Weekly Meeting

1. PROBLEM STATEMENT

Software maintainers spend a lot of time trying to understand the existing software before performing the actual modification. For modification tasks such as debugging, finding and understanding program elements relevant to a bug fix tends to take more time than actually fixing the bug. Finding related code elements is used in a variety of software development and maintenance activities. For instance, defect prediction tools identify fault-prone elements via dependency graphs [11, 13, 26], impact analysis tools estimate potentially impacted entities by analyzing related elements of a proposed change [10, 14], API search tools identify related elements in the call graph based on user query [5, 12], and feature localization tools find related elements which represent a specific concern [1, 20, 24].

Numerous measures have been proposed to identify related program elements by leveraging execution trace [3], historical changes [23, 25], and lexical information in identifiers and documents [8, 16], yet these data may not always be available and reliable compared to source code. A study on program investigation [18] shows that effective developers tend to find related elements by following structural dependencies. However, in any non-trivial software system, the number of structural dependencies to follow is much too large to be completely covered by a developer. As a result, developers must rely on their intuition to determine where to look. Based on the hypothesis that *it is possible to find related elements following program's structural dependency,* a variety of approaches have been reported to suggest elements of potential interest via structural metrics [4], program slicing [21], and topology-based analysis [17, 26]. We observe several limitations for existing approaches: First, we find that most techniques only support a single queried elements and work at a fixed granularity in a limited scope, e.g., direct callers/callees or sibling elements that share callers/callees, but they do not have a good representation for relations in different level, such as separating overriding relations across different classes and calling relations within a single method. Second, existing tools are sensitive to specific callers/callees but do not consider how queried elements are related with them in a common manner, i.e., existing tools do not consider which relations are more common and should be more interesting to users. Third, existing tools assume that it takes equal effort to explore each relation, yet recent study shows that elements with many nearby dependencies require more effort for investigation [9] and hierarchical relations are easier to follow and may have a higher potential to find interesting elements than calling relations [15].

To overcome these limitations, we propose to merge and cluster similar couplings in a layered-order given a set of program elements queried by users, and return a set of grouped elements that are relevant to the given set. We hypothesize that when users query one or more elements, they are interested in not only which elements are directly related to these elements, but also how these elements are coupled with other related elements in a common way, and these commonly related elements should be prioritized.

For example, user selects two bolded methods in class `JsonWriter`: `visit(JsonNumber)` and `visit(JsonBoolean)`. The underline elements represent methods that are invoked in queried ones: `print()` method, `JsonNumber.value`, and `JsonBoolean.value`. Suade [17], the only approach that supports multiple elements of interest, reports these three invoked methods, as well as other 13 callers of these two methods based on class hierarchy analysis to include all methods potentially called. These results can be useful, but a) Suade fails to detect any direct relations between two queried methods and investigates direct relevant elements for them separately, and report invoked methods (`JsonNumber.value()` and `JsonBoolean.value()`) and callers (`JsonNumber.visit()` and `JsonBoolean.visit()`) separately; b) the results are sensitive to the callers of queried elements (`JsonNumberOptional.recordOptional()`, `JsonFormatNumberHandler.printNumber()`, `JSON.write()`, and the rest 10 callers). I hypothesize that relevant elements that users are interested in should be `<JsonValue>.visit()` and `JSON.write()` as callers, and `<JsonValue>.value()` and `print()` as methods invoked by queried elements.

2. RELATED WORK

Most measures for related element identification are *structural* by following method call chains, control flow, and variable def-use. These works leverage coupling metrics [4], program slicing [21], and topology-based search [17, 26].

Static Metrics. Briand et al. propose a set of structural coupling metrics to measure coupling between two classes. such as Coupling Between Objects (CBO) and Response for

Table 1: Table of different structural measure to identify related elements

Type	Citation	Level	Represent	Input	Application	Limitation
Metrics	coupling [4], fault prediction [7, 13]	class	metrics	a single class	impact analysis	coarse-grained
Slicing	Weiser [22], Thin slice [21]	method	graph	a single element	navigation, code search	expensive, large slice
Topology -based	Suade [17, 19], network [26], dependency graph [11, 14, 15]	method	fuzzy set, network	1 or more elements	navigation, impact analysis	limited scope

