Using Language Clues to Discover Crosscutting Concerns

David Shepherd
Computer and Information Sciences
University of Delaware
Newark, DE 19716
shepherd@cis.udel.edu

Tom Tourwé
Centrum voor Wiskunde en Informatica
P.O. Box 94079
1090 GB Amsterdam, NL
tom.tourwe@cwi.nl

Lori Pollock
Computer and Information Sciences
University of Delaware
Newark, DE 19716
pollock@cis.udel.edu

ABSTRACT

Researchers have developed ways to describe a concern, to store a concern, and even to keep a concern's code quickly available while updating it. Work on identifying concerns (semi-)automatically, however, has yet to gain attention and practical use, even though it is a desirable prerequisite to all of the above activities, particularly for legacy applications. This paper describes a concern identification technique that leverages the natural language processing (NLP) information in source code. Developers often use NLP clues to help understand software, because NLP helps them identify concepts that are semantically related. However, few analyses use NLP to understand programs, or to complement other program analyses. We have observed that an NLP technique called lexical chains offers the NLP equivalent of a concern. In this paper, we investigate the use of lexical chaining to identify cross-cutting concerns, present the design and implementation of an algorithm that uses lexical chaining to expose concerns, and provide examples of concerns that our tool is able to discover automatically.

1. INTRODUCTION

A software application is a collection of many different, but related, concerns. In order to ensure its understandability, maintainability and evolvability, developers try to isolate each concern in a separate entity, using structuring techniques offered by the programming language they use. For example, when programming in Java, developers use classes and interfaces as much as possible to represent concerns, and inheritance and aggregation to represent the relations between them.

However, the tyranny of the dominant decomposition states that no matter how well an application is decomposed into modular units, some functionality (or concern) will always cross-cut this decomposition [16]. Traditional structuring techniques do not suffice to clearly and cleanly separate all concerns in the software, and the implementation of some concerns will be spread across parts of the application. Such concerns are referred to as *cross-cutting concerns*.

In order to link related, but distant, source code entities, thereby improving the understandability of the software, developers often resort to naming and coding conventions. For example, several design pattern implementations [4] span multiple classes and methods, but can be related through their (very specific) names, such as `acceptVisitor` or `addObserver`. Our key insight is to apply NLP to exploit this kind of semantic relationship (i.e., equivalence) as well as other types of semantic relationships between words.

In this paper, we report upon an experiment where we tried to identify cross-cutting concerns in existing source code, by exploiting the natural language clues that the developers left behind. In particular, we use a natural language processing technique called *lexical chaining* [9] to identify groups of "semantically" related source code entities, and we evaluate whether those groups represent cross-cutting concerns.

Although the research area is still in its infancy, several techniques for concern identification have already been developed [1, 8, 14, 17, 18]. Only one of these techniques, i.e. identifier analysis [18], directly investigates naming conventions, although the Aspect Mining Tool and the Aspect Browser were also designed to partially exploit naming conventions [5, 7]. The major contribution of our technique is that it recognizes more complex relationships between words than previous tools. This advancement allows us to analyze not only identifiers, but also comments, in an effort to understand source code. Previous work with identifier analysis groups classes and methods based on common substrings appearing in their name. Slight syntactic variations in these names have a large impact on the results of the existing techniques. Even worse, classes and methods with names that are semantically equiv-

alent will never be grouped. Given that current-day software applications consist of millions of lines of code and are developed by teams of developers, we can expect that naming conventions are not strictly adhered to, but often contain variations. Relying on the underlying semantics of a word and not simply its lexical form, lexical chaining can overcome variations in naming conventions.

2. LEXICAL CHAINING

2.1 What are lexical chains?

Lexical chaining is the process of grouping semantically related words in (part of) a document into chains [9]. A *chainer* takes as input a text and groups every word in that text in a chain with closely related words also appearing in the text. It outputs a list of *chains* that each contain closely related words. Chains show how the text "hangs together as a whole" [6]. If lexical chains are denoted in a text, one can see how different *semantic topics*, defined as the word members of a chain, flow through the piece, by connecting the members of the chain sequentially.

An example of lexical chaining is presented in Table 1, where the **auction** and the *money* concerns are represented in the text by means of superscripts (all members of the *auction* concern are marked with a [1], *money* with a [2]). Both concerns occur in the first and third paragraph, and one chain contains both the related words "money", "bills", and "funds". The second paragraph contains no members of any chain, so we can assume that none of the two semantic topics was discussed in that paragraph.

