern location tool or aspect miner) ultimately determines the most appropriate pair(s) and how to handle superfluous pairs from the extracted set based on the way the verb–DO pair information will be used by the client application.

We first describe the location and extraction of verbs from method signatures, followed by how to locate the DOs for these verbs. Our analysis of method signatures resulted in four main cases for the identification of verbs in method signatures, summarised in Table 1. We indicate the extracted verbs in italics and direct objects in bold. When the verb occurs in the beginning of the method name, such as 'add' in the method signature `Song.add Track(SongTrack object)`, we consider the verb to be a Left-Verb. A verb is called a Right-Verb when the verb occurs at the rightmost end of the method name, such as 'Performed' in the method signature `SelectColorsPanel.dynamicRadioButtonAction Performed(ActionEvent evt)`. Some words in the English language occur frequently in code and pose somewhat of a challenge for extraction in our model. Examples of these words are 'is', 'fire', 'on', 'init' and 'run'. Our solution to this problem is to handle each of these cases in their own specific manner because we found that their method name patterns are very consistent within each case. For example, consider the method `fireDirectoryUpdate()` in the `WebdavConnection class` with the method signature, `WebdavConnection. fireDirectory Update()`. In this case, the main verb is 'fire' but the verb 'update' provides a much more accurate description of the method's behaviour. Verbs that follow a specific naming pattern like those above are called

Table 1: Locating verbs in method signatures

Class	Verb	DO	Example
left–verb	leftmost word in method name	rest of method name	public URL *parse***Url**( )
right–verb	rightmost word in method name	rest of method name	public void **mouse***Dragged*( )
special–verb	leftmost word in method name	specific to verb	public void **HostList**.on*Save*( )
unidentifiable-verb	'no verb'	method name	public void **message**( )

`Special-Verb` cases, and we handle each case on an individual basis.

Some method names do not contain a verb and are therefore very difficult to analyse in terms of extracting a verb–DO pair. Consider the method `error` in the `JFtp` class. In this case, the verb could be any number of words, such as ‘handle’, ‘log’, ‘throw’, or ‘catch’. For this reason, we currently call these cases Unidentifiable-Verb. More examples of each class are given in Table 1.

The location of the direct object of a verb–DO pair is determined differently based on whether the verb is Left-Verb, Right-Verb, Special-Case or Unidentifiable-Verb. The handling of each case is described below.

4.4.1 Left-Verb: Consider the method signature `Song.addTrack(SongTrack object)` that we previously classified as Left-Verb. The extraction rule is based on the remainder of the method signature. If the DO can be clearly located in the method name based on POS tagging and pattern matching, the verb–DO pair is created only using the method name. For example, in the case of `addTrack`, the DO indeed can be located in the method name, and in this example, the extracted verb–DO pair is ⟨add, track⟩. If this rule does not apply, then we examine the formal parameter names. If the method has parameters, then the DO is extracted from the first formal parameter name. If the method has no parameters, then the DO is extracted from the method's class name. These are three of the six extraction rules for the Left-Verb case of method signatures. The remaining rules deal with other combinations of parameters, class names, and return types. To illustrate our extraction rules for method signatures, all six rules with examples are given for Left-Verb in Table 2.

4.4.2 Right-Verb: The extraction of direct objects for the Right-Verb case is similar to the Left-Verb case. Consider the `SelectColorsPanel.dynamicRadioButtonActionPerformed(ActionEvent evt)` method signature as an example. If the direct object can be identified in the method name, it is the only part of the signature used to extract the verb–DO pair. Otherwise, the parameters and class name are inspected (in that order) and used for extraction. In this example, the extracted verb–DO pair is ⟨performed, dynamic radio button action⟩. There are only three rules for extracting verb–DO pairs from Right-Verb signatures.

