Assessing Modularity via Usage Changes

Yana Momchilova Mileva
Saarland University
Saarbrücken, Germany
mileva@cs.uni-saarland.de

Andreas Zeller
Saarland University
Saarbrücken, Germany
zeller@cs.uni-saarland.de

ABSTRACT

Good program design strives towards modularity, that is, limiting the effects of changes to the code. We assess the modularity of software modules by mining change histories: The more a change to a module implementation changes its usage in client code, the lower its modularity. In an early analysis of four different releases of open-source projects, we found that changes can differ greatly in their impact on client code, and that such impact helps in assessing modularity.

Categories and Subject Descriptors: D.2.2 [Software Engineering]: Modules and interfaces

General Terms: Measurement, Reliability.

Keywords: Program analysis, Maintenance, Consistent updates.

1. INTRODUCTION

In software development, change is the only constant. New features are added, defects are fixed, or code is being refactored. Modular software design attempts to limit the effect of these changes, in particular by hiding implementation details behind interfaces, such that implementation changes would not induce changes in client code. During maintenance of a system, one needs to understand which modules suffer from low modularity. This is important for assessing the potential (non-local) impact of changes. It is important because such modules may be candidates for refactoring.

In this study, we investigate the extent to which a change to a module implementation requires changing client code—that is, code that is using elements of the module. Our early results show that the impact of changes varies greatly from module to module. This facilitates refactoring tasks, as discussed above; most important, though, is that change history is now frequently used for predicting defects [12, 7, 2, 4]. Knowing about the impact of potential changes and knowing about modularity of individual components could make such predictions much more precise.

PASTE'11, September 5, 2011, Szeged, Hungary.


```
void findJavadocInlineTags(...) {
 if (!this.requestor.isIgnored
   (CompletionProposal.JAVADOC_INLINE_TAG)) {
  ...
- CompletionProposal prop = this.createProposal
    (CompletionProposal.JAVADOC_INLINE_TAG, ...);
+ InternalCompletionProposal prop = createProposal
     (CompletionProposal.JAVADOC_INLINE_TAG, ...);
  prop.setCompletion(...);
  prop.setRelevance(...);
 }
}
```

Figure 1: Change in the *usage* of the `CompletionProposal` class that occurred in the Eclipse code between versions 3.4.2 and 3.5.2.

As an example, consider Figure 1, showing client usage of the `CompletionProposal` class in the Eclipse project. The figure shows how the usage of this class has changed between two Eclipse versions. This class went through some major implementation changes. If the client code had remained unchanged it would not have issued a compilation error, but would have resulted in unwanted behavior on the client side. A change in the implementation of `CompletionProposal` resulted in the need to modify its client code. The more such changes happen to this class, the lower its modularity—and the higher the likelihood of future changes to induce unwanted effects.

The approach we present in this paper is concerned with the influence of change in a module's implementation on the module's client code. As *module's implementation change* we consider any kind of change that was performed on the code of the module—anything from altering a method's signature to adding a new method call within a method; we track all changes. A *module's usage change* is any change in the usage of the module's elements (e.g. module's methods or variables) in the code of another software component. The specific challenge is to assess usage changes (and thus modularity) in a way that is *independent of the number of clients*, since modularity should be assessable even without a specific context. (Otherwise, a module with just one client would be far more "modular" than any module with hundreds of clients.) This requires us to *abstract* usage changes into common *patterns*—the more such unique patterns, the greater the change in usage.

Table 1: Evaluation Subjects.

Case study	# Modules all	# Modules changed	median loc
Eclipse 3.4.2	17,350	59	30
Eclipse 3.5.2	18,531	-	-
AspectJ 1.6.2	1940	17	35
AspectJ 1.6.3	1952	-	-

The present paper thus makes two contributions: 1) Leveraging patterns of usage change for the analysis of version histories; 2) Investigating the relationship between implementation and usage changes to assess modularity.

2. MINING MODULARITY

In this work we define *modularity* as *changes to a module's implementation that do not lead to changes in its usage.*

To assess modularity we analyze two variables—the number of module's implementation changes performed and the number of module's usage changes that followed. We perform our analysis by comparing an older and a newer version of the module's code and an older and a newer version of the analyzed project's code. By comparing the module's versions we learn how the module's code changed and from the comparison of the project's versions we learn how the usage of the module changed.

