



# Mining the Evolution of Software Component Usage

by

Yana Momchilova Mileva

Dissertation zur Erlangung des Grades des Doktors der Ingenieurwissenschaften der  
Naturwissenschaftlich-Technischen Fakultäten der Universität des Saarlandes  
Saarbrücken, März 2012

### Statutory declaration

I hereby swear in lieu of an oath that I have independently prepared this thesis and without using other aids than those stated. The data and concepts taken over from other sources or taken over indirectly are indicated citing the source. The thesis was not submitted so far either in Germany or in another country in the same or a similar form in a procedure for obtaining an academic title.

Place, date

(signature)

©2012 by Yana Mileva  
All rights reserved. Published 2012.

*Day of Defense*

*Dean*

*Head of the Examination Board*

*Members of the Examination Board*

July 9th, 2012

Prof. Dr. Mark Groves

Prof. Dr. Dr. h.c. Reinhard Wilhelm

Prof. Dr. Andreas Zeller,

Prof. Dr. Gerhard Weikum,

Dr. Alessandra Gorla





# Abstract

The topic of this thesis is the analysis of the evolution of software components. In order to track the evolution of software components, one needs to collect the evolution information of each component. This information is stored in the version control system (VCS) of the project—the repository of the history of events happening throughout the project’s lifetime. By using software archive mining techniques one can extract and leverage this information.

The main contribution of this thesis is the introduction of *evolution usage trends* and *evolution change patterns*. The raw information about the occurrences of each component is stored in the VCS of the project. By organizing it in evolution trends and patterns, we are able to draw conclusions and issue recommendations concerning each individual component and the project as a whole.

**Evolution Trends** An evolution trend is a way to track the evolution of a software component throughout the span of the project. The trend shows the increases and decreases in the usage of a specific component, which can be indicative of the quality of this component. AKTARI is a tool, presented in this thesis, that is based on such evolution trends and can be used by the software developers to observe and draw conclusions about the behavior of their project.

**Evolution Patterns** An evolution pattern is a pattern of a frequently occurring code change throughout the span of the project. Those frequently occurring changes are project-specific and are explanatory of the way the project evolves. Each such evolution pattern contains in itself the specific way “things are done” in the project and as such can serve for defect detection and defect prevention. The technique of mining evolution patterns is implemented as a basis for the LAMARCK tool, presented in this thesis.



# Zusammenfassung

Der Mittelpunkt dieser Arbeit ist die Analyse der Evolution von Software Komponenten. Um die Evolution von Software Komponenten verfolgen zu können, benötigt man Informationen über die Evolution jeder einzelnen Komponente. Diese Informationen sind gespeichert in Versionskontrollsystemen - den Speichern der kompletten Geschichte der Ereignisse, die sich in der Laufzeit eines Projektes zutragen.

Der Hauptbeitrag dieser Arbeit ist die Einführung von evolutionären Nutzertrends und evolutionären Änderungsmustern. Die unverarbeiteten Informationen über die Verwendung jeder einzelnen Komponente ist in dem Versionskontrollsystem eines Projektes gespeichert, und durch die Organisierung in evolutionären Änderungsmustern und Trends können wir Schlüsse daraus ziehen und Empfehlungen aussprechen für jede einzelne Komponente und das Projekt als Ganzes.

**Evolutionäre Nutzertrends** Ein evolutionärer Nutzertrend ist eine Möglichkeit die Evolution einer Software Komponente durch das komplette Projekt hindurch zu verfolgen. Der Trend zeigt Anstieg und Abnahme der Nutzung einer einzelnen Komponente, was Aussagen über die Qualität dieser Komponente zulässt. Das Werkzeug AKTARI, das in dieser Thesis vorgestellt wird, basiert auf solchen evolutionären Nutzertrends, und kann von Software-Entwicklern dazu genutzt werden diese Trends zu erkennen und Schlüsse über das Verhalten Ihres Projektes aus ihnen zu ziehen.

**Evolutionäre Änderungsmuster** Wir nennen ein Muster von Codeänderungen, das durch die komplette Projektgeschichte hindurch wiederholt wiederkehrt, ein evolutionäres Änderungsmuster. Diese wiederholt wiederkehrenden Änderungen sind projektspezifisch und erklären die Art wie sich das Projekt entwickelt. Jedes einzelne evolutionäre Änderungsmuster enthält die spezifische Art, wie im Projekt "Dinge getan werden" und kann somit zur Fehlererkennung und -vermeidung dienen. Diese Technik ist implementiert als Basis des Werkzeuges LAMARCK, welches in dieser Thesis vorgestellt wird.





# Acknowledgments

First and foremost, I thank my adviser Andreas Zeller for his support over the years. I thank both Max-Planck-Institut Informatik and Microsoft Research Cambridge Lab for supporting and sponsoring my PhD research. A special thank you goes to the reviewers of this thesis, Andreas Zeller and Gerhard Weikum, as well as to the head of the examination board Reinhard Wilhelm.

I would like to thank all researchers I was in contact with during the years of my PhD, for the fruitful ideas and lively discussions we shared. A big thank you goes especially to Valentin Dallmeier and Andrzej Wasylkowski for our research collaborations, as well as to Thomas Zimmermann, who introduced me to the enchanting field of mining software archives.

A thank you goes to all of my colleagues at the chair for Software Engineering at Saarland University for the time we shared together. I would like to express my special gratitude to my colleagues Kim Herzig, Gordon Fraser, Kevin Streit and Valentin Dallmeier for their support and feedback on earlier versions of this thesis. A very special thanks goes to Jeremias Rößler, who helped me proofread the thesis and who translated the abstract to German. I would also like to express my special gratitude to my office mates, Andrzej Wasylkowski and Jeremias Rößler, who made my days at the office full of joy and fascinating discussions.

This thesis would not have been possible without the support of all my friends and family and I would like to thank all the people who have been next to me during this time.



# Contents

<b>1</b>	<b>Introduction and Basic Concepts</b>	<b>1</b>
1.1	Contributions . . . . .	2
1.2	Example . . . . .	3
1.2.1	Trend Types . . . . .	5
1.2.2	Applications . . . . .	6
1.3	Publications . . . . .	8
1.4	Summary . . . . .	9
<b>2</b>	<b>Evolution of API Components</b>	<b>11</b>
2.1	Introduction . . . . .	12
2.2	Early Adopters vs. Late Followers . . . . .	12
2.3	Evolution of APIs . . . . .	14
2.3.1	API Users and Producers . . . . .	15
2.3.2	Collecting and Presenting Usage Data . . . . .	15
2.3.3	Predicting Usage Trends . . . . .	16
2.3.4	Evaluation . . . . .	18
2.3.5	Threats to Validity . . . . .	21
2.4	Evolution of API Versions . . . . .	22
2.4.1	API Users and Producers . . . . .	22
2.4.2	Extracting Version Dependencies . . . . .	23
2.4.3	Usage trends . . . . .	24
2.4.4	Threats to Validity . . . . .	28
2.5	The Aktari Tool . . . . .	29
2.6	Related Work . . . . .	31
2.7	Summary . . . . .	32

## CONTENTS

<b>3</b>	<b>Evolution of Object Usage</b>	<b>35</b>
3.1	The Problem . . . . .	35
3.2	Object Usage Evolution . . . . .	37
3.2.1	Temporal Properties . . . . .	39
3.2.2	Change Properties . . . . .	41
3.3	Mining Patterns . . . . .	44
3.3.1	Detecting Evolution Patterns . . . . .	44
3.3.2	Finding Missing Changes . . . . .	46
3.4	Evaluation . . . . .	47
3.4.1	Detecting Errors . . . . .	48
3.4.2	Preventing Errors . . . . .	55
3.5	Applications . . . . .	59
3.5.1	Detecting Errors . . . . .	59
3.5.2	Preventing Errors . . . . .	60
3.5.3	Threats to Validity . . . . .	60
3.6	Related Work . . . . .	61
3.6.1	Learning Evolution Rules . . . . .	61
3.6.2	Learning from Project History . . . . .	62
3.7	Summary . . . . .	63
<b>4</b>	<b>Assessing Modularity</b>	<b>65</b>
4.1	The Problem . . . . .	65
4.2	Collecting Usage and Implementation Changes . . . . .	68
4.2.1	Collecting Implementation Changes . . . . .	68
4.2.2	Collecting Usage Changes . . . . .	68
4.3	Evaluation . . . . .	70
4.3.1	Quantitative Evaluation . . . . .	70
4.3.2	Qualitative Evaluation . . . . .	71
4.4	Applications . . . . .	73
4.4.1	API producers . . . . .	73
4.4.2	API users . . . . .	74
4.4.3	Defect prediction techniques . . . . .	75
4.5	Threats to Validity . . . . .	75
4.6	Related Work . . . . .	75
4.7	Summary . . . . .	76
<b>5</b>	<b>Conclusions and Future Work</b>	<b>77</b>
5.1	Future Work . . . . .	79

*CONTENTS*

**Bibliography**

**81**



# List of Figures

1.1	The process of extraction and usage of trends of software components.	4
2.1	The diffusion of innovations according to Rogers.	13
2.2	Examples of the four usage trend types.	16
2.3	Declaring a dependency to the <i>servlet-api</i> library in a MAVEN Project Object Model	24
2.4	Usage trends of the <i>junit</i> library	24
2.5	Usage trends of the <i>log4j</i> library	25
2.6	AKTARI—Eclipse plug-in design	30
2.7	AKTARI—web-tool	30
2.8	AKTARI web-tool.	31
3.1	Method change from Eclipse 1.0 to 2.0. The old check has been replaced by a custom method call. Has this change been propagated consistently across Eclipse?	36
3.2	How LAMARCK works. Given two versions (a), LAMARCK extracts temporal properties (b) that characterize object usage in each version. The differences (c) are then mined for recurrent change patterns (d). These patterns can then be used to search for missing changes (e) in new code (f).	37
3.3	Object usage model for the target object (“this”) accessed by the methods shown in Figure 3.1 (Eclipse 1.0 version). Dashed lines denote empty transitions (without calls).	40
3.4	Concept Analysis Matrix	45
3.5	Method change from Eclipse 2.0 to 2.1. The added methods after the change address a font inconsistency.	52

# LIST OF FIGURES

3.6	Method change in AspectJ. After the change <code>buildView()</code> now takes an object derived from the <code>Ajde</code> singleton. . . . .	53
3.7	Method change from Eclipse 1.0 to 2.0. Before the change there was a call to a deprecated method. . . . .	54
3.8	Evaluation Setting. In 1/10 of the code, we artificially revert changes and check whether LAMARCK is able to predict them after learning from the changes in the remaining 9/10. . . . .	55
3.9	Influence of minimum support on LAMARCK's effectiveness in the case of Eclipse 1.0 vs. 2.0 (minimum confidence fixed at 0.8) . . . . .	58
3.10	Influence of minimum confidence on LAMARCK's effectiveness in the case of Eclipse 1.0 vs. 2.0 (minimum support fixed at 10) . . . . .	58
4.1	Change in the <i>usage</i> of the <code>CompletionProposal</code> class that occurred in the Eclipse code between versions 3.4.2 and 3.5.2. . . . .	67
4.2	The four basic stages of LAMARCK. . . . .	70
4.3	The evolution pattern corresponding to the code change indicated on Figure 4.1. The pattern shows what was the context of the change (indicated by the "O" properties) and what was deleted and added (indicated by the "D" and "A" properties). . . . .	70
4.4	(a) the number of changed lines of code for each of the Eclipse modules in the transition from version 3.4.2 to version 3.5.2; (b) the number of evolution patterns each Eclipse module was part of in the transition from version 3.4.2 to version 3.5.2. . . . .	71
4.5	Correlation between implementation changes and usage changes for the case of Eclipse 3.4.2 to 3.5.2. Each point represents a module. . .	72
4.6	Correlation between implementation changes and usage changes (AspectJ 1.6.2 to 1.6.3). Each point represents a module. . . . .	73



# List of Tables

1.1	Lexical level tokens evolution for the RHINO project: method calls. . .	4
1.2	Architectural level tokens evolution for the RHINO project: imports. . .	5
1.3	Renamed method calls. . . . .	7
2.1	Precision of the import statements usage trends recommendations. . .	19
2.2	Usage of library versions for January 2009 . . . . .	27
2.3	Switching back to older library versions for the period January 2007– January 2009 . . . . .	28
3.1	Change properties for the <code>removeSelectionListener()</code> method. The properties of the context are also present. . . . .	44
3.2	An evolution pattern occurring in 170 Eclipse methods. . . . .	46
3.3	Evaluation Subjects . . . . .	48
3.4	Evaluation Subjects . . . . .	48
3.5	Reported unique violations and their success rate. . . . .	49
3.6	LAMARCK’s effectiveness in discovering inconsistently applied changes. The table summarizes the results obtained for 50 random splits of the input data. See Section 3.4 for details on the evaluation scheme. . . .	56
4.1	Evaluation Subjects. . . . .	66



# Chapter 1

## Introduction and Basic Concepts

*“There are two mistakes one can make along the road to truth –  
not going all the way, and not starting.”*  
– Buddha.<sup>1</sup>

During software development, a vast amount of data is being generated—every version of the project’s code is being stored in version archives; every reported defect is being saved in bug-tracking systems; every piece of communication is being kept in email and forum archives. As of July 2011, for example, the software development open-source community SourceForge.net hosted more than 300,000 projects and the bug databases only of Eclipse and Mozilla combined contained more than 700,000 reports. The data stored in all those repository systems can be used to better understand software development, to empirically validate ideas and techniques and to serve as a base for finding solutions to existing research and industry problems. The research field that mines all of those archived data is called *mining software archives* (MSR)—it is the broad field of my thesis.