4.4.3 Special-Verb: `Special Verb` signatures require the most extraction rules because each special verb is handled differently. Overall, there are two main categories of extraction rules. The first set corresponds to the words that have special meaning in Java different from their English language meaning. This would include ‘fire’, ‘on’ and ‘start’. In these special verb cases, the method name can usually be analysed to provide a more accurate verb–DO pair than using the special verb. Consider the `WebdavConnection.fireDirectoryUpdate()` method signature. This method name follows the pattern of ‘fire’ followed by either an event or some combination of verbs, nouns, adverbs or adjectives. Specifically, this method pattern is ‘fire’ followed by a noun modifier and a verb-derived noun. In the ‘fire’ case, we extract a verb–DO pair by dropping ‘fire’ from consideration in the method name and re-analysing the method name to determine whether it is Left-Verb, Right-Verb, Special-Verb or Unidentifiable-Verb. In this example, ‘directoryUpdate’ is a Right-Verb, and the extracted pair is ⟨update, directory⟩ using the Right-Verb extraction rules. In addition, for the ‘fire’ special verb, we supplement the possible extracted set of verb–DO pairs with a pair of the form ⟨notify, class name⟩ because the purpose of ‘fire’ methods in Java is to notify listeners. Therefore another pair ⟨notify, Webdav Connection⟩ is added to the result set for this method.

The second set of Special-Verb cases includes words that are used in method names that frequently follow a pattern lending itself to extracting a verb that is more descriptive than the special verb observed in the name. Examples of these words often found in method signatures are ‘is’, ‘run’ and ‘do’. In these cases, it is often necessary to look outside the method name or artificially supplement the method name with additional information to obtain the most accurate verb–DO pair set.

Table 2: Extraction rules specific to Left-Verb method signatures

Form of signature	Example	Extraction process	Extracted pair
Standard left verb and DO in method name	public int getPages() in class Document	Extract verb and DO from method name	⟨get, pages⟩
No DO in method name, has parameters, no return type	public void removeFromMenu(String name) in class RosterTree	Extract verb from method name and DO from parameter name	⟨remove, name⟩
No DO in method name, no parameters, no return type	public void ActionWriter.divide() in class Writer	Extract verb from method name and DO from class name	⟨divide, writer⟩
Create Left-Verb, has return type	public List createWordSet(String text) in class ChatSplitPane	Extract verb from method name and DO from return type	⟨create, list⟩
No DO in method name, has parameters, return type is more specific than parameters in type hierarchy	public PerformanceEvent lookup(int key) in class MasterPlayer	Extract verb from method name and DO from return type	⟨lookup, performance event⟩
No DO in method name, has parameters more specific than return type in type hierarchy	public int convert(Song b) in class Midi	Extract verb from method name and DO from parameter type	⟨convert, song⟩

The method `IReportCompiler.run()` is a good example of the second set of Special-Verb cases. In this example, the pair ⟨run, IReportCompiler⟩ is first extracted based on Left-Verb analysis. However, since the class name contains a verb-derived noun, 'Compiler', we extract another verb–DO pair using the class name. In this case, we justify this pair because the verb 'run' without any additional information means that the main operation of the class is being carried out. In this case, 'compiling an IReport' is the main operation of the class and therefore by converting the verb-derived noun back to a verb and taking the rest of the class name as a direct object, we extract the pair ⟨compile, ireport⟩.

These extraction rules for specific Special-Verb cases cannot be generalised for other Special-Verb cases. We created the Special-Verb category to handle the unique cases that occur frequently and exhibit very consistent naming patterns which lend themselves to customised extraction processes.

4.4.4 Unidentified-Verb: Method signatures with no discernable verb pose a great challenge to our extraction process. Unfortunately, the lack of necessary NLP information makes it impossible to extract a descriptive verb–DO pair. Looking again at the `JFtp.error(String msg, Throwable throwable)` method, one can see that it is impossible to determine what the programmer intended as the verb. Therefore in the case of Unidentified-Verb, we attach a pair of the form ⟨implicit, method name⟩; in this case, ⟨implicit, error⟩. We treat all cases of Unidentified-Verb in a standardized way to allow the client program to easily detect and process them.