2.1 Collecting Implementation Changes

In order to see how the code of a module changed we track the number of changed lines of code between the two versions of the module. To minimize the noise we ignore empty lines as well as comment lines. Any other line that was changed we count as changed.[1]

Table 1 lists our test subjects, together with the total number of modules present in each one of them; the number of the modules that we were able to detect as changed and the median of the changed code lines per module. The small number of changed modules comes from the fact that the analyzed projects are in a relatively stable state in their project evolution. As we used an exact matching of the fully qualified name we might have also omitted the change in some modules, due to a change in the name of the module. The reported number of changed code lines comes from comparing the two versions of a project's module—which is why we report only one median number per project.

2.2 Collecting Usage Changes

In order for us to detect how a specific module and its elements are being used, we use our tool LAMARCK [6] to collect and extract the module's usage information.

LAMARCK receives as an input the binary code of two versions of a project and looks for the evolution patterns that occurred between those two versions. An *evolution pattern* is a code change pattern that occurred as a result of the project's evolution from the first version to the second.

[1]We acknowledge that more sophisticated approaches exist that track the evolution of a piece of code, however we believe that the approach we chose is sufficient to give us a good estimate of the changes in a module.

C:`CompletionProposal.setCompletion()` $\prec$ `CompletionProposal.setRelevance()`
D:`CompletionProposal.setCompletion()` $\prec$ `CompletionProposal.setRelevance()`
A:`InternalCompletionProposal.setCompletion()` $\prec$ `InternalCompletionProposal.setRelevance()`

Figure 2: The evolution pattern corresponding to the code change indicated on Figure 1. The pattern shows what the *context* of the change was (indicated by the "C" properties) and what was deleted and added (indicated by the "D" and "A" properties).

LAMARCK goes through four basic stages in order to produce those evolution patterns.

Mining object usage models. For each statically identifiable object used in each of the project's methods, an object usage model (OUM) is created. An *object usage model* is a finite state automata that shows how objects "flow" through various events in a method (e.g. returning a value). Previous research has demonstrated the potency of OUMs in expressing the behavior of an object [10, 6].

Extracting temporal properties. A temporal property is an ordered pair of events a and b associated with the same object. We use the expression $a \prec b$ to represent an ordering where event a may happen before event b. A few examples of such events are *calling a method on an object*, *type casting* or *returning a value* [6]. In this stage, LAMARCK statically extracts the temporal properties for each method of the two analyzed versions.

Extracting change properties. After extracting all the temporal properties per method for each of the two versions, LAMARCK compares the sets of temporal properties per method to come up with one set of evolution temporal properties for each method. These combined sets of properties consist of annotated temporal properties, which allow us to track how the code evolved. For example, "D: $a \prec b$" denotes that the temporal property of event a happening before event b has been deleted when the project evolved (see Fig. 2).

Mining evolution patterns. LAMARCK takes all the sets of annotated temporal properties and, with the help of concept analysis [3], detects the frequently occurring sets. We call those frequently occurring sets evolution patterns, as they are the code change patterns that resulted from the evolution of the project. Figure 2 shows the evolution pattern[2] that corresponds to the code change from Figure 1.

Our objective here is to see how the usage of a module has changed. A change in the usage of a module will be detected by the presence of an evolution pattern that contains elements of this module (e.g. calls to the module's methods). We represent the module's usage changes by the number of evolution patters, of which the module is part of. One

[2]The evolution pattern has been slightly modified to fit the format of the paper. LAMARCK works with the full signatures of the methods.