This thesis is dedicated to analyzing the history of software projects and to learning specifics related to the evolution processes in those projects. Everything always evolves. On the path of evolution, mistakes may occur that have to be corrected. *In order to speed-up the evolution, the mistakes of the past need to be avoided in the future.* In the first part of my thesis, I have been exploring how different components of a

---

<sup>1</sup>Buddha (563—483 B.C.), also known as Hindu Prince Gautama Siddharta, was the founder of Buddhism.

software projects evolve and what specific behavior can be observed. In the second part of my thesis, I have observed patterns of behavior in this evolution. All these observations serve for a project to learn from its past. Old, hidden mistakes can be identified and new mistakes, resembling old ones, are prevented from being introduced into the project.

## 1.1 Contributions

The main focus on this thesis has been the analysis of the evolution of software components. This work began by conducting observations on the processes of evolution and the emerging trends in software components. Once this observation was in place, we were able to extract evolution patterns out of it and give structure to the project's evolution.

In order for this work to be useful and applicable in real-life scenarios, we made sure that all the algorithms and tools scale to real-world programs (all the tools and techniques developed are publicly available). We hope that our work will encourage further exploration of the presented topics and techniques, by the research community.

The remainder of this thesis is structured as follows:

- In Chapter 2, we present the concept of usage trends of software components and discuss in detail the different types of trends. The classification of the trends into specific categories, e.g. *increasing* or *decreasing*, is the first step into using those trends as predictors for the future of a software project. We call those trends *evolution usage trends*, as they are based on the usage statistics of the specific component throughout the time. The presented technique is incorporated into our AKTARI tool. We applied AKTARI to APIs and their versions in order to explore the question if the quality of an API can be predicted by its popularity.
- In Chapter 3, the focus falls on the evolution of objects. By mining the changes between two versions of a project, we were able to identify reoccurring change patterns. Those *evolution change patterns* are part of the evolution of a project and can serve to detect yet undetected defects, to prevent the occurrence of such defects and to generate a documentation of the way a specific object is supposed to be used. In this chapter we present our LAMARCK tool that has a prediction accuracy of almost 100%.
- In Chapter 4, we harness the power of the evolution change patterns, presented in the previous chapter, to evaluate the modularity of software modules. A module

is classified to have high modularity, if the changes in its implementation do not influence its usage.

The remainder of this chapter defines common terms used throughout the thesis, illustrating them with an initial example (Section 1.2), and lists the publications this thesis is build upon (Section 1.3). The thesis concludes with a summary of its contributions and ideas for future work in Chapter 5.

## 1.2 Example

The source code of a software project is always undergoing changes and is constantly evolving. The history of these changes is usually stored into a so called version control system (VCS), e.g. CVS<sup>2</sup> or SVN<sup>3</sup>, which keeps a copy of the project’s history—this history is called the *project’s archive*.

One can learn a lot about the project from analyzing the information stored in its archive. In this example we will illustrate, using *tokens*, how the evolutionary information, stored in the VCS can be used to improve the project. Here we use the JAVA definition for *token*:

**Definition 1 (Token)** *All characters used in source code are grouped into symbols, called **tokens**—a token represents some syntactic content of an element. There are several categories of tokens: identifiers, keywords, separator, operator, literal and comment.*

In order to extract the tokens from the version control system, we used the APFEL tool [44], which extracts all the tokens from a project’s CVS archive and stores them in a database (together with some additional information regarding the token location, time of deletion of the token, number of occurrences of the token, etc.). Based on this collected data, one can come up with project-specific trends, related to the analyzed tokens. An example of a trend can be seen in the last column of Table 1.1.

**Definition 2 (Evolution usage trend)** *An evolution trend is a graphical representation of the usage evolution of a software component, throughout a given time interval, based on the number of usages of this component at each discrete moment in time, during this interval.*

---

<sup>2</sup>Stands for “Concurrent Versions System”: <http://cvs.nongnu.org/>

<sup>3</sup>Stands for “Subversion”: <http://subversion.tigris.org/>

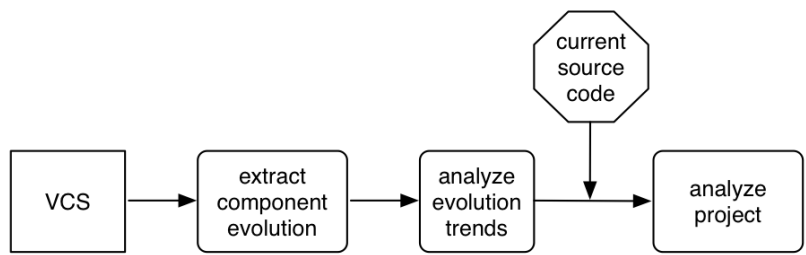


Figure 1.1: The process of extraction and usage of trends of software components.

	Number of occurrences per year									
Method Calls	1999	2000	2001	2002	2003	2004	2005	2006	2007	Evolution
getType	300	302	150	117	120	102	103	102	117	
getProperty	2	8	10	20	28	37	37	37	41	
getNextSibling	258	260	142	2	2	2	2	2	2	
addByteCode	248	220	108	106	0	0	0	0	0	
putProp	240	242	71	67	18	14	14	14	16	
reportConversionError	30	30	16	0	0	17	17	17	17	

Table 1.1: Lexical level tokens evolution for the RHINO project: method calls.

Figure 1.1 illustrates this whole process of extracting the tokens from the VCS and analyzing their evolution more in detail—starting from the source code archives stored in the VCS, we extract usage data for each token and create its evolution trend. Having those trends available, we can classify and analyze them into different trend types (e.g. increasing or decreasing). Once the evolution trends are built and analyzed, they can be used to analyze the project and serve different areas of the software development process, like defect prediction and detection, recourses optimization, etc. This general schema can be used to track the evolution of any kind of program component, e.g. objects, modules or entire libraries and is being used throughout this entire thesis (see Chapters 2, 3 and 4).


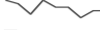



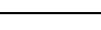
Imported classes	Number of occurrences per year									Evolution
	1999	2000	2001	2002	2003	2004	2005	2006	2007	
java.io.Serializable	0	0	4	9	10	10	10	13	15	
java.io.IOException	11	10	7	11	9	9	6	8	8	
java.util.Hashtable	32	32	16	12	14	14	13	13	13	
java.util.Vector	31	33	14	3	3	4	4	4	4	
java.util.Enumeration	23	26	9	8	7	6	3	3	3	
java.util.Stack	8	8	3	0	0	0	0	0	0	

Table 1.2: Architectural level tokens evolution for the RHINO project: imports.

### 1.2.1 Trend Types

As mentioned in the previous section and illustrated in Figure 1.1, after collecting the evolution trends we analyze and classify them. Let us take a closer look at the different types of trends that exist.

As one can see, from the depicted trends in the last column of Table 1.1, each trend's curve looks different. In order to be able to analyze those curves, we classify them into four main categories. We differentiate between four types of trends, based on the slope of the trend's curve. We have discrete data points and can compute the slope between each two data points. *The ratio between the positive and the negative slopes determines the overall curve slope*, i.e. if there are more segments with positive slope than with negative, the overall slope is determined as positive. Thus an **increasing trend** is a trend, which has a positive cumulative curve slope; a **decreasing trend** is a trend with negative cumulative curve slope; a **stable trend** has a slope of 0 (i.e. it is a straight line) and an **undecided** trend is a trend, whose type cannot be deterministically identified, because its segments have an equal number of positive and negative slopes.

Now let us take a closer look at some of the more interesting types of evolution trends. During the evolution of a project, gradually decreasing and gradually increasing numbers of components occurrences can usually be observed. One of the interesting cases is, for example, when a component has completely or almost completely disappeared from the code—then its usage is mapped to a decreasing evolution trend. This behavior may occur due to code restructuring or change in the project's requirements. No matter, however, what the reason for the disappearance of a component is—it *has been deleted on purpose* and as such, most probably, must not be re-used again. A location where such a component reappears after some time, is a potential code defect location.

Let us now take a look at a specific component example: “*Can it be said that the usage of the import statement `java.util.Stack` is obsolete?*” The general answer to such a question is “No”. However, when it comes to the RHINO<sup>4</sup> project, the answer is “Yes”. As it can be seen from Table 1.2, the number of usages of the *import `java.util.Stack`* statement goes down to zero over time and remains zero. This means that this import statement has become obsolete and re-using it in this particular project might contradict the established project structure, logics or architecture. In this case, a future usage of *import `java.util.Stack`* in the RHINO project can be classified as a violation of a project specific evolution trend. What such an evolution trend provides is project-specific information—no general tool would ever recommend against the usage of *`java.util.Stack`*, as it is a standard JAVA library class—in this particular, project-specific case however this is exactly the case and it could not have been spotted unless by analyzing the evolution behavior of this class in the specific project.

### 1.2.2 Applications

Let us now discuss briefly the possible applications of evolution trends. Those applications will later be illustrated more in detail in each separate chapter of this thesis.

#### Correction

Evolution trends can be used to discover unknown code defects. Knowledge about decreasing trends, for example, can be used to not only recommend possible substitutes to deprecated component, but to also discover such deprecated components that still exist at some code locations. This can be done by checking the entire project code against the already collected trends and detect possible trend violations. Such deprecated tokens can exist for example, because of backward-compatibility or defective code. With this approach can be detected those unknown, but still existing code smells and defects, which might cause a post-release failure of the project.

Going a step further, based on the similarity of the contexts of two trends, further correction recommendations can be issued. A warning message of the following kind can guide the developer: “*Based on the evolution trend of component A at location X and the code similarity between location X and Y, please apply the same evolution trend behavior of component A for component B at location Y.*”

---

<sup>4</sup>RHINO is an open-source implementation of JavaScript written entirely in Java. It is typically embedded into Java applications to provide scripting to end users. <http://www.mozilla.org/rhino/>



old method call	new method call
addForwardGoto	addGotoOp
badArg	argBug
generateCodeFromNode	generateExpression
getNextSibling	getNext
reportError	reportSyntaxError
signature	scriptSignature

Table 1.3: Renamed method calls.

### Prediction

Having the evolution trends available can be of help for planning the resources for the future development and support of the software project. The evolution trends make it possible to predict how the evolution of a component will continue. Based on this information, project managers can create plans for the following months of the project, deciding on which weak or strong components to focus on. For example, when a software project manager needs to take a decision regarding the distribution of the testing resources, she might decide to focus her team's attention on those components, which are exhibiting unexpected evolution behavior. What is expected behavior, one can deduce both from the specifications of the project and from the history of a component and its evolution trend (e.g. a decreasing trend should continue to decrease—see Chapter 2).

### Trends Correlations

Having available all the evolution data, one can also detect correlations between component trends. One can, for example, detect pairs of trends, where one trend rises when some other decreases [44]. Having the components trend information available, makes it possible to detect the deleted components and to give an appropriate recommendation for their substitution. Such recommendation can either be given at real-time, while the developer is typing the code and is about to use a component with a decreasing trend, or at a random moment in time, by just running the detected trend correlations against the project's code.

For example, after investigating the behavior of the method call tokens in RHINO (see Table 1.1), we have discovered several method calls substitutions (see Table 1.3), based on the decreasing number of occurrences of some tokens and the increasing number of occurrences of others. In RHINO, we detected 31 different method calls

substitutions, e.g. `addForwardGoto` has been substituted with `addGotoOp`.

Another example of trends correlations is analyzing the evolutionary behavior of `import` statements. An observation that the usage of one particular `import` statement has been decreasing, while simultaneously the usage of another `import` statement has been increasing with the same rate, can lead to recommendations of the kind “Use *import statement X*, instead of *import statement Y*.” This scenario can occur, for example, when an `import` class becomes deprecated.

Detecting correlations between trends can be of great value to developers, as it not only serves as a warning against the usage of the deprecated component, but also as a suggestion for which other component can be used instead.