We hypothesize that semantic topics correspond to a "concern" in text. In this case, the *money* chain corresponds to the concern of bills and the money needed to pay those bills, and the **auction** chain corresponds to the auction the parents conducted to raise funds.

2.2 How to compute lexical chains?

In order to compute lexical chains, we need to be able to calculate the *semantic distance*, or the strength of relationship, between two given words [2]. For example, there exists a strong relationship between a `novel` and a `poem`, both being literary works. A weaker relationship exists between a `novel` and a `thesis`, both being writings, which is a less specific relationship.

Researchers have shown that it is easy for humans to determine *semantic distances* between two, closely related words, and that they do so with reasonable consistency [12]. However, semantic distance is more difficult to determine computationally.

In order to compute the distance automatically, a database of known relationships between words, such as WordNet [2], is often used. The semantic distance between two words is then approximated by using the lengths of the relationships path between the two words in WordNet. The longer the length, the less related the two words are likely to be.

Besides the length of the relationship, knowing the part of speech of each word helps to calculate the *semantic distance*. For example, the words "address" and "destination" could be semantically related if both are used as nouns. However, in the sentence "Once he reached his final destination at the front of the room, he turned to address the crowd", the word "address" is used as a verb and thusly has an entirely different meaning, and should not be grouped with the word "destination". Programs such as *Qtag* (a part of speech tagger) can be used [19] to perform this primitive type of *word-sense disambiguation* [10].

A 9-year-old boy successfully underwent surgery Wednesday to remove most of a brain tumor he nicknamed "Frank", and which was the subject of an **online auction**[1] to help raise *money*[2] for *medical bills*[2]. Cells from the tumor, which had been treated with chemotherapy and radiation, will now be studied to determine if it is malignant. David Dingman-Grover, of Sterling, Virginia., went into surgery around 10 a.m. at Cedars-Sinai Medical Center, said Frank Groff . . . David named his tumor after Frankenstein's monster, who scared him until he dressed up as the fictional character for Halloween. His parents **sold**[1] a bumper sticker reading "Frank Must Die" on eBay to raise *funds*[2] for his treatment.

Table 1: Paragraphs with Chains

3. USING LEXICAL CHAINING FOR CONCERN MINING

We hypothesize that some lexical chains in a software application will correspond to high-level concerns. As explained earlier, developers are forced to leave natural language clues, due to the lack of adequate structuring techniques for cross-cutting concerns. These clues provide information about the functionality of the code. Therefore, if two regions of code contain similar words, used as the same parts of speech, it is likely that these two code segments are dealing with the same functionality.

However, because a single class and the methods it defines presumably also implement a concern, it is not sufficient simply to look for any combination of words. To find cross-cutting concerns we look for chains that have members with a high amount of scatter (i.e., the word members are from many different source files). We suspect that these chains will often correspond to high-level concerns that are scattered throughout code.

3.1 Relevant Word Selection

The source code of a software application consists of many different strings, of many different types, and in many different contexts. Many of those strings are not useful for deducing semantic relationships between pieces of code. Therefore, we focus on specific subsets of the strings, in order to avoid meaningless associations. Specifically, we only consider *comments* and *field, type* and *method names*, because we believe programmers tend to leave important semantic clues within these constructs.

Due to their primary purpose and inherent form, we process each of these constructs slightly differently:

- **Comments:** Comments are usually written in sentence or phrase form. Thus, we use speech tagger to tag words occurring in them.
- **Method Names:** We separate any method names containing any capital letters into separate words. For example, a method `goAndDoThatThing` consists of five words: *go, and, do, that* and *thing*. We separate the name, and then use a speech tagger on this phrase.
- **Field and Class Names:** We assume field and class names are nouns. We split these identifiers in the same way that we split method names.

We also filter out words that are unlikely to provide useful information, such as "and" and "the", using a *stop list* [9]. If a word appears on this list we do not use it to chain.

3.2 Lexical Chaining Algorithm

Our lexical chaining algorithm first retrieves all words from every source file in the whole program. It then starts building chains by processing every word. For each word, it finds the chain that is most closely related to that word, and adds the word to that chain. If no chain is closely related to that word, it starts a new chain with that word. After processing every word, we are left with a list of word chains of varying lengths.

This algorithm is computationally expensive, since a typical program consists of many words, and the time needed grows in proportion to the number of unique input words ($O(unique\_input\_words)$). The algorithm's runtime also grows with the length of the semantic distance that is considered. If the algorithm considers relationships up to distance 10, it has much worse performance than if it considers relationships up to distance 5.