(A) Methods queried by user

```
/* org.apache.jena.json.JsonWriter */
1. public class JsonWriter implements JsonVisitor {
2.  OutputStreamWriter writer;
3.  public void visit(JsonNumber jsonNumber)  {
4.   print(jsonNumber.value()) ;
5.  }
6.  public void visit(JsonBoolean jsonBoolean) {
7.   String x = Boolean.valueOf(jsonBoolean.value())?"true":"false";
8.   print(x);
9.  }
10. private void print(Object s) {...
11.  writer.println(s);
12. } ...}
```

(B) Callers of these two methods

```
 /* org.apache.accumulo.core.sync.SortedMapIterator */
1. public class JsonNumber extends JsonValue {
2.  Format format;
3.  @Override
4.  public Object value()
5.   { return format.getValue(value); }
6.  @Override
7.  public void visit(JsonVisitor visitor)
8.   { visitor.visit(this) ; }
9. }
10. public class JsonBoolean extends JsonValue {
11.  @Override
12.  public Object value()
13.   { return value; }
14.  @Override
15.  public void visit(JsonVisitor visitor)
16.   { visitor.visit(this) ; }
17. }
 /* Three of other 13 callers */
18. public class JsonFormatNumberHandler extends JsonHandler {
19.  public static void printNumber(JsonNumber jNum) {
20.   JsonWriter w = new JsonWriter(output);
21.   jNum.setFormat(format);
22.   w.visit(jNum);
23. } ...}
24. public class JSONNumberOptional {
25.  JsonBoolean jBool;
26.  JsonNumber left, right;
27.  public static void recordOptional() {
28.   JsonWriter w = new JsonWriter(output);
29.   w.visit(left);
30.   w.visit(jBool);
31.   w.visit(right);
32. } ...}
33. public class JSON {
34.  public static void write(JsonValue jValue) {
35.   JsonWriter w = new JsonWriter(output);
36.   w.visit(jValue);
37. } ...}
```

Figure 1: Example of suggested elements given a set of interest

a Class (RFC). A set of metrics are proposed to predict defects based on the complexity or defect history [7,13]. Yet static metrics are usually too coarse-grained that limits their usefulness to recommend code of immediate interest.

Program slicing. Weiser [22] proposes to identify a part of program that may affect the values computed at some point of interest. Thin slicing [21] only consists of producer statements for the seed and hierarchically expanded to include statements explaining how producers affect the seed. Yet computing slicing can be expensive and slices are often too large to be useful for users.

Topology-based analysis. Suade algorithm [17] ranks elements based on the closeness of their structural association with program elements in a set of interest. It only returns elements that are directly related to the set of interest. FRAN [19] extends Suade by considering the sibling elements that share common caller or callee with queried elements. FRAN focuses on calling relation in C given a single queried function. Zimmermann and Nagappan [26] use network analysis to identify *central binaries* based on dependency graph, aiming to predict fault-prone modules based on defect history. Other usage of identifying related elements rely on calling graphs with class hierarchy analysis for dynamic binding [11, 14, 15]. However, none of them groups relations in different granularities and identifies commonly used relations in layered-order.

Considering that structural coupling often results in a large number of relationships within a limited scope, researchers have proposed other alternative measures to identify related elements.

Briand et al. [3] handle dynamic binding in object-oriented programs by analyzing of the execution of a program. Dynamic slicing is a variant of slicing that select related code piece considering program execution trace [2]. Specifically, dynamic slicing only considers program dependencies that occur in a specific execution of the program. In contrast to static approaches, dynamic measures relies on the availability and quality of test cases for an executable program, thus they cannot be applied to incomplete code or code that cannot be executed.

Ying et al. [23] report elements that are often changed together during program evolution tasks at the file level, Fluri et al. identify frequently co-changed files and filter co-changed methods that result from structural changes [6]. However, reliance on change history implies that the approach cannot be used when queried elements are never changed before.

Revelle et al. [16] exploit relations captured from the source code lexicon using Information Retrieval techniques and propose a set of semantic metrics to define a coupling between classes based on textural information from code and comments. Hill et al. [8] use lexical information to identify related methods corresponding to queried ones and expand call chains to find relevant elements missed by lexical search. The main tradeoff for these measures is that they assume that similar terms always indicate related functionality.