## 1.3 Publications

Ground results for the work presented in this thesis have been published in the following publications:

- **Yana Mileva** and Andreas Zeller. Project-Specific Deletion Patterns. In *RSSE’08: Proceedings of the 2008 international workshop on Recommendation systems for software engineering*, pages 41–42, New York, NY, USA, 2008. ACM.
- **Yana Mileva**, Valentin Dallmeier, Martin Burger and Andreas Zeller. Mining trends of library usage. In *IWPSE-Evol’09: Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IW-PSE) and software evolution (Evol) workshops*, pages 57–62, New York, NY, USA, 2009. ACM.
- **Yana Mileva** Valentin Dallmeier and Andreas Zeller. Mining API popularity. In *TAIC PART’10: Proceedings of the 5th international academic and industrial conference on Testing - practice and research techniques*, pages 173–180, Berlin, Heidelberg, 2010. Springer-Verlag.
- **Yana Mileva**, Andrzej Wasylkowski and Andreas Zeller. Mining evolution of object usage. In *ECOO’11: Proceedings of the 25th European conference on Object-oriented programming*, pages 105–129, Berlin, Heidelberg, 2011. Springer-Verlag.
- **Yana Mileva** and Andreas Zeller. Assessing Modularity via Usage Changes. In *PASTE’11: Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on*

*Program analysis for software tools*, pages 37–40, New York, NY, USA, 2011. ACM.

## 1.4 Summary

This first chapter introduces the field of mining software archives (MSA) and gives an initial view of the rest of this theses, as well as presents the concepts of an *evolution trend* and *trend types*. We saw a first example of evolution trends of specific software components and explored some of the applications the evolution trends have.

After this initial introduction, we are ready to go deeper into the different types of evolution trends and components analysis. This first chapter introduced a general approach, used throughout this thesis, for collecting evolution data and extracting evolution usage information of a specific software component. The rest of the thesis explores how the evolution of different software components look like and what implications it has on the project as a whole.



## Chapter 2

# Evolution of API Components

*“I’m an inventor. I became interested in long-term trends because an invention has to make sense in the world in which it is finished, not the world in which it is started.”*

– Ray Kurzweil<sup>1</sup>

When designing a piece of software, one frequently must choose between multiple external libraries and library versions that provide similar services. Which library is the best one to use? Is the latest version the best one to use? Has it been widely adopted already?

We mined hundreds of open-source projects and their external dependencies in order to observe the popularity of their APIs and to give recommendations of the kind: *“Projects are moving away from this API component. Consider a change.”* We consider the open-source community to be a crowd of experts and want to extract and learn from their experience. Such wisdom of the crowds can provide valuable information to both the API users and the API producers, by helping them avoid pitfalls experienced by other developers, and by showing important emerging trends in API usage.

Preliminary results of the work presented in this chapter are published at IWPSE-Evol’09 [25] and TAIC PART’10 [26].

---

<sup>1</sup>Raymond “Ray” Kurzweil is an American author, scientist, inventor and futurist. He is involved in fields such as optical character recognition (OCR), text-to-speech synthesis, speech recognition technology, and electronic keyboard instruments.

## 2.1 Introduction

Nowadays, hardly any software project exists that does not use external libraries and their APIs. However, despite this strong connection between projects and external libraries, no proper manner of evaluating the quality and success of an API and its versions exists. Means of communication like emails, newsgroups and bug-tracking systems are indeed present, and they might be concerned with the API quality, but this information is not to be found there in a structured or unbiased way. For instance, the absence of a bug report, might mean that the API is not being used at all, or that it is being heavily used, successful and free of bugs. *In our research we consider the usage popularity of an API to be indicative of its success. We consider the lack of popularity or the decrease in popularity to be indicative of lack of success. Popularity of an API is measured by the number of its users.*

## 2.2 Early Adopters vs. Late Followers

The work presented in this chapter, is based on evaluating the popular choices of API users and aims at also predicting the users' behavior in the future. Before we delve into the specifics of the technique, I would like to elaborate more on the source of the ideas we got for this direction of our research.

**Wisdom of the crowds principle** The collective knowledge is greater than the knowledge of the individual.

This observation is based on the book *The Wisdom of the Crowds* [37], where the author, James Surowiecki, goes at length about the aggregation of information in groups that results in decisions, which are better than the decision any single individual could have taken. The observations of the author are supported by numerous case studies, coming primarily from the fields of economics and psychology.

While in the case of Surowiecki's book, the crowds he describes are crowds of random people, in our case we restrict the domain of the crowd to the developers community. In this thesis many observations are made and conclusions are drawn based on the wisdom of the crowd of *experts*, as we consider the open-source community a community of experts, when it comes to API usage. Let me give an example: when discussing the popularity of a given library, we rely on the decision the crowd of users takes, related to this library, i.e. whether to use this library or not. In this case the usage experts related to a library's usage are its users, thus we rely on the decision of the crowd of those experts, when drawing conclusions or issuing recommendations.

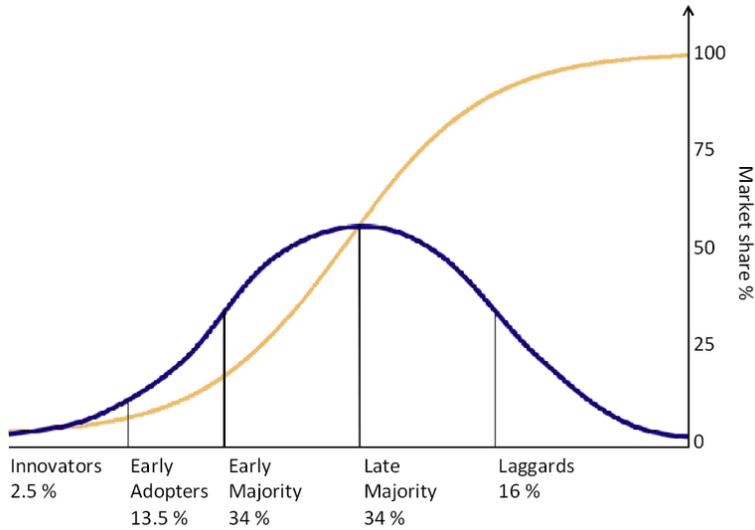


Figure 2.1: The diffusion of innovations according to Rogers.

**Definition 3 (Early adopter)** *An early adopter is an individual, who adopts a technology as soon as it is available.*

**Definition 4 (Late follower)** *A late follower is an individual, who adopts a technology after a certain period of time.*

The above two definitions are based on Everett Rogers' theory of *Diffusion of innovation* [35] that groups the adopters in five categories: *innovators*, *early adopters*, *early majority*, *late majority* and *laggards*. Innovators are the first individuals to adopt an innovation; the early adopters are the second fastest category, who adopt an innovation; the early majority adopts an innovation after a varying degree of time; the late majority adopts an innovation after the average individual of the society adopts it and the laggards are the last to adopt an innovation. Those different categories have been introduced to describe the process of adoption of a new product or technology. According to Rogers' theory the adoption of an innovation follows an S curve (see Figure 2.1<sup>2</sup>).

<sup>2</sup>Source: Wikipedia [http://en.wikipedia.org/wiki/Diffusion\\_of\\_innovations](http://en.wikipedia.org/wiki/Diffusion_of_innovations). Accessed on December 19, 2011.

We have simplified Rogers’ theory for the needs of our research and differentiate only between two types of adopters – *early adopters* and *late followers*. In our case the early adopters group contains Rogers’ groups of innovators and early adopters and the late followers group contains the early majority, late majority and laggards.

To be more specific, when it comes to library versions usage, for example, users can be classified in a similar way—those who start using the newest library version immediately after it is out and those who prefer to wait and see if it is safe to switch to. Every user has reasons for such behavior, such as the need for new functionality (early adopters) or security (late followers). *Our analysis is being fed by the data collected from the early adopters and is being used to give recommendations to the late followers.*

We have developed a tool, called AKTARI<sup>3</sup> (see Section 2.5), that analyses the adoption of libraries and their versions, based on the early adopters data and is able to issue recommendations that will concern the late followers. Early adopters are rarely influenced by recommendations, as they know about the risks and benefits of being the first ones to use a new library or library version and their reasons for switching are not risk-driven. However, our approach is able to assist late followers in making an informed decision about which library and library version to use. *Even though AKTARI is specifically developed to deal with libraries, the approach itself is applicable to the adoption of any new technology in the life cycle of a software project.*

## 2.3 Evolution of APIs

In order to leverage the wisdom of the crowds of API users, we need to mine a large body of projects. For this purpose we collected information from hundreds of open-source projects and analyzed the overall global API elements usage trends. We evaluate the popularity of an API, based on the popularity of its components (classes and interfaces). To explore the API elements usage, we mined information from the source code history of 200 APACHE<sup>4</sup> and SOURCEFORGE<sup>5</sup> projects.

Let us give an example of an API component popularity trend. As seen on Figure 2.2, the `java.io.StringBufferInputStream` class is not being very popular and its usage is declining. Investigation of the reasons behind this drop in popularity showed that the class is in fact defective. Such information regarding API elements can serve both the API users as well as the API producers.

---

<sup>3</sup>“Aktari” is the Swahili word for “crowd”.

<sup>4</sup><http://www.apache.org/>

<sup>5</sup><http://sourceforge.net/>



### 2.3.1 API Users and Producers

Even though external libraries are an important part of projects' source code, no direct feedback means exist that can determine and display the popularity of an API.

When analyzing API popularity we focus on the “weak spots” of the API, i.e. those elements that are unpopular or declining in popularity. Having such information available can be beneficial for the following two groups:

**API users.** The software developers who use external libraries in their projects can profit from knowing which are the “weak spots” in an API. If a lack of or a decrease in the usage of an API element exists this is an indication that the majority of the users prefer not to use it. Such information will help the software developers make better choices regarding the usage of API elements by avoiding the indicated bad experience of their peers.

**API producers.** In order to deliver a better product, the library developers need to know how their API is being used by the end user. Once they identify the “weak spots” of their API, they can direct their attention to investigating why the popularity is such. Similarly to market analysis results, such data can help the API producers provide a better product as a whole based on the preferences of their users. In this case we consider an API producer any member of the team involved in the API production—developers, testers and managers.

After discussing the importance of API popularity data, in the next section we present our approach to collecting and analyzing API usage trends.

### 2.3.2 Collecting and Presenting Usage Data

For our analysis we used projects from both APACHE and SOURCEFORGE—two of the most widely used open-source repositories. We downloaded and analyzed the source code of 200 projects (50 from SOURCEFORGE and 150 from APACHE) for the period beginning of January 2008—end of January 2009).

To analyze the popularity of an API at the given time period, we counted the number of projects in which each API element was used in that period. In order to do that we analyzed the number of projects per month that used a specified `import` statement. By analyzing the usage of each import statement, we implicitly analyze the overall usage of the API itself and explicitly analyze the API elements. From January 2008 to January 2009, in the 200 projects, we detected the usage of 23 401 unique `import` statements.

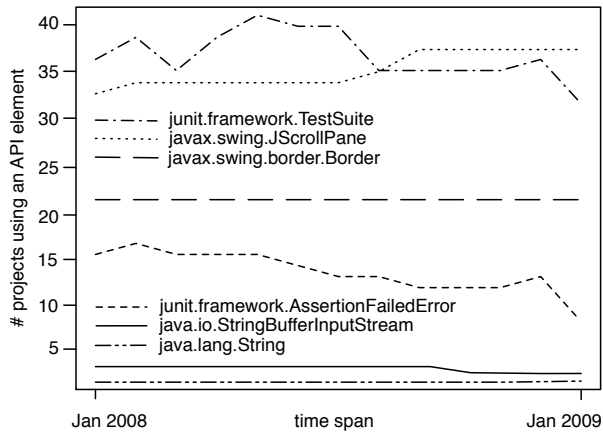


Figure 2.2: Examples of the four usage trend types.

Our AKTARI tool analyses the collected data and plots a usage trend (for a definition please refer to Section 1.2) for the collected API elements (see Figure 2.2). The data is accumulated over the entire number of projects that use the specified `import` statement. Based on that data the usage trends are being plotted. AKTARI also comes as an interactive web-version (see Figures 2.7 and 2.8) that plots the collected data for the user<sup>6</sup>.

Our popularity analysis is straightforward in its nature—we compute the number of times per month a specific API element is used throughout the 200 open-source projects we mine. The usage recommendations, however, that our AKTARI tool gives based on this popularity analysis, are not so straightforward and its evaluation will be addressed in the next sections.