4.5 Finalising the set of reported verb–DO pairs

After the extraction process generates a set of verb–DO pairs for each method, based on comments and method signatures, we revise the final reported set by applying a small set of filtering rules. Any verb–DO pair in which the reported verb (or DO) is an abbreviation of another verb (or DO) that also appears with the same DO (or verb) is removed. For instance, the pair ⟨precompose, src⟩ is removed from the reported set that already contains ⟨precompose, source⟩. Lastly, a verb–DO pair that contains a DO that matches another verb–DO pair with the same verb and a more descriptive DO will be removed. For example, ⟨create, UI⟩ is removed because we also extracted ⟨create, Component UI⟩. Our experience has shown that a set of one to four verb–DO pairs, typically one, is extracted automatically for each method.

4.6 Analysis of extraction costs

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. Let N be the largest number of comments in a method. Method signatures are quicker to process, and their processing time can be bounded by a small constant P. Creating the AOIG for a program requires examining every method signature and any comments associated with each method only once, (O(m)). Since both extraction subprocesses only require a small, constant time, the time costs can be reduced from $O((L * N + P) * m)$ to O(m).

The space for extraction is in terms of the representation of the results, namely the AOIG. 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 DO 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.

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 10s per source file for a file with about ten methods and ten multi-sentence comments. The AOIG construction process spent about 1 s analysing the method signatures and about 9 s analysing 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 2.40 GHz).

5 Evaluation study

We implemented our verb–DO pair extractor, which we call AOIGBuilder, in Java as a plug-in to the Eclipse Development Environment. Therefore once a user has created (or imported) a project in Eclipse, the user can easily trigger the AOIGBuilder for that project. Another advantage of implementing the AOIGBuilder within Eclipse is the ease with which other software tools can access the AOIG.

To evaluate the effectiveness of the automatic verb–DO extraction approach, we designed and conducted a study that focused on gaining insight into how well the automatically extracted verb–DO pairs represent the behaviour of the corresponding methods. We asked three different human subjects to examine each method and produce a set of verb–DO pairs that they believed best represented the behaviour of the method. We then computed precision and recall for our automatically extracted verb–DO pairs by comparing with the manually generated verb–DO pairs used to capture the behaviour of the methods. We now explain the details of our experimental methodology.

5.1 Methodology

We strove for simplicity in our experimental design in order to facilitate the interpretation of the results. We chose to manipulate three independent variables: targeted methods of extraction, the subject applications from which the methods were selected and human subject annotators. We refer to our human subjects as annotators in the rest of this paper.

In selecting target systems for our study, we applied a number of criteria intended to increase the validity of our results. We selected six open-source Java applications. To avoid small systems, we chose systems with at least 10 000 non-commented lines of code, 900 methods and 175 class declarations. We chose systems with at least 10 000 downloads so we would have systems with an active user community. Lastly, to avoid antiquated code, all of our systems were registered with an open-source repository within the past six years. The size characteristics for each subject application are displayed in Table 3. We calculated these characteristics using the Eclipse RefactorIT plug-in [20].

DGuitar is a viewer for Guitar Pro tablature files in any operating system. DrawSWF is a simple drawing application that generates animated SWF Files, which redraws everything you have drawn with the mouse in the editor. The Jeti Applet is a Java Jabber client that supports most chat features such as file transfer, group chat, emoticons and formatted messages. JFTP has a convenient user interface for performing ftp operations, generally for file and/or directories transfer. JRobin is a 100% pure Java alternative to RRDTool, which is an industry standard for generating charts and graphs. PlanetaMessenger is an Instant Messenger with support for all well-known instant messenger networks including ICQ, MSN, AIM, Yahoo! and Jabber, through a powerful plug-in engine.