In the current prototype, we use a low threshold for distances, because we are chaining over an entire program. This allows us to achieve reasonable run time for analysis while still preserving many interesting relationships between words. In the future, we plan to examine alternative approaches, such as chaining over only one source file at a time and comparing the chains generated by each file.

To further improve analysis, we employ caching techniques, to ensure that if a word has been chained previously and it appears again later in the program, we do not recompute the most closely related chain once again.

3.3 Inspecting the Results

We implemented a lexical chaining tool as an Eclipse plugin [3]. It currently automates all parts of the algorithm, including the word selection and lexical chaining. Upon running the tool, the user waits for the tool's completion, and then browses the chains which are presented in the Lexical Chain Viewer. This viewer presents a developer with all chains that are constructed, for a given program, and allows him to inspect the chain, as well as the source code entities that correspond to the words in the chain.

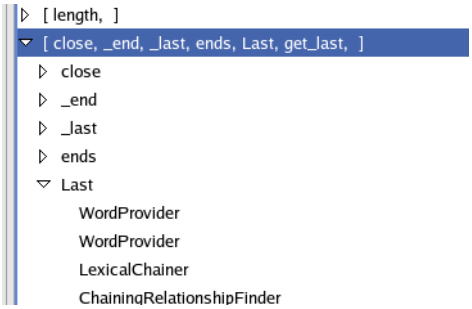

Figure 1: Lexical Chain Viewer

We present an example, from one of our case studies, in Figure 1. Here, a chain that includes the words "close", "_end", "_last", "ends", "Last" and "get_last" is highlighted. The chained words, represented as children of the chain, have references to the source file, and the specific location within that file, as their children. In the figure, the word "Last" has been expanded so that its children are visible.

4. CASE STUDIES

To evaluate the potential effectiveness of using lexical chains to locate concerns in source code, we used our tool on several source code bases. First, we used the non-GUI subset of our own plugin's code, consisting of approximately 1000 lines of code (without comments). Because we know this code very well, this allowed us to experiment with the technique, verify its results and fine-tune it where needed. Second, we used PetStore [15], a well-studied source code base that has been used as a subject for aspect mining before [8, 13], consisting of approximately 10K lines of code. We had little knowledge of the inner-workings of the PetStore source code, so we were not aware of most of its cross-cutting concerns. In order to illustrate how the tool is used and what it is capable of, we focus only on one chain for each case study.

4.1 Exploring our Lexical Chainer

Running the tool on its own code produced approximately 170 chains, and took approximately 12 minutes to run. When the tool finishes, it presents a list of chains. Within thirty seconds, we browsed through a few less interesting chains to the chain presented in Figure 1. This chain suggests (semantically), that we might have functionality within our code that notes the last object in a series, or possibly the most recent action in a series. Strings like `_last`, `end`, and `get_last` that we used frequently in more than one class make us suspicious that this functionality might be implemented in a cross-cutting manner.

As we investigate the members of this chain, we find that the members {Last, _last, get_last} are all related to saving state after a certain event has occurred. We discuss the reasons that this concern was implemented in this way and how we could refactor it in the next paragraphs.

Implementation of the *Saving State* concern

In our plugin, we have a `Lexical Chainer` object which uses a `Word Provider` to obtain words as input to its internal, lexical chaining algorithm. The `Lexical Chainer` needs input (a `Word` object) in each cycle of its main loop, which `Word Provider` provides. However, at some points, the `Lexical Chainer` wants to access information about the last `Word` that the `Word Provider` provided and the last iteration through its main loop. This is because the `Lexical Chainer` tries to combine the two next words from the `Word Provider` to form a phrase, like "peanut butter", because the two words together often capture the semantics much better than the two words taken apart. If the `Lexical Chainer` is successful in combining the next two words, then it needs to increment the `Word Provider`, otherwise, it needs to know the part of speech for the original `Word` in order to chain it alone. We call the functionality of saving this state and allowing access to this information the *Saving State* concern.

The following factors contributed to this functionality being implemented across both classes and within numerous methods:

- The `Lexical Chainer` needs several types of state information. This information does not neatly fall in one class, but is spread over the `Lexical Chainer` class and the `WordProvider` hierarchy. For example, the `Lexical Chainer` class defines a `_usedExtraOnLastIteration` instance variable, to record whether the algorithm used two words in its last loop iteration, and the `SimpleName- WordProvider` class defines the `_lastPos` field, to save the last `Word`'s part of speech.
- The developer realized that he needed these pieces of state at varying times throughout development.
- The saving of the state information is an event driven function. When execution reaches certain points in the algorithm, state must be saved. Therefore, state saving code tends to exist in several different methods, as events that cause state saves exist in several places. Examples of such methods are `tryToCombineWords` in class `LexicalChainer` and `getNextWord` in class `SimpleName- WordProvider`.