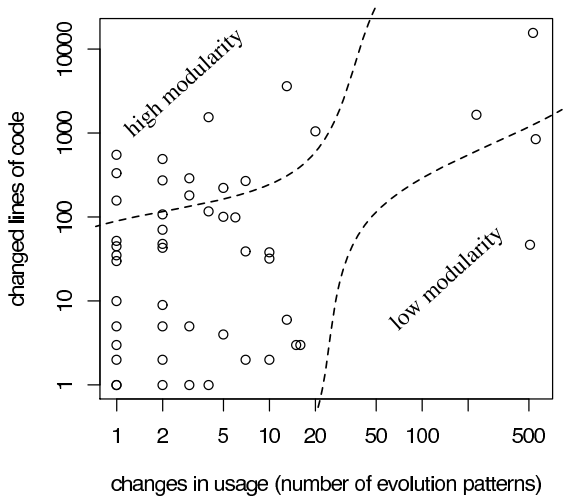

Figure 3: Correlation between implementation changes and usage changes (Eclipse 3.4.2 to 3.5.2). Each point represents a module.

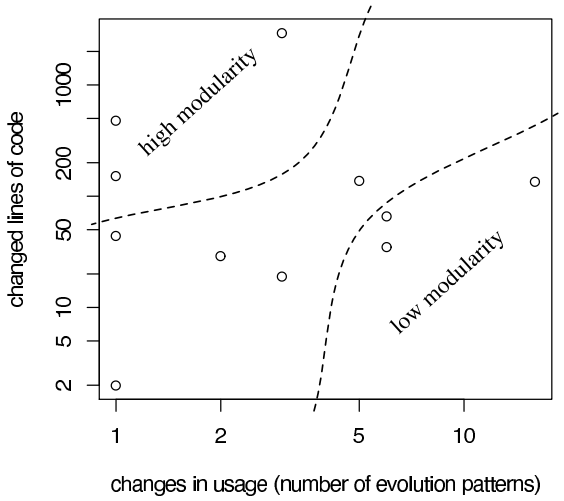

Figure 4: Correlation between implementation changes and usage changes (AspectJ 1.6.2 to 1.6.3). Each point represents a module.

would notice that if a module is present in a lot of evolution patterns this would mean that its usage has changed significantly. In the world of module's clients, module's code changes that lead to changes in the usage of this module are not welcomed, as any such change might lead to potential defects in the client's code.

3. EVALUATION

We examined the test subjects presented in Table 1 to assess the modularity of their modules based on the existence of implementation and usage changes. In our experiments we consider as modules Java classes.

3.1 Quantitative Evaluation

In order to see how much the code of a module changed between two versions we collected the number of changed lines of code between those two module versions. To see how the usage of the module changed, we collected the number of evolution patterns it was part of.

Figure 3 presents the correlation between the Eclipse's modules' implementation and usage changes for the transition from version 3.4.2 to version 3.5.2. Figure 4 shows the correlation data for the AspectJ project for the transition from version 1.6.2 to 1.6.3. As one can see from these figures, the modules that have a large number of usage changes are easy to spot. Those modules would be the ones which we would point out as modules with **low modularity**. Modules with low number of usage changes we classify as modules with **high modularity**.

3.2 Qualitative Evaluation

Let us now take a look at a few examples of modules and discuss their modularity.

A module with low modularity would be a module whose implementation changes lead to a lot of usage changes. For the Eclipse project an example of such a module is the `CompletionProposal` class. This class had more than 840 changed lines of code between the two revisions and those changes affected more than 550 evolution patterns. We found bug reports related to this class reported for Eclipse 3.4.2 and targeted for 3.5.2. In the bug comments of one of them was explicitly stated that the clients need to "apply a patch" to fix the problem (see Eclipse Bug #281575).

Another example of a module with low modularity would be the `AsmManager` class from the AspectJ project (see Figure 1). The implementation changes for this class amount to 135 changed lines of code and these changes affected 16 evolution patterns.

As an example of a module with high modularity we would point out the `StyledText` Eclipse class. This class was also a subject to a lot of implementation changes, amounting to 1547 changed lines of code. These changes however influenced only four of our evolution patterns, resulting in a very low impact of the applied implementation changes.

3.3 Threats to Validity

As any other empirical study, our study is prone to threats to its validity. Our early results are unlikely to generalize to arbitrary projects. The fact that Java makes it hard to distinguish implementation and interface calls for better analysis methods. Despite our systematic checks of data and results, our implementation could contain errors that affect the outcome.