We randomly selected 50 methods to be annotated from each of the six programs, for a total of 300 methods. We were careful not to include constructors or getter and setter methods, as these two method categories are trivial for the AOIGBuilder to extract verb–DO pairs. Constructors occur in most classes and, as per the rules of Java, have to match the class name. Since classes represent objects, class names rarely contain a verb. For this reason, the verb–DO pair we create for every constructor is ⟨create, classname⟩. Similarly, getter and setter methods occur frequently and are trivial for AOIGBuilder to analyse and extract appropriate verb–DO pairs. For instance, for a method `List.getLength()`, the only appropriate pair is ⟨get, length⟩. Since all ‘get’ and ‘set’ methods follow this pattern, a representative verb–DO pair is almost always extracted.

The annotators were chosen through personal contacts of the authors. None of the authors served as annotators. To qualify for the study, participants had to have Java programming experience. The participants were a mix of advanced and intermediate Java programmers and had no knowledge of the AOIGBuilder rules. Because the characteristics of the participants did not constitute an independent variable in this study, we did not record such characteristics.

5.1.1 Annotation procedure: Each method was assigned to three different annotators. Therefore, the 50 randomly selected methods from each program were selected as five randomly selected sets of ten methods, giving a total of thirty ten-method sets to be annotated. Each of the six annotators were assigned to annotate 15 sets of 10 methods, randomly selected from the 30 possible sets of methods. Because one of the annotators annotated a wrong set of methods, 40 of the 300 methods were annotated by only two annotators.

Table 3: Characteristics of subject applications

Program	Lines of code	Number of methods	Number of classes
DGuitar	13 422	1030	212
DrawSWF	27 674	2352	285
JetiApplet	38 130	2369	773
JFTP	34 344	2042	283
JRobin	19 524	1631	254
PlanetaMessenger	11 105	948	176

Each annotator was instructed to examine each of their assigned methods and *try to characterize their function and behavior using a ⟨verb, direct object⟩ pair* (Statements in italics are excerpts from the instructions given to the annotators). They were told to create a most representative verb-DO pair and any other possible pairs that they believed accurately represented the behavior of the method, if they believed that multiple pairs accurately reflected what the method achieved. The annotators were instructed to *use everything at their disposal (method signature, method body, parameters, class name etc.)*.

5.1.2 Measuring effectiveness: We measured the effectiveness of the AOIGBuilder by computing precision and recall and also performing a qualitative study of the automatically extracted verb–DO pairs. High precision means the extracted verb–DO pair set contains few pairs that misrepresent the associated methods, whereas high recall implies that most of the verb–DO pairs that are closely representative of the methods are included in the automatically extracted pair set.

To compute precision and recall, we needed to define what it means for a ‘match’ between the AOIGBuilder extracted verb–DO pairs for a given method *m* and the verb-DO pairs provided by the three human annotators of method *m*. We determine that an AOIGBuilder verb–DO pair matches an annotator's pair if they both indicate the same behaviour for the underlying method. Because of the use of synonyms, variations in verb forms and other complications, it is difficult to determine a binary relation of match/no-match objectively. Thus, we defined a score for each match. A verb–DO pair vo_1 and verb–DO pair vo_2 are assigned a score of 1 if there is a very close match of the representation of the underlying method's behaviour (e.g. ⟨make, Menu⟩ and ⟨create, Menu⟩); a score of 0.5 if there is a partial match, and a score of 0 if pairs vo_1 and vo_2 indicate very different behaviour. To score the matches between AOIGBuilder's pairs and the annotators' pairs for each method, three of the authors examined the pairs and agreed on the score assignment, keeping in mind the use of synonyms, verb forms and intended meaning, without trying to guess what the annotator had in mind. For example, we did not consider the verb ‘equals’ extracted by AOIGBuilder to match the intent of the annotator's extracted verb ‘assess’.