Refactoring the *Saving State* concern

The *Saving State* concern involves saving the last provided `Word` as well as the last loop iteration's state.To refactor the code that saves the needed state we first considered adding the needed state to the `Word Provider` class. Part of the *Saving State* concern does not clearly fit into `Word Provider`, because it requires that we save state from the last iteration of the lexical chaining algorithm (such as whether we used two words or not). We also do not wish to add the functionality to the `Lexical Chainer` class, because the storing of the last provided `Word` would more naturally fit in the `Word Provider` class.

Without AOP, we are forced to implement this functionality across both classes, with the last `Word`'s state in `Word Provider`, and the last loop's state in `Lexical Chainer`. Using AspectJ, however, we can easily move this state to an aspect. This aspect would define state-related instance variables, such as `_usedTwoWords` and `_lastPos`, and would provide appropriate accessors for those variables. Additionally, it's advice code would consist of state access and update behavior, which would be woven into the `tryToCombineWords` and `getNext-Word` methods in classes `LexicalChainer` and `SimpleName-WordProvider`.

4.2 Exploring PetStore

Running our tool on PetStore generated approximately 700 chains. PetStore provides approximately 57,000 words to chain, and the tool took approximately 7 hours to complete.

One feature of the store is that confirmation emails are sent out to customers that make purchases. Upon running our tool, we found (among other interesting chains) a chain with the members: {SEND_CONFIRMATION_EMAIL, checks, confirmation, insure, check }. By inspecting the source code entities associated with that chain, we discovered that this chain corresponded to the customer-notification feature.

Implementation of the *Customer Notification* concern

Table 2 shows three of the source locations that our tool found. The bolded, italicized words are the actual points in code that our tool found. We present this subset of the original chain, because it neatly corresponds to the core of the customer notification concern.

- In class `ServiceLocator`, we find where the environment property corresponding to this feature is stored. This property determines whether the administrator of the store wants to send out a confirmation email or not.
- In class `PurchaseOrderHelper`, the method `process-Invoice` is in charge of checking whether all of the invoices of a given `PurchaseOrder` are shipped. This method's return value affects whether a confirmation is sent or not.
- Lastly, `MailInvoiceMDB` houses the rest of the core of this feature. It tracks the preference of a given `MailIn- voiceMDB` with its field `sendConfirmationMail`, and it is in charge of actually sending the mail upon completion. It does this through the poorly named method `doTransition`, whose comments ("send a Mail message to mailer service, so customer gets an email") make its purpose more clear.

From this core of a scattered concern, we could use program exploration tools [20, 11] to expand and find the rest of the concern manually. For example, we could search for all methods that use the `sendConfirmationMail` variable of class `MailInvoiceMDB`. In this way, we would find the `ejbCreate` method, which forms part of the concern.

5. LESSONS LEARNED

After constructing the tool and using it to conduct experiments we have learned some valuable lessons, both

In com.sun.j2ee.blueprints.servicelocator.ejb .ServiceLocator
/** * @return the boolean value corresponding * to the env entry such as * SEND_***CONFIRMATION***[1]_MAIL property. */ public boolean getBoolean(String envName) . . .
In com.sun.j2ee.blueprints.purchaseorder.ejb .PurchaseOrderHelper
/**This method processes invoice information received * from supplier by opc. Its job is to update the * LineItem fields for the received invoices. * Additionally it ***checks***[1] if all invoices of the given * PO are shipped, it will return true to indicate all * invoices for a purchase order have been joined * together and the order is completely fulfilled. * @param po * @param lineItemIds * @return true or false to indicate if order is * completely done */ public boolean processInvoice(PurchaseOrderLocal po, Map lineItemIds) { . . .
In com.sun.j2ee.blueprints.opc.customerrelations.ejb .MailInvoiceMDB
private boolean send***Confirmation***[1]Mail = false; ... public void ejbCreate() { . . . if (send***Confirmation***[1]Mail) { String xmlMail = doWork(recdText); doTransition(xmlMail); }

Table 2: Reported Chain Embedded in PetStore Source Code

for using the tool to discover latent concerns and for future research on lexical chaining for concern mining.