In summary, these alternative measures aim to capture relations that are missed by existing structural coupling by leveraging revision history, lexical information, and execution traces. Our approach aims to extend the structural

measures to group relations in different granularities and identify commonly used relations based on available and reliable structural dependency.

3. REFERENCES

[1] B. Adams, Z. M. Jiang, and A. E. Hassan. Identifying crosscutting concerns using historical code changes. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 305–314, 2010.

[2] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI), White Plains, New York, USA, June 20-22, 1990*, pages 246–256, 1990.

[3] E. Arisholm, L. C. Briand, and A. Føyen. Dynamic coupling measurement for object-oriented software. *IEEE Trans. Software Eng.*, 30(8):491–506, 2004.

[4] L. C. Briand, J. W. Daly, and J. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Trans. Software Eng.*, 25(1):91–121, 1999.

[5] S. Chatterjee, S. Juvekar, and K. Sen. SNIFF: A search engine for java using free-form queries. In *Fundamental Approaches to Software Engineering, 12th International Conference, FASE 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, pages 385–400, 2009.

[6] B. Fluri, H. C. Gall, and M. Pinzger. Fine-grained analysis of change couplings. In *5th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2005), 30 September - 1 October 2005, Budapest, Hungary*, pages 66–74, 2005.

[7] T. Gyimóthy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Trans. Software Eng.*, 31(10):897–910, 2005.

[8] E. Hill, L. L. Pollock, and K. Vijay-Shanker. Exploring the neighborhood with dora to expedite software maintenance. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, pages 14–23, 2007.

[9] R. Holmes, T. Ratchford, M. P. Robillard, and R. J. Walker. Automatically recommending triage decisions for pragmatic reuse tasks. In *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*, pages 397–408, 2009.

[10] H. H. Kagdi, M. Gethers, and D. Poshyvanyk. Integrating conceptual and logical couplings for change impact analysis in software. *Empirical Software Engineering*, 18(5):933–969, 2013.

[11] Z. Li and Y. Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 306–315, 2005.

[12] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 111–120, 2011.

[13] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, pages 452–461, 2006.

[14] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference, ESEC/FSE 2003, Helsinki, Finland, September 1-5, 2003*, pages 128–137, 2003.

[15] S. Rastkar, G. C. Murphy, and A. W. J. Bradley. Generating natural language summaries for crosscutting source code concerns. In *IEEE 27th International Conference on Software Maintenance, ICSM 2011, Williamsburg, VA, USA, September 25-30, 2011*, pages 103–112, 2011.

[16] M. Revelle, M. Gethers, and D. Poshyvanyk. Using structural and textual information to capture feature coupling in object-oriented software. *Empirical Software Engineering*, 16(6):773–811, 2011.

[17] M. P. Robillard. Automatic generation of suggestions for program investigation. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 11–20, 2005.

[18] M. P. Robillard, W. Coelho, and G. C. Murphy. How effective developers investigate source code: An exploratory study. *IEEE Trans. Software Eng.*, 30(12):889–903, 2004.

[19] Z. M. Saul, V. Filkov, P. T. Devanbu, and C. Bird. Recommending random walks. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, pages 15–24, 2007.

[20] D. C. Shepherd, Z. P. Fry, E. Hill, L. L. Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *Proceedings of the 6th International Conference on Aspect-Oriented Software Development, AOSD 2007, Vancouver, British Columbia, Canada, March 12-16, 2007*, pages 212–224, 2007.

[21] M. Sridharan, S. J. Fink, and R. Bodík. Thin slicing. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 112–122, 2007.

[22] M. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.

[23] A. T. T. Ying, G. C. Murphy, R. T. Ng, and M. Chu-Carroll. Predicting source code changes by

mining change history. *IEEE Trans. Software Eng.*, 30(9):574–586, 2004.

[24] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang. SNIAFL: towards a static noninteractive approach to feature location. *ACM Trans. Softw. Eng. Methodol.*, 15(2):195–226, 2006.

[25] Y. Zhou, M. Würsch, E. Giger, H. C. Gall, and J. Lü. A bayesian network based approach for change coupling prediction. In *WCRE 2008, Proceedings of the 15th Working Conference on Reverse Engineering, Antwerp, Belgium, October 15-18, 2008*, pages 27–36, 2008.

[26] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 531–540, 2008.