4. RELATED WORK

Martin Robillard [9] explores the question of API usability, by making a user study on what makes APIs hard to learn. He showed that one of the ways developers learn about the API design is through examples (i.e. evolution patterns). This supports our hypothesis that usage and implementation are tightly linked.

Dagenais and Robillard [1] also recognized that a change in the implementation of an API might lead to problems in the client code. Their SemDiff tool recommends replace-

ments for API methods that were deleted during the evolution of an API.

The approach developed by Hovemeyer and Pugh [5] detects bug patterns in the usage of an API. Our approach could also be used for detecting code defects, as we look at modules with low modularity, i.e. any code that is using such a module would be a subject to potential code defects.

Change impact analysis [8] would allow us to get an account of modules impacted by a change. However, the modularity of a module should not depend on the number of clients. By focusing on usage changes and abstracting those changes into evolution patterns, we can easily summarize common usage changes even across a wide range of modules.

Co-changes in version histories [13] are components frequently change together; these can also be used to assess and predict the impact of changes. Again, we need a common abstraction method to become independent of the number of modules.

Finally, *centrality measures* [11] relate the likelihood of a component failure to the number of dependent clients. Again, we want to our modularity measure to be independent of a particular context.

5. CONCLUSION AND CONSEQUENCES

Directing one's resources toward the weakest spots in a project has always been of vital importance in the software development process. We presented an approach that assesses the modularity of software modules and can thus assist in directing one's attention toward modules with low modularity.

Our future work on improving the presented approach will concentrate on the following topics:

Implementation and usage trends. We are planning on investigating how evolution patterns change over time in correlation with the implementation changes. If we notice a continuous correlation between implementation changes and a high number of usage changes, this would further strengthen our observations regarding which modules are of low modularity.

Classification of implementation changes. In the presented approach we are using a simple method for estimating the number of changes in module's implementation. Our observations will be more precise if we are able to detect if a change is in effect introducing a new feature, as this might lead to usage changes, but would not indicate a low modularity of the module.

User studies. We are highly interested in the ways a software developer or manager will approach this modularity information. In that regard, we are planning on performing user studies to further validate the usefulness of our approach.

Acknowledgments: Yana Mileva is funded by Microsoft Research Cambridge Lab and Max Planck Institut für Informatik. The authors thank Kim Herzig, David Schuler and Jeremias Rößler for their comments on earlier version of the papers. Special thanks go to Valentin Dallmeier and the anonymous reviewers for their suggestions for improvement.

6. REFERENCES

[1] B. Dagenais and M. P. Robillard. Semdiff: Analysis and recommendation support for api evolution. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 599–602, Washington, DC, USA, 2009. IEEE Computer Society.

[2] M. Gegick, P. Rotella, and L. Williams. Predicting attack-prone components. In *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, pages 181–190, Washington, DC, USA, 2009. IEEE Computer Society.

[3] D. N. Götzmann. Formale Begriffsanalyse in Java: Entwurf und Implementierung effizienter Algorithmen. Bachelor thesis, Saarland University, 2007. Publication and software available from (accessed 6 April 2010) http://code.google.com/p/colibri-java/.

[4] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 78–88, Washington, DC, USA, 2009. IEEE Computer Society.

[5] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39:92–106, December 2004.

[6] Y. Mileva, A. Wasylkowski, and A. Zeller. Mining evolution of object usage. In *Proceedings of the 25th European Conference on Object-Oriented Programming*, ECOOP '11, July 2011.

[7] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy. Change bursts as defect predictors. In *Proceedings of the 21st IEEE International Symposium on Software Reliability Engineering*, November 2010.

[8] X. Ren, B. G. Ryder, M. Stoerzer, and F. Tip. Chianti: a change impact analysis tool for java programs. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 664–665, New York, NY, USA, 2005. ACM.

[9] M. P. Robillard. What makes apis hard to learn? answers from developers. *IEEE Softw.*, 26:27–34, November 2009.

[10] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 35–44, New York, NY, USA, 2007. ACM.

[11] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 531–540, New York, NY, USA, 2008. ACM.

[12] T. Zimmermann, N. Nagappan, and L. Williams. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 421–428, Washington, DC, USA, 2010. IEEE Computer Society.

[13] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, pages 563–572. IEEE Computer Society, May 2004.