### 2.3.3 Predicting Usage Trends

Due to the nature of software projects development, the API elements usage trends are also quite diverse in their shapes. As it is not possible to frame the trends into any other general way, we differentiate between **four main types of trends, based on the slope of the line** (see Section 1.2.1): *increasing* trend (`javax.swing.JScrollPane`), *decreasing* trend (`junit.framework.AssertionFailedError`), and a trend that remains *stable* (`javax.swing.border.Border`)—see Figure 2.2. The trends

<sup>6</sup><http://www.st.cs.uni-saarland.de/softevo/aktari.php>

that cannot be classified in either of the above categories we classify as *undecided* (e.g. `junit.framework.TestSuite`)

All kinds of trends carry valuable information, but the ones that are most interesting for us are the *decreasing* ones. A decrease in the usage of an API element can be due to factors like: defects in the API, deprecated classes or methods, or the availability of a different and better API. However, no matter what the reasons behind a usage drop are, the API users should be warned about this element. When detecting a decreasing trend or an unpopular API element (element used by less than 1% of the projects) our AKTARI tool issues a warning of the kind “*Projects are moving away from this API element. Consider a change.*” This warning is based on the history and the type of the trend—the history of usage is indicative of the future of usage<sup>7</sup>. This message serves to raise the awareness of the API users of potential problems with the specific API. It is also a red flag for the API producers.

Because of the high interest in the shape of the trend, we would like to be able to predict how the trend will continue to behave, in order to use that information in recommending for or against the API usage. Due to the diverse nature of the shapes of the trend curves, it is hard to find a predicting method that will work for all types of curves. However, as we are interested only in the general direction of a trend, i.e. if the trend is *increasing*, *decreasing* or *stable*, **linear regression** will give us descriptive enough results for our evaluation. Linear regression is a statistical approach, that assumes that the relationship between the variables is linear. One of its applications is the goal of predicting and forecasting of events.

**Definition 5 (Linear Regression)** *In statistics, simple linear regression is the least squares estimator of a linear regression model with a single explanatory variable. In other words, simple linear regression fits a straight line through the set of  $n$  points in such a way that makes the sum of squared residuals of the model (that is, vertical distances between the points of the data set and the fitted line) as small as possible<sup>8</sup>.*

When dealing with real-life scenarios one often has to choose between speed and precision. In the presented work we had to choose the best prediction model for the task and decided for linear regression, due to its speed related to other models and the type of the data we mine. For completeness however, we will mention a few other models available for predicting the behavior of trends.

---

<sup>7</sup>An astute reader will notice that the situation may occur, where a *sudden* drop or rise in the trend happens. Unfortunately such sudden changes are not possible to predict, without having some other history data indicative of it. A way to solve that problem would be to examine only a specific period of the entire history span—the last 6 months, for example and make predictions based on that.

<sup>8</sup>Source: Wikipedia [http://en.wikipedia.org/wiki/Simple\\_regression](http://en.wikipedia.org/wiki/Simple_regression) Accessed on February 13th, 2012.

**Reverse arrangement test (RAT)** The RAT test is a statistical test for detecting trends, that makes no assumption about what model a trend might follow (while linear regression assumes linearity). This test assumes an equal interval of time between each event.

**Multivariate adaptive regression splines (MARS)** The MARS analysis is a form of regression analysis that can be seen as an extension of linear models, i.e. the MARS models are more flexible than linear regression models.

**Autoregressive Integrated Moving Average (ARIMA)** In statistics and econometrics, and in particular in time series analysis, an ARIMA model is fitted to time series data either to better understand the data or to predict future points in the series (forecasting). ARIMA helps to choose a model that best fits the time series.

As the prediction model of linear regression fits well our needs it is the model we chose for our evaluation. The evaluation results are discussed in the following section.

### 2.3.4 Evaluation

Usage popularity data and trends carry valuable information to the API producers as of how popular their API and its elements are (also in comparison to other similar libraries) and thus assists them in taking future maintenance and management decisions. As mentioned earlier, such usage or popularity data is also useful for the users of an API. That is why it is important to be able to predict the future behavior of a given usage trend.

Our AKTARI tool is able to offer usage recommendations, based on the past usage trends of an API element. More specifically, if a decrease in popularity or low popularity of the specified API element is observed, we will warn against the usage of this element.

In this section we proceed to evaluate the correctness of the above statements and the usefulness of the technique, from both quantitative and qualitative perspective.

#### Quantitative Evaluation

We analyzed the past usage trends of API elements as accumulated over all the 200 analyzed projects. Based on the notion that the type of the past accumulated usage trend will not change in the future, we give usage recommendations regarding a specific API element.

*Hypothesis 1: The past usage trend of an API element is predictive of the future usage trend of the same element.*

Table 2.1: Precision of the import statements usage trends recommendations.

Trends		Actual		
		Increasing	Stable	Decreasing
Predicted	Increasing	<b>67%</b>	33%	0%
	Stable	1%	<b>98%</b>	1%
	Decreasing	0%	18 %	<b>82%</b>

In order to evaluate the correctness of our recommendations we took the usage trends of the `import` statements for the period beginning of January 2008—end of January 2009 (13 months) and split the period it into two parts—2/3 vs. 1/3. We wanted to see if the trend that we observe in the first period will continue to have the same type during the second period. In order to check that, we used linear regression on the first period January 2008—October 2008 (8 months) and on the entire period (all 13 months) and compared the results for each import statement for the first period and for the entire period.

Using linear regression prediction model we were able to predict the behavior of the evolution trend. We have evaluated our results in terms of *precision* and *recall* (see Table 2.1).

**Definition 6 (Precision)** *Precision is the fraction of retrieved instances that are relevant. The precision values range from 0 to 1 (or 0% to 100%), where high precision means that an algorithm returned more relevant results than irrelevant. A perfect precision score of 1 means that every result was relevant.*

**Definition 7 (Recall)** *Recall is the fraction of relevant instances that are retrieved. The recall values range from 0 to 1 (or 0% to 100%), where high recall means that an algorithm returned most of the relevant results. A perfect recall score of 1 means that all relevant results were retrieved.*

As it is evident from Table 2.1, we have very high precision values when it comes to predicting that an increasing trend will continue increasing, a stable trend will remain stable and a decreasing trend will keep on decreasing. Note that, due to the nature of software projects development, it is very rare to observe an accumulated usage trend that has no fluctuations in its shape, thus a precision close to 100% in all cases would not be reflective of the real software development process. What these results show is that the general evolution trend of a software component usually remains stable.

The recall values, we computed, are also quite high—ranging from 0.82 to 0.95.

Finally, our accumulated evaluation showed that in total in 84% of the cases our prediction of the future trend and thus our recommendations are valid.

Those results are a clear evidence that our recommendations regarding the future behavior of usage trends are accurate. In conclusion of the evaluation of our hypothesis, we can state that:

*The past usage trend of an API element is indeed predictive of the future usage trend of the same element.*

### Qualitative Evaluation

One can expect that with the evolution and growth of projects the usage of the specific `import` statements will increase. This is indeed true for those libraries and their APIs that are widely used by the software developers (*javax.swing.JScrollPane*, see Figure 2.2). However, there are cases where the usage is decreasing. As mentioned before, we are mainly interested in those cases where a decrease in the usage is present or the actual usage is very low. We assume that such usage (or low popularity) of a specific import statement is indicating that this API element is starting to be considered outdated, obsolete or defective.

*Hypothesis 2: One should consider the library's popularity before using it.*

In Figure 2.2, one can see examples of classes with decreasing or low usage. We manually investigated the possible reasons behind some of those trends. Here are a few interesting cases:

***java.lang.String:*** Even though the `String` class is a widely used class, there is an explicit rule in the Java Coding Convention stating that this class should never be imported. Thus a developer importing it violates the Java Coding Conventions and introduces a **code smell** in the source code. A code smell is a source code issue that does not currently lead to failures of the program, but might do so in the future.

***junit.framework.AssertionFailedError:*** In the case of this class we found a commit log message reporting an issue with `AssertionFailedError` when using it in combination with the `jmock` library. This is an example of a library **compatibility problem** for projects using these libraries.

***java.io.StringBufferInputStream:*** In this case, the `StringBufferInputStream` class is declared deprecated, as it does not correctly convert characters into bytes,

i.e. it is erroneous. Even though the documentation of the API explicitly states that the class `StringReader` should be used instead, there are still projects that use the deprecated `StringBufferInputStream` class. Based on the reasons why this class was deprecated (wrong conversion of characters) one can assume that those projects that are still using it are susceptible to **code defects**.

These examples show real source code problems, introduced due to the inappropriate usage of API elements. This allows us to confirm our hypothesis and to state that:

*Before using a specific API element, one should consider its popularity gradient.*

### 2.3.5 Threats to Validity

As any empirical study, this study has limitations that must be considered when interpreting its results.

**The approach may cancel itself.** When one gives recommendations based on past trends, there is always the question: what will happen if everybody indeed starts following the recommendations—will those recommendations not cancel the future refreshment of their own source? This issue is solved by the introduction of the notion of **early adopters** and **late followers** [25] (see Section 2.2). This notion states that there will always be users, who take the role of early adopters of a technology (in our case API element), despite the risks and the recommendations.

**The approach needs more data.** For our approach to give correct information and recommendations, we need to have enough data available. It might happen that in the data body we have, there are only few projects using a specific library—in this case if even one project moves away from this library this will influence greatly the evolution trend type. We believe that this is a problem all prediction techniques carry and nobody knows when the collected data is enough. This problem can be avoided to some extent, though not completely, by using thresholds for the prediction data reported, based on the overall number of usages of a library, compared to the whole data body.

**The approach is applicable only to Java projects.** It might seem at first glance that our approach is designed strictly for Java projects, as our examples are for Java APIs. However the analysis of the `import` statements can be easily transferred to analysis of imports in any other language—the way we scan through a Java source file, can be used to scan through any other programming language source file.

## 2.4 Evolution of API Versions

Most of today's software projects heavily depend on the usage of external libraries. Each of those libraries comes in different versions. One would assume that it is wise to always rely on the latest library version, as it is the version, which is currently the most refined one, well-structured and bug-free. However, in practice, things are different.

In this section, we leverage the wisdom of the crowds, when it comes to the usage of individual library versions. We consider the choice of the majority to be the wise choice in regard to which library version should be used. To explore the choices made, we mined information from the history of 250 APACHE<sup>9</sup> projects and the external libraries they use, in order to answer the following question: *Which library versions are the most popular ones?*

Mining the usage of an API version vs. mining the usage of the API itself is trickier, as usually there is no straightforward way to identify the API version used in the majority of the software projects. As our focus is analysis of JAVA projects we could focus on those projects, that are managed by MAVEN<sup>10</sup>, a widely adopted project management tool for JAVA projects. In MAVEN, library dependencies are stored explicitly in meta information files, which makes it possible to extract and analyze version usage.

To analyze the global usage of library versions, we used 250 real-life open-source projects from the APACHE foundation<sup>11</sup>, one of the largest and most popular repositories for open-source JAVA projects. In total, we analyzed the usage of 450 different external libraries versions per month over the period of two years.

### 2.4.1 API Users and Producers

The decision to use a specific library version usually depends on factors like reliability, functionality, usability, documentation, quality and compatibility. All of those factors come into play when a user decides whether to use a specific library version or not. By analyzing the choices made by software developers with respect to the usage of library versions, we estimate which are the most popular ones and thus the ones recommended for usage.

Having such information available can be valuable for two groups of developers:

**Library users.** Library users can highly profit from knowing how frequently a particular library version is used by the majority. Suppose, for example, that many

---

<sup>9</sup><http://www.apache.org/>

<sup>10</sup><http://maven.apache.org/>

<sup>11</sup>The reader might notice that we use different data sets for evaluating the techniques presented in Section 2.3 and Section 2.4. The data set from Section 2.4 can be used for evaluating the Section 2.3 technique, but the other way around is not possible, due to the need to use only Maven-managed projects.



people have switched back from a particular version—due to a bug in it. Then, warning potential new users of this version can help them avoid facing the same issues again. Users can thus save time and improve the quality of their software product.

**Library developers.** Up to now, library developers did not have any other means for getting user feedback except direct communication with the users. With our approach, they can evaluate how successful a particular version is and thus improve the future development of their library. Our technique thus gives library developers a means of collecting indirect feedback from users to provide better service.

## 2.4.2 Extracting Version Dependencies

In a project managed by MAVEN, meta data describing the project data is represented by MAVEN's *Project Object Model* (POM). MAVEN relies on the presence of a central repository that stores different versions of commonly used libraries.

Since its first release in 2002, MAVEN has become one of the leading open-source project management tools for JAVA projects. The actual number of projects using MAVEN cannot be measured directly. However, in September 2008, the central MAVEN repository was hit over 250 million times [16]. Usually, a MAVEN installation checks the central repository only once per day, which implies that MAVEN is actively used by a large number of developers.