The most difficult task when using our tool is identifying interesting lexical chains, especially since a large number of chains can be found. However, we learned that by examining the members of the chains, more interesting chains are easily distinguished from uninteresting ones. More interesting chains often have the following properties:

- *The members are words which do not correspond well to a concern that is already modularized as an object.* For instance, a chain with members {lexical_chainer, chainer, chain} would not be interesting in our system, because we have a `Lexical Chainer` object. A chain with members {Depth, shallowest, _depth} would be interesting in our system, because we have no object that corresponds semantically to this chain.
- *There are more than two members.* Often, if a concern has been implemented in code, the text used to do so will contain numerous forms of the word that will increase the size of the chain. This is because the concern is discussed in comments, as well as used in variable and method names. Comments are especially likely to generate different forms of a similar word for a given concern. A chain with members {changed, change, modified, changer} is more likely to correspond to a significant concern than is the chain with members {change}. Also, a chain with only one member, like {change} is likely to represent a known concern, whereas a chain with different forms of the same word is not.
- *The members are cross-cutting multiple classes.* Since we are particularly looking for cross-cutting concerns, chains whose words correspond to methods defined in several different and unrelated classes are more promising than other chains.

The chains our tool outputs are currently unsorted. However, based on the two last observations, sorting the results seems an obvious extension. Chains with more members should be put in front of chains with less members, for example. Additionally, all chains with an equal amount of members could be sorted according to the "cross-cuttingness" of these members.

We have also learned some valuable lessons for future research using lexical chaining to discover latent concerns in source code.

- *Limit Scope* - Although researchers in NLP have used lexical chaining to chain an entire document, we believe that, when performing lexical chaining on source code we could achieve better results if we performed chaining over single source files, and then compared chains of one source file with other source files. Limiting our scope to a single source file would allow us to explore weaker relationships between words. When the scope is the entire source code base, we cannot afford to consider all but the strongest relationships between words, or the resulting set of chains tends to converge towards a small number of "mega"-chains (chains with a large number of members).
- *Investigate Types of Relationships* - Although, in the current implementation, we consider only the path length of relationships between words, we should investigate the use of the semantics of these relationships and their use in discovering concerns.

6. CONCLUSION

In this paper, we have shown that using lexical chaining to discover cross-cutting concerns in source code is promising, but potentially requires a large overhead. However, because the technique considers source code entities as well as comments, and because it relies on semantic relationships, it finds concerns that other, similar techniques ignore. Additionally, we showed that, in the absence of support for aspects, cross-cutting concerns are indeed implemented using naming conventions and intention-revealing comments.

7. REFERENCES

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

[2] A. Budanitsky. Semantic distance in wordnet: An experimental, application-oriented evaluation of five measures, 2001.

[3] Eclipse Homepage. http://www.eclipse.org. 2005. (Febuary 1, 2005).

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

[5] W. G. Griswold, Y. Kato, and J. J. Yuan. Aspect browser: Tool support for managing dispersed aspects. In *Workshop on MDSOC*, 2000.

[6] M. Halliday and R. Hasan. *Cohesion in English.* Longman, London, 1976.

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

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

[9] J. Morris and G. Hirst. Lexical cohesion computed by thesaural relations as an indicator of the structure of text. *Comput. Linguist.*, 17(1):21–48, 1991.

[10] I. Nancy and V. Jean. Word sense disambiguation: The state of the art, 1998.

[11] M. P. Robillard and G. C. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *Int. Conf. on Soft. Eng.*, 2002.

[12] H. Rubenstein and J. B. Goodenough. Contextual correlates of synonymy. *Commun. ACM*, 8(10):627–633, 1965.

[13] D. Shepherd, J. Palm, L. Pollock, and M. Chu-Carroll. Timna: A framework for the combination of aspect mining analyses. In *Technical Report, University of Delaware*, November 2004.

[14] D. Shepherd and L. Pollock. Aspects, views, and interfaces. In *Wkshp on Linking Aspect Technology and Evolution at AOSD*, March 2005.

[15] I. Sun Microsystems. Pet Store Demo homepage, http://developer.java.sun.com/developer/releases/petstore/. (Febuary 1, 2005).

[16] P. Tarr, H. Ossher, W. Harrison, and J. Stanley M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *ICSE*, 1999.

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

[18] T. Tourwé and K. Mens. Mining Aspectual Views using Formal Concept Analysis. In *Int. Wkshp. on Source Code Analysis and Manipulation*, pages 97 – 106. IEEE Computer Society, 2004.

[19] D. Tufis and O. Mason. Tagging romanian texts: a case study for qtag, a language independent probabilistic tagger.

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