After scoring each annotator's pairs with respect to the AOIGBuilder's pairs for the same method, we computed recall and precision as follows. For precision, we needed a measure of the number of correctly matched AOIGBuilder pairs divided by the total number of extracted AOIGBuilder pairs. We considered AOIGBuilder's output to be a good representative verb–DO pair for a given method if an AOIGBuilder pair matched with any of the annotators (Different people can look at a method differently or use different terms to describe a method's behaviour. But, if at least one annotator's interpretation matches AOIGBuilder's verb–DO pair, then we cannot say AOIGBuilder incorrectly represents the method's behaviour.). Let *M* be the set of methods we inspected. Given a method m_i, let AOIG(m_i) be the verb–DO pairs extracted by AOIGBuilder. Let $a_{i,1}, a_{i,2}, \ldots, a_{i,3}$ be the pairs extracted by the three annotators for the method m_i (Note that we use three annotators for our explanation, but the computation uses scores from only two annotators for those cases where there were only two annotators.). The MatchScore($a_{i,j}$) is the score of 0, 0.5 or 1 assigned to the match between AOIG(m_i) and $a_{i,j}$.

$$\text{precision} = \frac{\sum_{i=1}^{|M|} \max(\text{MatchScore}(a_{i,1}), \text{MatchScore}(a_{i,2}), \text{MatchScore}(a_{i,3}))}{\text{total number of pairs extracted by AOIGBuilder}} \quad (1)$$

$$\text{recall} = \frac{\sum_{i=1}^{|M|} \sum_{j=1}^{3} \text{MatchScore}(a_{i,j})}{\text{total number of annotations}} \quad (2)$$

where

$$\text{total number of annotations} = \sum_{i=1}^{|M|} \sum_{j=1}^{3} \delta(a_{i,j}) \quad (3)$$

and

$$\delta(a_{i,j}) = \begin{cases} 1 & \text{if } j\text{th annotation was available for method } m_i \\ 0 & \text{otherwise} \end{cases}$$

After we computed precision and recall, we closely examined about 20 methods where there was a low score for matches between AOIGBuilder and annotators. In particular, we examined the method signature, comment and body to characterise the cause of the mismatch.

5.2 Threats to validity

The set of methods that we used for comparison of humans and automatically generated verb–DO pairs is a threat to validity because the methods might favour the verb–DO pair automatic extraction process. To minimise this threat, we randomly selected the methods from a large pool, and across multiple programs, and eliminated the known trivial cases from our study (getters, setters and constructors). The scoring of the matches used to determine precision and recall is a threat to validity, as it was difficult to objectively determine the matches between the annotators and the automatically generated verb–DO pairs, and a set of three authors made the determination. To minimise this threat, the three authors had to come to agreement, and the scoring was done in one continuous session to provide consistency among scoring similar pairs from different methods. We believe that the subjectivity of the matches could be partially reduced by providing more specific instructions to the human annotators; however, we did not want to bias them in their judgement by too specific instruction.

The selection of human subjects as annotators is a threat to validity as they are all graduate students, albeit Java developers. Since we used six Java programs in our study, it is possible that our results cannot be generalised to all Java applications or programs in other languages. To maximise the generality, we used reasonably sized, popular open-source applications.

5.3 Results and discussion

Based on our comparison with human annotators, our computed precision for the AOIGBuilder was 57% and recall was 64%. We take these results to indicate a higher degree of success than the percentages may suggest. First, recall that we purposely did not include the setters, getters or constructors for evaluation purposes because we felt that the inclusion might inflate our numbers. Thus, the results presented here represent the accuracy after elimination of many of the simpler cases. A second point to note is that a precision of 57% indicates that, on average, we get at least a partial match for the methods and hence we get at least the verb or DO correct. In fact, there were only six cases out of 295 where the AOIGBuilder's output was deemed to be inappropriate (i.e. given a score of 0 among all annotators). Third, we applied fairly stringent standards in determining matching scores. As noted above, there are many cases where the output was not an exact match, although we could claim that the annotator intended the same meaning and that such an annotator would have in all likelihood considered AOIGBuilder's output to be appropriate. Finally, there were a few cases where we marked AOIGBuilder's verb–DO pair as not matching an annotator's pairs even though the annotator's DO matched the verb reported by AOIGBuilder, or vice versa.