In MAVEN, required libraries are described as dependencies in an XML file called *pom.xml*, a descriptive declaration of a POM. Library usage information is stored in `<dependency>` elements. These elements list all the libraries that the project depends on. Each element has three mandatory children: a *group id* (company, team, organization, or other group), an *artifact id* (unique id under group id that represents a single dependency), and a *version* (specific release). These three components uniquely identify a specific version of a library. Thus, `<dependency>` elements define all required libraries *unambiguously*, including their version number. Figure 2.3 shows an excerpt from a *pom.xml* file.

To analyze the usage of library versions, we collected dependencies for all of the 250 APACHE projects on a monthly basis over a period of two years. For each month, we determined the library usage information for all dependencies referenced by at least one project.

```

<project>
  <dependencies>
    <dependency>
      <groupId>servlet-api</groupId>
      <artifactId>javax.servlet</artifactId>
      <version>2.3</version>
    </dependency>
  </dependencies>
</project>

```

Figure 2.3: Declaring a dependency to the *servlet-api* library in a MAVEN Project Object Model

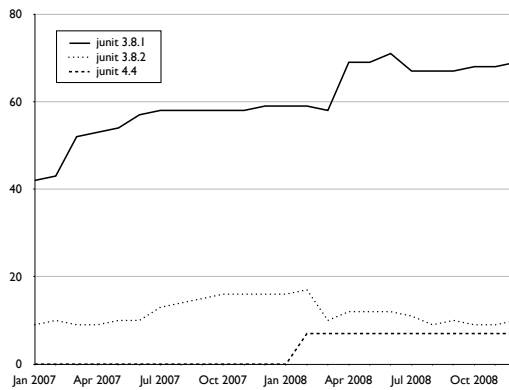


Figure 2.4: Usage trends of the *junit* library

### 2.4.3 Usage trends

Suppose you are a head of a team developing a library. How can you make sure that you are providing the best service for its users? In order to determine the evolution of the usage of libraries and their versions, we mined the entire archive history of the aforementioned 250 APACHE open-source projects for the period January 2007 to January 2009 (excluding).

As a first example, consider the usage trend of the *junit* library for this period. In Figure 2.4, one can see that version 3.8.1, which is the oldest version of this library used in that period of time, remained the most popular and widely used one. It was even more popular than the latest 4.4 version. From a user's point of view, we investigated the reasons behind this behavior and found out that there was a big API change from 3.x to 4.x. There are examples of projects' problem reports where it is stated that switching

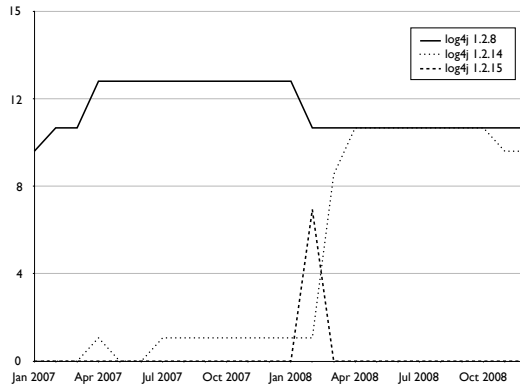


Figure 2.5: Usage trends of the *log4j* library

will be “lots of work” (e.g. *Gentoo bug entry 129773*). Also due to the change in the API there was an issue with backwards-compatibility—we found examples of users who decided to stick to the older versions as the *4.x* ones had compatibility issues with the *ant* library. Also, the *4.x* versions required a newer *jdk* version; the developers were concerned that this might be a reason for their clients to not use their product.

Figure 2.5 shows another example of usage trends of the different versions of the *log4j* library. The usage of *log4j 1.2.8* was at its highest point in mid-2007. What strikes however is the usage of the *log4j 1.2.15* version: At the moment it was released, there was a peak in its usage history and then a fast and sudden drop shortly afterwards. This was due to a bug in its implementation (see *Apache bug entry 43304*). Projects decided to switch to the new version (indicated by a drop in the usage of the *1.2.8*), but after discovering the bug they switched back (drop in *1.2.15*) and switched to the closest earlier version (increase in *1.2.14*). The fact in this case is that *1.2.15* was rejected by its users.

Those are only a few of the many examples of trends in library versions usage. For developers of libraries, such trends clarify the ways the users are using individual versions. As developers know what the differences between each version are, they can link the specific library version features to the library popularity and thus analyze the users’ needs.

*Trends in library usage are a method for displaying the preferences of the users.*

### Most popular versions

When a software developer decides to use a library, she has to decide which of the available versions is the best one to use. As this choice is made at a certain fixed moment in time, the recommendation should also be based on the usage at this particular moment in time.

To identify the most popular and thus recommendable library versions for usage in January 2009, we mined the 250 projects and their library dependencies for January 2009. Some of our results are depicted in Table 2.2.

To measure the popularity of a particular version, we consider the number of current usages of the version the user wants to switch from and the number of current usages of the version the user wants to switch to. For example, for a developer using *derby 10.1* that wants to switch to *derby 10.2*, we would recommend *not* to do so, as in only

$$\frac{\# \text{ derby } 10.2}{\# \text{ derby } 10.2 + \# \text{ derby } 10.1} = \frac{1}{1 + 6} = 14\%$$

of the cases version *10.2* is used. We again investigated the reasons behind this usage behavior and found a commit message stating that the developers will stick to version *10.1*, “until TranQL can handle *10.2*”, which reveals a compatibility problem in the newer *10.2* version. Developers should be warned about such issues with the versions in order to make a better informed choice as of which version is recommendable for them to use.

Of course, every developer may have individual reasons when and why to switch to a new version. However, if a large majority of developers uses a specific library version (e.g. *junit 3.8.1*) this information should be taken into account.

*Knowledge about popular versions helps developers in deciding,  
which versions to choose.*

### Switching back to earlier versions

When the developers of a software project switch from an old library version to a newer one, they usually do so either because the old version had problems that were fixed in the new one or because the new one offered more and/or better functionality. On the other hand, there are users who prefer to wait before they switch—such that once they switch, they will not have problems with a defective library version. However, identifying when it is safe to switch is a difficult task. Here again, the vote of the majority comes into play.

Table 2.2: Usage of library versions for January 2009

Library name and version	Times used
junit 3.8.1	60
junit 3.8.2	9
junit 4.4	7
log4j 1.2.8	10
log4j 1.2.14	9
log4j 1.2.15	0
servlet-api 2.3	4
servlet-api 2.5	1
derby 10.1	6
derby 10.2	1

For projects that migrate early to a new library version, it might be that they *migrate back* to an older version. In most cases, the reason for switching back is that the new version has some issues that make it unusable for the specific needs of the project. If this is the case, the end user should be *warned* about such library versions and should avoid switching to them.

Again, we have mined the same 250 APACHE projects and their history (January 2007–January 2009). However, this time we were interested in the number of times people switched back from a particular library version.

Table 2.3 presents the cases of the *junit*, *log4j*, *servlet-api* and *derby* libraries (these libraries are among the top most widely used libraries in our set of projects for the specified period). The first column in the Table gives the library name and version. The second column shows the number of times a particular library version was used in this period. The third column shows how frequently this specific library version was discarded, and the developers switched back to an *older* version in the same period. The fourth column gives the percentage of times a particular library version was switched back from.

Switching back and forth between versions is very time consuming and can introduce bugs into the code (if, for example, the library API has changed, the project code also has to be changed). That is why developers usually do not switch back from a particular library version once they have switched to it—unless it really has a problem. As one can see, the most popular version of *junit* is 3.8.1. No project ever switched

Table 2.3: Switching back to older library versions for the period January 2007–January 2009

Library	# usages	# switched back	%
junit 3.8.1	1501	0	0%
junit 3.8.2	293	1	<1%
junit 4.4	84	0	0%
log4j 1.2.8	269	3	2%
log4j 1.2.14	114	0	0%
<b>log4j 1.2.15</b>	<b>7</b>	<b>4</b>	<b>57%</b>
servlet-api 2.3	182	0	0%
servlet-api 2.5	10	1	10%
derby 10.1	147	0	0%
derby 10.2	31	0	0%

back from using this version, thus indicating that this is a very good version of *junit* to use. The *servlet-api 2.5* version, for example, was switched back from in 10% of the cases. We found bug reports pointing to a problem the 2.5 version has with *Tomcat 5.5* (e.g. *Grails bug entry 2053*). The case that strikes the most, however, is the *log4j 1.2.15* version, as in no less than 57% of the cases people switched back from it. We investigated this case and found out that the reason why so many users decided to switch back is a bug in this version that prohibited its usage for all MAVEN projects that depended on *log4j*, but did not depend on the *java mail* and *jms* libraries. The problem with this library version is also indicated in Figure 2.5 and Table 2.2, and was thus detected by all of our techniques.

Finding reverts to previously used versions can show the library developers how big the impact of a library issue is. It can also show the library users how reliable a specific library version is and thus give them yet another indication if they should switch to it.

*The number of times a library version was switched back from is a strong indicator of the quality of the library.*

#### 2.4.4 Threats to Validity

As any empirical study, this study has limitations that must be considered when interpreting its results. We identified the following threats to validity.

**The number of projects may affect the outcome.** It is possible that the addition or the removal of a particular project from the set of analyzed projects might influence the results. However, we have mined hundreds of projects and we believe that this possibility is small.

**Results might not hold for non-MAVEN projects.** The advantage of MAVEN is that it eases dependency management. However, it does not impose restrictions on which version of a library can be used. Also, the libraries used in a project depend on the scope of the project and not on its management tools. We are therefore convinced that our results are also valid for projects that do not use MAVEN.

**The implementation may have errors.** A final source of threats is that our implementation could contain errors that affect the outcome. To control these threats we did a careful cross-checking of the data and the results to eliminate mistakes in the best possible way.

## 2.5 The Aktari Tool

In this chapter we presented approaches for analyzing the popularity trends of APIs and API versions. The presented techniques are combined in our general API analysis AKTARI<sup>12</sup> platform. Our recommendation is for the users to use the tool, when taking a decision related to APIs and their versions.

When one needs to take a decision related to a specific API, AKTARI is able to display the specific trend that relates to this API (see Figure 2.2). Having this information available an API user can decide if it is recommendable to use a specific API and an API producer can evaluate where her API stands in the global API usage picture. A future planned extension of AKTARI will be able to display the trends for groups of APIs that are related to each other (i.e. GUI APIs), so that one can easily do the comparison between the usage trends of those APIs that are in fact competitors to each other.

When one needs to take a decision related to a specific API version, our recommendation is to consider all the available information that the tool can provide.

When giving recommendations as to which library to use, one should take into account as many factors as possible. For new projects, backwards-compatibility for their clients is not an issue—in this case considering only the popularity at a certain moment in time might be misleading, as it does not take into account all kinds of

---

<sup>12</sup>“Aktari” is the Swahili word for “crowd”.






				
Aktari	Javadoc	Declaration	Console	Progress
Library	Version	Popularity	Switched back from	Trend
log4j	1.2.15	0%	57%	Stable
junit	3.8.1	84%	0%	Increasing

Figure 2.6: AKTARI—Eclipse plug-in design

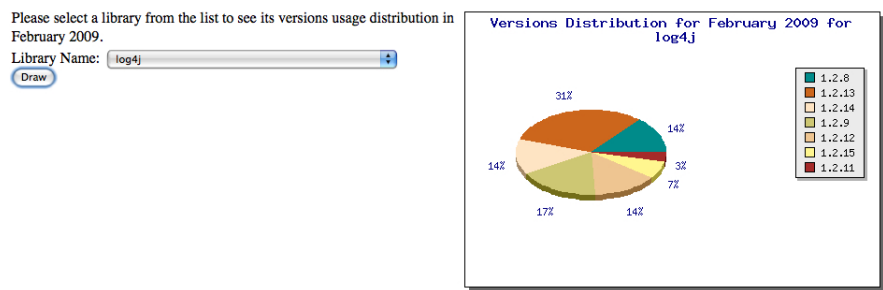


Figure 2.7: AKTARI—web-tool

factors, including the age of the projects. However, adding information about how many times a library was switched back from will give a better recommendation, since it also considers if a library is bug-free and thus the developers of a new project will be able to choose the newest and most reliable version.

Combining usage trends with times users switched back from a library gives an indication to the library developers what design mistakes they made and also how big the impact of a bug in their library is.

We have combined and integrated our library versions analysis techniques into both an Eclipse plug-in (see Figure 2.6) and a web-tool (see Figures 2.7 and 2.8) versions of AKTARI. The plug-in can assist library users in selecting the most recommendable library version, according to the majority of users. It can detect which versions are being used by the project and gives information regarding the global usage of these versions.

The web-tool is available for the library developers who want to check the usage trend of their library. It offers diagrams (like the ones in Figure 2.4) as well as pie charts that represent each of the three analysis techniques described, and thus assists developers in analyzing usage, success and popularity of their library. The tool is



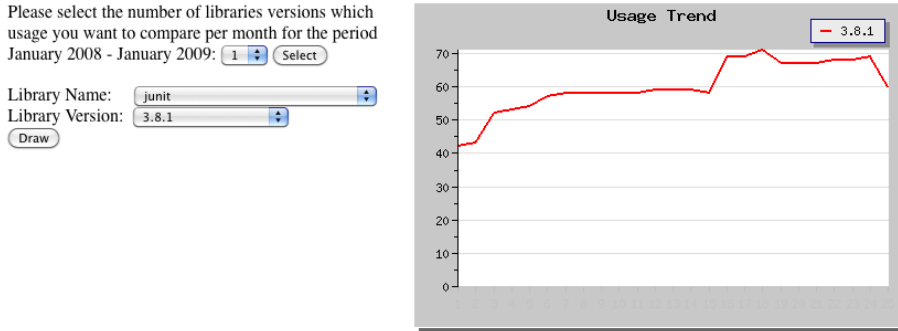


Figure 2.8: AKTARI web-tool.

available at <http://www.st.cs.uni-saarland.de/softevo/aktari.php>.

## 2.6 Related Work

Analyzing popularity and success factors of software projects is a relatively new research field. PopCon [12] is a prototype tool that collects popularity information regarding the Eclipse API. Based on the usage frequency of the API elements, the authors direct the library users to using the popular elements. For this research, the authors used data from a fixed point in time and investigated the usage of the Eclipse API. In comparison, we used data that spans a period of one year and goes through the code history of 200 projects. Thus our approach presents a much more general approach to estimating API popularity.

Schuler and Zimmermann [36] collect information about the popularity of library methods and thus help the library developers plan and prioritize the development effort. They also present the notion of usage expertise, which manifests itself whenever developers call an API method.

The AKTARI tool [25] collects information regarding the popularity of different library *versions* of the same library and provides means for evaluating the quality of those versions. The data used for this analysis is limited to open-source projects that use the Maven<sup>13</sup> platform.

A lot of related work has been also done to support developers in adjusting their code to a new version of an API. SpotWeb [38] is a tool that crawls open source repositories to mine frequent usage patterns for libraries. These patterns are then presented to

<sup>13</sup><http://maven.apache.org/>

the developers who want to start using a library. In contrast to this work, our approach tries to suggest whether a developer should at all use the specific library API and its elements.

Perkins [32] presents an approach to refactoring deprecated API methods. The author directly offers the API users a substitute code for the deprecated methods, by replacing those methods with their bodies and appropriate replacement code.

The MAPO [43] tool helps developers understand API usages better and write API client code more effectively. Given a query that describes an API element, this tool mines source code search engines results and presents a short list of frequent API usages.

To the best of our knowledge, our approach is the first that tries to recommend or dissuade from using a specific API element based on its global usage history as inferred from a large body of projects.

A lot of related work has also been done to support developers in adjusting their code to a new version of a library.

SpotWeb [38] is a tool that crawls open source repositories to mine frequent usage patterns for libraries. These patterns are then presented to a developer that wants to start using a library. In contrast to this work, our approach tries to suggest when a developer should switch to a new version of a library.

Another tool that aims at making the process of switching versions easier is the CatchUp! [11] tool. It is a plugin for Eclipse that records refactoring operations applied when switching to a new library. Recorded refactorings can then be replayed for clients that also want to switch to that version.

Dagenais and Robillard [1] use a partial program analysis technique to suggest replacements for calls to methods that are no longer present in the new version of a library. The presented tool gathers suggestions by mining the version history of the library and is based on the assumption that changes to replace a deleted method happen in the same change set.

Holmes and Walker [13] analyze library's API popularity and based on the usage frequency of the API elements direct the library users to using the popular ones.

To the best of our knowledge, our approach is the first that tries to recommend or dissuade from switching library *versions* based on global usage history.

## 2.7 Summary

In this chapter we investigated the question of API popularity. Our conclusions are based on data collected from hundreds of open-source projects and thus leverage the wisdom of a crowd of experts.

We have developed a prototype tool, called AKTARI, that analyzes the collected information and plots API elements usage trends. This information can be of great value for the API producers, when identifying the weak spots of their product. Based on the past usage trends, we are also able to give usage recommendations to the API users. The large and diverse set of projects that we analyze, as well as our high precision evaluation results, ensure that the recommendations offered by AKTARI are valid. As we have seen in Section 2.3.4, neglecting the vote of the majority can lead to introducing defects and code smells in the project code.

In conclusion we can state that:

*API elements usage trends are a method for displaying the preferences of the API users in the past and for predicting their future.*



## Chapter 3

# Evolution of Object Usage

*“According to the concept of transformational evolution, first clearly articulated by Lamarck, evolution consists of the gradual transformation of organisms from one condition of existence to another.”*

– Ernst Mayr<sup>1</sup>

As software evolves, so does the interaction between its components. In order for the software code to evolve consistently, all components need to be updated according to the evolution of the components they depend on. But how can we check if components are updated consistently? In order to track the evolution of the different software components, we focused on the evolution of the objects associated with a component. Preliminary results of the work presented in this chapter are published at ECOOP’11 [24].

### 3.1 The Problem

In software development, change is the only constant. New features are added, defects are fixed, or code is refactored to improve maintainability. In each of those cases, changes must be *consistently applied* to avoid defects and maintenance problems.

There are many reasons why changes are not propagated consistently throughout the code.

As an example of a project-wide change, consider the example Eclipse method `removeSelectionListener()` shown in Figure 3.1. In Eclipse 1.0, this method

---

<sup>1</sup>Ernst Mayr (1904 — 2005) was one of the 20th century’s leading evolutionary biologists.

```

void removeSelectionListener(Listener listener){
    if (!isValidThread ())
        error (SWT.ERROR_THREAD_INVALID_ACCESS);
    if (!isValidWidget ())
        error (SWT.ERROR_WIDGET_DISPOSED);
    if (listener == null)
        error (SWT.ERROR_NULL_ARGUMENT);
    if (eventTable == null) return;
    eventTable.unhook (SWT.Selection, listener);
    eventTable.unhook (SWT.DefaultSelection, listener);
}

void removeSelectionListener(Listener listener){
-->    checkWidget ();
    if (listener == null)
        error (SWT.ERROR_NULL_ARGUMENT);
    if (eventTable == null) return;
    eventTable.unhook (SWT.Selection, listener);
    eventTable.unhook (SWT.DefaultSelection, listener);
}

```

Figure 3.1: Method change from Eclipse 1.0 to 2.0. The old check has been replaced by a custom method call. Has this change been propagated consistently across Eclipse?

calls `isValidThread()` and `isValidWidget()` to verify that its preconditions are satisfied. In Eclipse 2.0, however, these two calls have been replaced with a call to `checkWidget()`—a new method which encompasses the two original checks and which can easily be extended to implement additional checks. This change has been applied across several Eclipse 2.0 methods that performed similar checks. But how do we know we found them all? And how do we ensure that new code actually uses `checkWidget()` rather than falling back to the old style?

In this chapter, we address these problems and introduce a tool that solves them, called LAMARCK<sup>2</sup>. LAMARCK analyzes the *changes* that occurred between two versions of the same project, determines the *object usage* in both versions and derives *evolution patterns*—that is, changes that have been consistently applied at multiple lo-

---

<sup>2</sup>Jean-Baptiste Lamarck (1744–1829) was an early proponent of organic evolution, proposing that organisms became transformed by their efforts to respond to the demands of their environment. He was, however, unable to explain a mechanism for this. [23, under “Lamarck” and “evolution”]

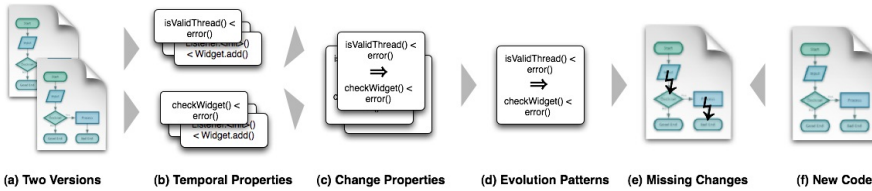


Figure 3.2: How LAMARCK works. Given two versions (a), LAMARCK extracts temporal properties (b) that characterize object usage in each version. The differences (c) are then mined for recurrent change patterns (d). These patterns can then be used to search for missing changes (e) in new code (f).

cations, like the one in Figure 3.1. These patterns can be forwarded to developers, informing them of object usage changes. As formal descriptions, however, they can also be used to detect *violations*—places in the code where a change should have been applied, but was not.

The remainder of this chapter follows the flow of information through our LAMARCK tool (Figure 3.2). In Section 3.2, we introduce the concept of *software evolution* from the perspective of evolving object usage. Section 3.3 introduces the concept of *evolution patterns* and shows how they and their violations (i.e., missing changes) can be detected. In Section 3.4, we present our evaluation on *detecting* and *preventing* errors. We conclude the chapter with a discussion on possible applications and conclusions.

## 3.2 Object Usage Evolution

As stated initially, **our goal is to have changes applied consistently across a piece of software**. To apply changes consistently, we need a notion of *similarity*—a similarity of changes, but also of contexts in which these changes are to be applied. Choosing an appropriate abstraction level for similarity is tricky. If we choose it too low, we end up with *syntactic* similarity, where changes are deployed consistently only across copy-and-paste clones. If we choose it too high, we end up with *semantic* similarity, which is generally undecidable.

To characterize changes, we use an abstraction that is well understood in software design—application programming interfaces (APIs). The key idea is to monitor how *the usage of object APIs evolves in individual versions*. This allows us to abstract away from syntactic similarity, yet detect inconsistent evolution in object usage: If a component still interacts with an API in a deprecated fashion, it is in need of an update.

Of course, there are established concepts to deter usage of “old” APIs: Individual functions may be marked as “deprecated”, causing a warning during compilation time, or simply not offered at all. However, “old” API usage may not necessarily mean using “old” functions, it may be a specific “old” combination of existing functions that is no longer up to date. In Figure 3.1, the methods `isValidThread()` and `isValidWidget()` still exist in Eclipse 2.0, and are still being used; it is this specific usage in this specific context, though, that now has evolved. Any characterization of object usage thus must express *relationships* between individual functions—to characterize both the evolving APIs as well as the context into which they are embedded.

In earlier work [39], Wasylkowski, Zeller, and Lindig described a formalism that satisfies these requirements—so-called *temporal properties* of object usage. To express the fact that method `a()` may be used before method `b()`, they use the syntax `a() < b()`. The temporal properties for the Eclipse 1.0 version of the code shown in Figure 3.1, for instance, include `isValidThread() < isValidWidget()` and `isValidThread() < error()`.

However, temporal properties can also carry *dataflow* information, denoting objects that are *shared* across these properties. The term “return value of `a()` < second argument of `b()`” means that the return value of `a()` is used as the second argument of `b()`. In the Eclipse 1.0 version in Figure 3.1, the properties thus actually read “target of `isValidThread()` < target of `isValidWidget()`” and likewise, because the two methods share the same target object (the implicit `this` object).

If the API usage changes, the temporal properties will also change. Since temporal properties encode dataflow, such changes will also characterize changes in argument ordering, or changes in the order of method calls. In Figure 3.1 for instance, the properties

```
target of isValidWidget() < target of error()
target of isValidThread() < target of error()
target of isValidThread() < target of isValidWidget()
target of error() < target of isValidWidget()
target of error() < target of error()
```

will be replaced by

```
target of checkWidget() < target of error()
```

Note that this change in temporal properties encodes both the change itself (from `isValidWidget()` and `isValidThread()` to `checkWidget()`) as well as the context (the `error()` method). Being able to express temporal ordering *as well as* data flow, and to include changes *as well as* context, is what makes this particular representation so well-suited to propagate changes at high precision.



### 3.2.1 Temporal Properties

We define software evolution as the evolution of temporal properties. But first, let us give details on how to extract temporal properties from a single project version.

**Definition 8 (Temporal property)** *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  $b$ . An event associated with an object is one of the following:

- a method call (including constructor calls) with the object being used as the *target* or as an *argument*: `x.bar(y, z)` is an event associated with  $x$ ,  $y$ , and  $z$ .
- a method call with the object being the value that was *returned* by the method: `x = map.items()` is an event associated with  $x$ .
- a field access with the object being the value that was *read*: `x = System.out` is an event associated with  $x$ .
- a cast with the object being *cast* to a different type: `(String) x` is an event associated with  $x$ .

Events are represented as precisely as possible (e.g., a method call is represented using the fully qualified name of the class defining the method, the method’s name and its signature). In the presented examples we omit most of those details to improve readability. We chose to focus on those types of events, as we found that these event types are well-suited to characterize the complex patterns of API usage in terms of their data flow and control flow [39]. The aim is to find a balance between the comprehensive results by static analysis and scalability.