For these reasons, we are encouraged by the results obtained. Further, an error analyis revealed the areas for improvement of verb–DO extraction as well as raised issues that we find to be instructive for using natural language information in software maintenance tasks.

One of our first conclusions was that, in a few cases, our system takes the action word to be words that are not even verbs. Examples include the system's choice of 'hash' as the verb in the extracted verb-DO pair from a method named `hashCode()` (DGuitar) and the extraction of 'jb' as the verb from the method `jb-Init()` (JetiApplet). Additionally, there were other cases where 'tag' and 'string' (`stringLengthMB()` in DrawSWF) were chosen as verbs. Although these words can sometimes serve as verbs, in the programming context, they are more likely to serve as nouns. Future versions of our system can benefit from this analysis. Currently, we are mining a large number of programs to learn word usage, and feeding back information into the extraction process.

The last example also shows another problem for the current system. In this example, the method name, `stringLengthMB`, does not contain any verb. The annotators read the body of the method to annotate this method with the pair ⟨return, string length⟩. This information is fairly obvious from the last statement of the method, but we currently do not extract information from the body of the methods. A similar example is the method `version()` (DGuitar) which is also annotated by the subjects as ⟨return, version (number)⟩.

The need to look at the body of the method can also be seen from other examples where the method signature is not very informative and sometimes even misleading. An example of the latter situation is in the method `displayPiece()` (DGuitar) where a quick inspection of the body reveals that ⟨set, tracks⟩ would be more appropriate. In this case, our method extracted the verb as 'display' and the DO as 'piece'. Similarly, our system's output was considered erroneous in our evaluation for the method `performAction()`, where annotators marked the verb–DO pair as ⟨find, duration⟩ based on an inspection of the body.

It would seem to be a difficult task in general to analyse the body and decide which, among the various pairs that can be extracted from the body, should be used for the entire method. But observations from these cases reveal that although this may seem to be difficult, in general, there seem to be cases for which some patterns can be exploited. For instance, when the method is a void method, and one of the last statements in the method is an assignment to a field of the class to which the method belongs (i.e. an assignment to 'this. ⟨field name⟩'), then the annotators have often used 'set' or 'assign' as the verb and information extracted from the field name for the DO. Examples include annotations of ⟨set, button color⟩ for `singleRadioButtonActionPerformed()` or ⟨set, tracks⟩ for `displayPiece()`.

In a few cases, the annotators marked a method as ⟨do, nothing⟩ since the method body was empty. Since we extract from method signatures, as an example, the system's extraction of ⟨execute, Information Query⟩ from `execute(InforQuery, Backend)`(JetiApplet) was marked incorrect. In other cases, our system's output was marked incorrect, although the verb or DO extracted was almost synonymous with those chosen by some annotators. These include verbs such as 'die' chosen by our system and annotator preferring 'exit' or the AOIGBuilder's 'fill' when an annotator chose 'enter'. Although we can see the clear relationship between the two (filling or entering data in a table entry), we chose to label this as incorrect since the two verbs are generally not synonyms. We had a few other examples in our evaluation (e.g. cancel against remove, test against equals, print against show) of verbs as well as objects (dialog against window).

6 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 [21]. Our work does not seek to link code to design, only to facilitate browsing the existing features, and our technique is automated, whereas this work was semi-automated. Researchers have also used conceptual graphs to represent programs, for use in code retrieval [22]. 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 [23, 24]. Baniassad and Clarke [23] created one of the first techniques to use NLP to mine for aspects in requirements, the Theme approach, which is semi-automated. Sampaio et al. [25] later created a technique for automatically identifying aspects in requirements, called EA-Miner. Both works have served as inspiration for our approach, but the nature of analysing requirements as opposed to source code has led to notably different approaches. The focus of EA-Miner 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 [24].