To extract temporal properties we have adapted our earlier tool, JADET [39]. JADET works on bytecode level and extracts temporal properties in two steps:

**Mining object usage models.** Object usage models are finite state automata that show how objects “flow” through various events in a method (Figure 3.3). Object usage models are created by performing intraprocedural dat flow analysis on the program’s methods.

**Extracting temporal properties.** Temporal properties, as extracted from object usage models, provide a succinct and easy-to-manipulate representation of how objects actually “flow” through various events.

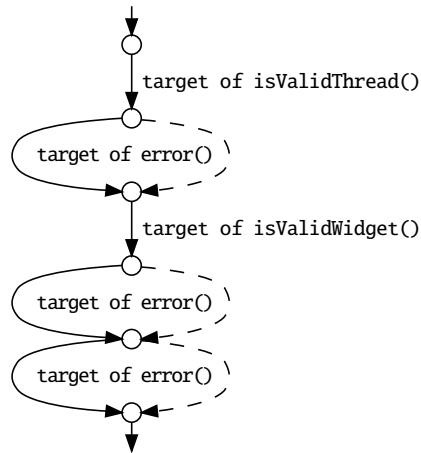


Figure 3.3: Object usage model for the target object (“this”) accessed by the methods shown in Figure 3.1 (Eclipse 1.0 version). Dashed lines denote empty transitions (without calls).

To illustrate how JADET works, let us once again consider the example method `removeSelectionListener()` from Eclipse version 1.0 in Figure 3.1. The first step is to mine object usage models. We create an object usage model for each statically identifiable object used by the method. These objects are: *formal parameters* of methods (including the implicit `this` parameter), *objects created* via `new`, *return values* of method calls (as in `x = map.items()`), values read from *fields* (including static fields, as in `x = System.out`), and explicit *constants* (such as `null` and `"OK"`). The `removeSelectionListener()` method uses three objects: the `listener` argument, the `eventTable` field, and the implicit `this` object. For each of those objects we build an object usage model that will show how the object is being used (i.e., in which events it participates). For example, the object usage model for the implicit `this` object is shown in Figure 3.3. Dashed edges represent “no-op” transitions. This model expresses the fact that the calls to `error()` are optional, whereas the two calls to `isValidThread()` and `isValidWidget()` always happen, and always in the same order.<sup>3</sup>

<sup>3</sup>The reader will notice that the object usage model is not fully correct, because it assumes that after the call to `error()` the method’s execution proceeds further. This limitation is not too important, though, as such “bail-out” methods do not occur too often and thus their influence on the overall results is small.

After we have extracted object usage models, we can abstract each of them into a set of temporal properties. The idea here is as follows: If there is a path through the object usage model (from the initial state to any other state), along which events  $a$  and  $b$  occur, and there exists an occurrence of  $a$  that happens earlier on that path than some occurrence of  $b$ , we create a temporal property  $a \prec b$ —expressing the fact that  $a$  may precede  $b$ . For this purpose we again use JADET, but with the following major modification. JADET, when abstracting object usage models into temporal properties, drops the dataflow information that is present in the models. For example, if an object usage model contains two successive events, “target of `lock()`” and “target of `unlock()`”, JADET will abstract it into a temporal property `lock()  $\prec$  unlock()`. For this work, we extended JADET to actually put *dataflow information* in the temporal properties. For example, extracting temporal properties from the object usage model shown in Figure 3.3 results in the following set:

```
target of Widget.isValidThread()  $\prec$  target of Widget.error()
target of Widget.isValidThread()  $\prec$  target of
Widget.isValidWidget()
target of Widget.error()  $\prec$  target of Widget.isValidWidget()
target of Widget.error()  $\prec$  target of Widget.error()
target of Widget.isValidWidget()  $\prec$  target of Widget.error()
```

Once we do this for every single object usage model extracted from a method, we can create a union of those models’ temporal properties and store it as the set of temporal properties that characterize the method. If we consider the method from Figure 3.1 (Eclipse 1.0 version) `removeSelectionListener()`, the set of temporal properties that characterizes it will contain the temporal properties shown above and the following temporal properties, obtained from the object usage model created for the `listener` argument and the `eventTable` field (the other two objects accessed by `removeSelectionListener()`):

```
2nd arg of EventTable.unhook()  $\prec$  2nd arg of EventTable.unhook()
field Widget.eventTable  $\prec$  target of EventTable.unhook()
```

Further details on extracting temporal properties can be found in the paper by Wasylkowski et al. [39].

### 3.2.2 Change Properties

Temporal properties tell us how a specific method uses APIs. Therefore, if we want to see how the APIs usages in a method evolved and changed between two versions of a

project, we can look at how the temporal properties of the method changed. For this purpose, we introduce the notion of *change properties*:

**Definition 9 (Change property)** *A change property is constructed from the comparison of two sets of temporal properties coming from two versions of the same method.*

To identify the same method between two versions of the project, we use the method’s name, signature, and the class in which it is defined (if a method was renamed, we will not be able to track its evolution). A change property is a temporal property annotated with information about the temporal property’s evolution between the two versions (more on that below). We built LAMARCK for the purpose of extracting change properties. LAMARCK extracts change properties in three stages, which we detail using the example from Figure 3.1.

### Extracting Temporal Properties

In the first stage, LAMARCK identifies the common methods between the two versions of the analyzed project and extracts the sets of *temporal properties* for each method in each version separately.

As an example, consider versions 1.0 and 2.0 of Eclipse. One of the methods that occurs in both versions is `removeSelectionListener()` from the `Button` class, shown in Figure 3.1. After identifying all common methods between the two versions, LAMARCK extracts temporal properties for each method in each version with the help of the modified JADET tool (as explained in Section 3.2.1). Here is the set of temporal properties for the `removeSelectionListener()` method, as implemented in Eclipse 1.0<sup>4</sup>:

```
EventTable.unhook() < EventTable.unhook()
field Widget.eventTable < EventTable.unhook()
Widget.error() < Widget.error()
Widget.error() < Widget.isValidWidget()
Widget.isValidThread() < Widget.error()
Widget.isValidThread() < Widget.isValidWidget()
Widget.isValidWidget() < Widget.error()
```

The temporal properties in version 2.0 are much simpler:

```
EventTable.unhook() < EventTable.unhook()
field Widget.eventTable < EventTable.unhook()
Widget.checkWidget() < Widget.error()
```

---

<sup>4</sup>From now on, we will omit presenting dataflow information in temporal properties, in order to increase the readability of the properties, unless this is needed to understand the property.

### Extracting Change Properties

In the second stage, LAMARCK compares the sets of temporal properties in both versions of each method and creates a set of *change properties* for this method—that is, temporal properties annotated with information about their evolution. If a temporal property is only present in the first version, LAMARCK transforms it into a change property annotated with *D* (for *deleted*); if a temporal property is only present in the second version, LAMARCK transforms it into a change property annotated with *A* (for *added*). In our `removeSelectionListener()` example, LAMARCK will create the following set of change properties:

```
D: Widget.error() < Widget.error()
D: Widget.error() < Widget.isValidWidget()
D: Widget.isValidThread() < Widget.error()
D: Widget.isValidThread() < Widget.isValidWidget()
D: Widget.isValidWidget() < Widget.error()
A: Widget.checkWidget() < Widget.error()
```

These change properties show how the Eclipse `removeSelectionListener()` method evolved from the point of view of its temporal properties between versions 1.0 and 2.0.

### Adding Context

In the final stage, LAMARCK extends the set of change properties created for each method in the previous stage with special change properties expressing the *context* of the change. For this purpose, LAMARCK adds to the set of change properties created in the previous stage another set of change properties, obtained by annotating *all* temporal properties from the earlier version of the method with *O* (for *original*)—these properties we consider the context of a change. Whereas the change properties created in the second stage represent the change itself and thus will allow us to find *evolution patterns*, the change properties created in this stage represent the *context* of the change and will allow us to find locations in the project where the change should have happened, but did not. This will allow us to find *missing changes*—the main contribution of this work.

In our example, Table 3.1 shows the final set of change properties extracted by LAMARCK for the example method from Figure 3.1:

In a similar manner, LAMARCK goes through all the methods common to two versions of the analyzed project and produces a set of change properties for each such method.

Table 3.1: Change properties for the `removeSelectionListener()` method. The properties of the context are also present.

```
O: EventTable.unhook() < EventTable.unhook()
O: field Widget.eventTable < EventTable.unhook()
O: Widget.error() < Widget.error()
O: Widget.error() < Widget.isValidWidget()
O: Widget.isValidThread() < Widget.error()
O: Widget.isValidThread() < Widget.isValidWidget()
O: Widget.isValidWidget() < Widget.error()
D: Widget.error() < Widget.error()
D: Widget.error() < Widget.isValidWidget()
D: Widget.isValidThread() < Widget.error()
D: Widget.isValidThread() < Widget.isValidWidget()
D: Widget.isValidWidget() < Widget.error()
A: Widget.checkWidget() < Widget.error()
```

### 3.3 Mining Patterns

Having generated the change properties, LAMARCK can use them to mine evolution patterns (i.e., sets of change properties that repeat in many methods) and find missing changes (i.e., methods where a certain change should have been applied but was not).

#### 3.3.1 Detecting Evolution Patterns

In Section 3.2, we have shown how LAMARCK can express evolution of methods using their change properties. Because we are interested in tracking the evolution of the whole project, we need to *aggregate* the individual changes over the entire project. More specifically, if a certain set of change properties occurs frequently throughout the project's evolution (i.e., is common to many methods), we treat it as an *evolution pattern*. For detecting evolution patterns LAMARCK uses *formal concept analysis* and its implementation provided by the Colibri/Java tool [9].

**Definition 10 (Formal Concept Analysis)** *Formal concept analysis is a way of automatically deriving an ontology, i.e. a set of concepts, from a collection of objects and their properties.*

Formal concept analysis is, broadly speaking, a technique for finding patterns [7]. Its input is a set of objects, a set of properties, and a cross table associating objects with properties. In our case, the set of objects is the set of common methods between the two

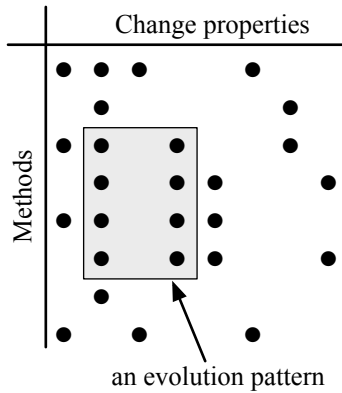


Figure 3.4: Concept Analysis Matrix

analyzed versions. The set of properties is the set of all change properties created from the entire project. Figure 3.4 shows an example of such a cross table. A dot is present in a place where a row and a column cross when the change property represented by the column was extracted for the method represented by the row (i.e. this property exists in this method).

The result of formal concept analysis are *concepts* found in the cross table. Generally speaking, a concept is a set of objects and a set of properties, such that every object in the concept is associated with all properties in the concept, and both sets are maximal (i.e., it is not possible to add elements to one of the sets without influencing the other one). In our case, a concept is a set of change properties and a set of methods such that the change properties occur in each method. To illustrate it on Figure 3.4—a concept is represented as a rectangle<sup>5</sup> in the matrix. The *size* of the concept on Figure 3.4 is 2, as it consists of 2 change properties. The number of methods in the concept is called the *support* of the concept. In the case of Figure 3.4, the support of the depicted concept is 4, because there are 4 methods that exhibit this group of properties.

If we restrict ourselves to finding concepts that have high support values, we will in effect be finding sets of change properties that occur in many methods—and therefore likely candidates for *evolution patterns*. For this purpose we use a parameter called the *minimum support*. LAMARCK returns only those concepts that have support values at least equal to the value of the minimum support parameter, and treats all returned

<sup>5</sup>Please note that not every rectangle needs to be contiguous—it is enough if there is a transposition of columns and rows in the cross table such that the rectangle becomes contiguous.

concepts as evolution patterns.

Table 3.2 shows an example of an evolution pattern as extracted by LAMARCK from Eclipse versions 1.0 and 2.0. This evolution pattern expresses the change to the `removeSelectionListener()` method shown in Figure 3.1. As it was applied to 169 other methods in Eclipse, the support of this evolution pattern is 170.

Table 3.2: An evolution pattern occurring in 170 Eclipse methods.

```
O:Widget.error() < Widget.error()
O:Widget.error() < Widget.isValidWidget()
O:Widget.isValidThread() < Widget.error()
O:Widget.isValidThread() < Widget.isValidWidget()
O:Widget.isValidWidget() < Widget.error()
D:Widget.error() < Widget.error()
D:Widget.error() < Widget.isValidWidget()
D:Widget.isValidThread() < Widget.error()
D:Widget.isValidThread() < Widget.isValidWidget()
D:Widget.isValidWidget() < Widget.error()
A:Widget.checkWidget() < Widget.error()
```

### 3.3.2 Finding Missing Changes

Finding evolution patterns is useful for understanding and for documentation purposes, and in Section 3.4.1 we give examples of interesting and useful evolution patterns found by LAMARCK. However, there is a very important question that can often occur while changing a project: *Was the change applied consistently throughout the entire project's code?* If we consider the evolution pattern shown in the preceding section, methods where the change expressed by the pattern was not applied, but should have been, become locations with potential future defects and/or maintenance problems. Therefore, it is important to be able to answer the stated question.