IR technology has been applied to the source code to search for relevant code segments, and a number of IR-based tools exist [26–28]. IR technology creates its own definition of related words by performing an analysis on each word's context and frequency. Latent semantic indexing (LSI) is one IR technology that has been applied to program comprehension [29]. LSI attempts to find related words by deducing 'meaning' from the context in which words in a system are used. However, the quality of the inferred semantics of words depends on the distribution and frequency of the used words, which can vary widely from system to system. Information retrieval technology does not address natural language issues (e.g. synonyms and morphological forms). A developer whose query fails because of this limitation will broaden the search terms, leading to large result sets (i.e. low precision) that are difficult to understand. In spite of its apparent shortcomings, researchers have successfully applied IR to locate concepts [19, 30] and reconstruct documentation traceability links in source code [31, 32].

7 Conclusions and future directions

In this paper, we presented strategies for extracting precise verb–DO pairs from method signatures and comments, with the goal of identifying verb–DO pairs that accurately represent the behaviour of each method. Overall, our evaluation study indicated that our current techniques can obtain 57% precision and 64% recall. The extraction of natural language information such as verb–DO pairs from source code has been shown to improve the effectiveness of various software maintenance tools including program search and navigation tools, debuggers and refactoring tools.

Analysis of our data from the human annotators for the 300 methods suggests various improvements to our AOIGBuilder. Thus far, we have focused on the extraction and the use of verb–DO pair information from source code. We plan to examine the alternate natural language clues, such as indirect objects, which we have found could be useful. We also plan to evaluate the use of our current AOIGBuilder within the context of several client tools.

8 Acknowledgment

This material is based upon the work supported by the National Science Foundation under Grant No. CCF-0702401.

9 References

1 Erlikh, L.: 'Leveraging legacy system dollars for e-business', *IT Professional*, 2000, **2**, (3), pp. 17–23

2 Sillito, J., Murphy, G.C., and Volder, K.D.: 'Questions programmers ask during software evolution tasks'. 14th Int. Symp. Foundations of Software Engineering (FSE), 2006

3 Shepherd, D., Fry, Z.P., Hill, E., Pollock, L., and Vijay-Shanker, K.: 'Using natural language program analysis to find and understand action-oriented concerns'. Int. Conf. Aspect-oriented Software Development (AOSD), 2007, pp. 212–224

4 Shepherd, D., Pollock, L., and Vijay-Shanker, K.: 'Towards supporting on-demand virtual remodularization using program graphs'. 5th Int. Conf. Aspect-Oriented Software Development (AOSD), 2006, pp. 3–14

5 Shepherd, D., Palm, J., Pollock, L., and Chu-Carroll, M.: 'Timna: a framework for combining aspect mining analyses'. 20th Int. Conf. Automated Software Engineering (ASE), 2005, pp. 184–193

6 Gosling, J., Joy, B., and Steele, G.: 'Java language specification'. Available online at: http://java.sun.com/docs/books/jls/second_edition/html/names.doc.html accessed September 2006

7 Carroll, J., and Briscoe, T.: 'High precision extraction of grammatical relations'. 7th Int. Workshop on Parsing Technologies, 2001, pp. 1–7

8 Ceccato, M., Marin, M., Mens, K., Moonen, L., Tonella, P., and Tourwe, T.: 'A qualitative comparison of three aspect mining techniques'. Int. Workshop on Program Comprehension, 2005, pp. 13–22

9 Shepherd, D., Pollock, L., and Vijay-Shanker, K.: 'Case study: supplementing program analysis with natural language analysis to improve a reverse engineering task'. 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE), 2007, pp. 49–54

10 Ratnaparkhi, A.: 'A maximum entropy part-of-speech tagger'. Empirical Methods in Natural Language Processing Conf., (EMNLP), 1996, pp. 133–142

11 Brants, T.: 'Tnt: a statistical part-of-speech tagger'. 6th Conf. Applied Natural Language Processing, 2000, (Morgan Kaufmann Publishers Inc.), pp. 224–231

12 Skut, W., and Brants, T.: 'A maximum-entropy partial parser for unrestricted text'. 6th Workshop on Very Large Corpora, 1998

13 Abney, S.: 'Partial parsing via finite-state cascades'. Workshop on Robust Parsing, 8th European Summer School in Logic, Language and Information, 1996, (Cambridge University Press), pp. 337–344

14 Kudo, T., and Matsumoto, Y.: 'Chunking with support vector machines'. North American Chapter of the Association for Computational Linguistics (NAACL). Second meeting of the North American Chapter of the Association for Computational Linguistics on Language Technologies 2001, 2001, pp. 1–8
15 Ravikumar, M.N.K.E., and Vijay-Shanker, K.: 'Beyond the clause: extraction of phosphorylation interactions from medline abstracts', *Bioinformatics, Suppl 1*, 2005, **21**, (1), i319–i328
16 Torii, M., and Vijay-Shanker, K.: 'Using machine learning for anaphora resolution in medline abstracts'. Proc. Pacific Symp. Computational Linguistics, 2005
17 Morton, J.B.T., and Bierner, G.: 'OpenNLP maxent package in Java'. Available at http://maxent.sourceforge.net/
18 Shepherd, D., Tourwe, T., and Pollock, L.: 'Using language clues to discover crosscutting concerns'. Int. Workshop on Modeling and Analysis of Concerns in Software (MACS) at ICSE, 2005, pp. 1–6
19 Zhao, W., Zhang, L., Liu, Y., Sun, J., and Yang, F.: 'Sniafl: towards a static non-interactive approach to feature location', *Trans. Softw. Eng. Methodol. (TOSEM)*, 2006, **15**, (2), pp. 195–226.
20 Aqris Software: 'RefactoIT Plugin 2.5'. Available online at: http://www.refactorit.com/, accessed September 2006
21 Baniassad, E.L.A., Murphy, G.C., and Schwanninger, C.: 'Design pattern rationale graphs: linking design to source'. 25th Int. Conf. Software Engineering (ICSE), 2003, (IEEE Computer Society), pp. 352–362
22 Mishne, G.: 'Source code retrieval using conceptual graphs'. Recherche d'Information Assiste par Ordinateur (RIAO), 2004
23 Baniassad, E., and Clarke, S.: 'Theme: an approach for aspect-oriented analysis and design'. 26th Int. Conf. Software Engineering (ICSE), 2004, (IEEE Computer Society), pp. 158–167
24 Sampaio, A., Loughran, N., Rashid, A., and Rayson, P.: 'Mining aspects in requirements'. Workshop on Early Aspects in Int. Conf. Aspect-oriented Software Development (AOSD), 2005
25 Sampaio, A., Chitchyan, R., Rashid, A., and Rayson, P.: 'EA-Miner: a tool for automating aspect-oriented requirements identification'. 20th Int. Conf. Automated Software Engineering (ASE), 2005
26 Codase Inc.: 'Codase'. 2007, available at: http://www.Codase.com
27 Koders Inc.: 'Koders'. Available at: http://www.koders.com
28 Krugle Inc.: 'Krugle'. 2007, available online at: http://www.krugle.com
29 Maletic, J.I., and Marcus, A.: 'Supporting program comprehension using semantic and structural information'. 23rd Int. Conf. Software Engineering (ISCE), 2001, pp. 103–112
30 Marcus, A., Sergeyev, A., Rajlich, V., and Maletic, J.I.: 'An information retrieval approach to concept location in source code'. 11th Working Conf. Reverse Engineering (WCRE), November 2004, pp. 214–223
31 Antoniol, G., Canfora, G., Casazza, G., Lucia, A.D., and Merlo, E.: 'Recovering traceability links between code and documentation', *IEEE Trans. Softw. Eng. (TSE)*, 2002, 28, (10), pp. 970–983
32 Marcus, A., and Maletic, J.I.: 'Recovering documentation-to-source-code traceability links using latent semantic indexing'. 25th Int. Conf. Software Engineering (ICSE), 2003, pp. 125–137