Our evolution patterns contain, amongst others, change properties annotated with the letter “O”—these are temporal properties that were present in the earlier version. They form the *context* of the change, and answer the question: *In which context does the change happen?*

As an example of such a change context, consider the evolution pattern shown in the preceding section, Table 3.2. Any method that has in its change property set all the change properties annotated with “O” from this pattern, but none of the other “A” or “D” change properties, is in fact a method that exhibited the same “starting



conditions”, like other methods that evolved, but it itself did not evolve. Potentially, there is a missing change in that method.

LAMARCK finds such missing changes again using the Colibri/Java tool [9] for formal concept analysis. Generally, whenever we have two concepts, and the set of properties in one of them is a subset of the set of properties in the other one, we have a potential violation. The reason why one set is a subset of another is that there are some properties missing—in our case, missing change properties. For details on the technique, see [21].

However, it can happen that an evolution pattern occurs in many methods, but is also missing in many methods. As it is not possible to say with absolute certainty that all those other methods are violating the pattern, we introduce another parameter—the so-called *minimum confidence*. The *confidence* of a violation is a number between 0 and 1, calculated as the ratio  $s/(s+v)$ , where  $s$  is the support of the evolution pattern (i.e., the number of methods that did evolve exactly according to the pattern), and  $v$  is the number of methods that violate the pattern (i.e., those that exhibit only the “O”-annotated change properties from the pattern).

## 3.4 Evaluation

LAMARCK detects evolution patterns that carry information regarding which sequences of method calls have been removed or added in a certain method between two versions of the project. In this section, we are going to evaluate LAMARCK’s usefulness.

We evaluate LAMARCK in two different evaluation settings:

**Detecting errors** In Section 3.4.1, we run LAMARCK on the subjects “as is” and did a manual evaluation of the reported violations for whether they are real code issues or false positives. It turns out that 33%–62% of the reported violations indeed are code smells or defects. In this scenario LAMARCK looks for missing changes, i.e. locations that were omitted by the developers when applying a change.

**Preventing errors** In Section 3.4.2, we simulate settings in which programmers had to apply a change and we wanted to see if LAMARCK can assist them during this process and prevent them from omitting a change. It turns out that in such a setting, LAMARCK has almost no false alarms (the precision ranges from 90%–100%); almost all of the inconsistencies detected by LAMARCK actually were in need for update and in later versions this update was performed in exactly the way as predicted by the pattern.

Table 3.3: Evaluation Subjects

Case study	# Methods		
	older version	newer version	Matching
Eclipse 1.0 vs. 2.0	34197	49862	19952 (58%)
Eclipse 2.0 vs. 2.1	49862	61358	44692 (90%)
AspectJ 1.6.0 vs. 1.6.3	38544	36729	35546 (92%)
Azureus 4.1.0.0 vs. 4.4.0.0	38328	40506	32200 (84%)

Table 3.4: Evaluation Subjects

Case study	#Patterns	Time
Eclipse 1.0 vs. 2.0	133	13m34s
Eclipse 2.0 vs. 2.1	2019	10m57s
AspectJ 1.6.0 vs. 1.6.3	58	7m41s
Azureus 4.1.0.0 vs. 4.4.0.0	17	9m57s

For our experiments, we used the projects and versions<sup>6</sup> given in the first column in Table 3.3. In the second and third columns we give the number of methods in the earlier and the later versions of the analyzed project. The fourth column contains the total number of methods that we managed to match between the two project versions, expressed both as an absolute value and a percentage of the number of methods in the earlier version.

### 3.4.1 Detecting Errors

In the first part of our evaluation, we want to assess how well LAMARCK is able to detect defects due to missing changes. For this purpose, we applied it to the subjects described in Table 3.3; if a violation of a pattern is reported, this means that a location in the project's code has been found that does not comply with the change pattern extracted from the project's history. In other words, we have found a location where the developers were supposed to change something, but they did not and thus we found a missing change. All of the presented results in the evaluation section are computed

<sup>6</sup><http://archive.eclipse.org/eclipse/downloads/> · <http://www.eclipse.org/aspectj> · <http://sourceforge.net/projects/azureus/develop>

Table 3.5: Reported unique violations and their success rate.

Case study	Time	#Violations	%Issues
Eclipse 1.0 vs. 2.0	62s	6	33%
Eclipse 2.0 vs. 2.1	111s	8	62%
AspectJ 1.6.0 vs. 1.6.3	72s	7	57%
Azureus 4.1.0.0 vs. 4.4.0.0	86s	5	40%

for a minimum support value of 10 and minimum confidence value of 0.8.

***Hypothesis 1:** LAMARCK detects missing changes.*

### Patterns reported

We ran LAMARCK on the test subjects presented in Tables 3.3 and 3.4. In the last two columns of Table 3.4, we give the number of extracted evolution patterns and the total analysis time in minutes and seconds (including the time used to extract object usage models and temporal properties). The way to interpret the number of patterns detected is generally “the more, the better”, as the more patterns we detect the more errors we will be able to catch. The time results are for an Intel Core 2 Duo 2.57 GHz machine with 4GB of RAM, averaged over 10 runs.

The first thing that stands out is the big difference in the number of patterns detected. This is due to the size of the projects and the difference between the versions. Generally speaking, if the two versions are too far apart in time, the source code would have evolved too much for us to manage to match the methods. This becomes evident in the change from Eclipse 1.0 to Eclipse 2.0. Here, only 133 patterns are reported; as Eclipse has changed quite a lot for this initial period there were only a few common methods (see Table 3.3) that could be detected and only so much patterns reported. In contrast, consider the 2019 reported patterns for Eclipse 2.0 vs. 2.1; we attribute this much higher number to the closeness of the structure of Eclipse in this minor release increment.

One would also notice the difference in the patterns detected in Eclipse and in the Azureus and AspectJ projects. This is due to the fact that the latter two are smaller than Eclipse, leading to fewer changes in fewer locations. We chose a minimum support of 10 for all of our experiments, which means that a pattern should appear in at least 10 locations, which is not so often the case for smaller projects.

We present a few examples of interesting patterns later in this section.

### Issues detected

Using the detected evolution patterns, LAMARCK looks for possible violations of those patterns in order to detect missing changes in the project's code. A violation reported by LAMARCK need not be an issue. We manually investigated all of the reported violations and classified them into the three categories of code defects, code smells (i.e., potential defects) and false positives.

We define code defects, code smells and false positives as follows:

- **Defects.** Those are real defects in the source code that will lead the project to fail by crashing or producing erroneous results.
- **Code smells.** This category contains all reported violations that are not at present defects in the code, but have the potential to become such. In this category fall also all reported cases, which might be improved in terms of readability, maintainability or performance of the program.
- **False positives.** This category contains all reported violations, that are neither defects, nor code smells.

In Table 3.5 we report our findings. The first column of Table 3.5 lists again our test subjects. The second column gives the time needed in seconds for LAMARCK to detect the pattern violations for each of our case study subjects. The third column contains the number of unique pattern violations detected by LAMARCK. The success rate of the reported violations is presented in the last column of the table and takes into account both the reported defects and code smells (summarized as “issues”), as in both cases the source code needs to be corrected. Our highest true positive rate is 62%—that is, 62% of the locations where LAMARCK detected an issue were in need of correction. Even though a false positive rate of 38% does indicate that there is still room for improvement, our results show that every second of our reports points to a bug (and this is in production code, which should have far fewer defects). This result highlights the potential benefits of our approach.

Two of our subjects had much lower true positive rates:

- Between Eclipse 1.0 and 2.0, the true positive rate is 33%—that is, 2 out of 3 violations are false alarms. This is due to several refactorings between these two major releases and most of the reported false positive pattern violations were reported for methods that have evolved too much for the pattern to still hold. In other words, the distance is too large to learn from.

- Most of the false positives reported by LAMARCK for the Azureus project were due to the different compiler versions used for compiling the two versions. For our experiments we used the byte code provided by the project and as LAMARCK is working on byte code level, this resulted in reporting of violations that boil down to equal source code, but different byte code. One should note that these are not faulty recommendations as the byte code did indeed change, but this change just had no visible manifestation in the end source code.

Generally speaking, any static defect detection tool will suffer from false positives—in particular, if the properties checked against were learned from other code instances. The question is whether the issues could also be detected in another, possibly cheaper way. Tools like FindBugs<sup>7</sup>, for example, check for specific API misuses by implementing a specific analysis for each API. This would be cheaper (in terms of computational power) and more precise, yet less general and more expensive (in terms of human labor). For the kind of issues LAMARCK detects, there is yet no other alternative; and our results indicate a reasonable efficiency.

### Qualitative Analysis

Now let us take a look at a few examples of the patterns and their violations detected by LAMARCK.

Our first example is the one presented in Figure 3.5. As one can see from this source code example new method calls have been added to the method. This change is detected by LAMARCK by the following pattern:

```
O: Composite.<init>() <- Control.setLayout()
O: Composite.<init>() <- Control.setLayoutData()
A: retval of Control.getFont() <- 1st arg of Control.setFont()
```

This pattern tells us that when we are setting the layout and the layout data on a `Control` object, we should also set the font on the same object (the `Composite` class inherits from the `Control` class). It originates from a widely spread issue in Eclipse 2.0, where people were not setting the font of the `Control` object they were working with, which could lead to problems when trying to set a font in a control based on the font of its parent. In Eclipse 2.1 this issue was corrected by adding calls to `getFont()` and `setFont()` in the appropriate locations, resulting in LAMARCK detecting it as a pattern. The support value for this pattern is 66, i.e. it was applied 66 times during the change from Eclipse 2.0 to 2.1. The violation of the pattern expresses itself in missing the following change property:

---

<sup>7</sup><http://findbugs.sourceforge.net/>

```

void createControl(Composite parent) {
    Composite comp = new Composite();
    ...
    Composite locationComp = new Composite();
    GridLayout locationLayout = new GridLayout();
    ...
    locationComp.setLayout(locationLayout);
    GridData gd = new GridData(GridData.FILL_BOTH);
    locationComp.setLayoutData(gd);
    ...
}

void createControl(Composite parent) {
--> Font font = parent.getFont();
    Composite comp = new Composite();
    ...
    Composite locationComp = new Composite();
    GridLayout locationLayout = new GridLayout();
    ...
    locationComp.setLayout(locationLayout);
    GridData gd = new GridData(GridData.FILL_BOTH);
    locationComp.setLayoutData(gd);
--> locationComp.setFont(font);
    ...
}

```

Figure 3.5: Method change from Eclipse 2.0 to 2.1. The added methods after the change address a font inconsistency.

A: retval of `Control.getFont()` < 1st arg of `Control.setFont()`

what this tells us is that there was supposed to be added a call to `getFont()` before the call to `setFont()`. LAMARCK was able to detect locations in the Eclipse 2.1 code, where this change was not applied, thus marking those locations as locations that violate an evolution pattern and expose a code defect in the project. What this means is that in the faulty methods the developer is creating a new component, but is not setting its font properly. As discussed earlier the omission of the `getFont()` method call was a widely spread issue in Eclipse 2.0. The violation found by LAMARCK shows that the Eclipse developers failed to locate all methods where this change needed to be

```

void navigation(IProgramElement node)
{
    if (node == null) return;
    ...
    treeViewBuilder.buildView
        (Asm.getDef().getHierarchy());
}
...
}

void navigation(IProgramElement node)
{
    if (node == null) return;
    ...
    treeViewBuilder.buildView
-->    (Ajde.getDef().getModel().getHierarchy());
}
...
}

```

Figure 3.6: Method change in AspectJ. After the change `buildView()` now takes an object derived from the `Ajde` singleton.

applied. In this case LAMARCK was able to point to such omitted locations.

Our second example comes from the AspectJ project and is shown in Figure 3.6. The change pattern that LAMARCK detected for this piece of code is the following:

```

O: retval of Asm.getDef() <- Asm.getHierarchy()
D: retval of Asm.getDef() <- Asm.getHierarchy()
A: retval of Ajde.getDef() <- Ajde.getModel()
A: retval of Ajde.getModel() <- Asm.getHierarchy()

```

What this pattern, with a support of 10, tells us is that the return value of the deleted `getDef()` method call was substituted with the return value of the added `getDef()` and `getModel()` method calls. As one can notice, the deleted methods are from class `Asm`, while the added ones are from class `Ajde`. After investigating the case, it turns out that the `Asm` class was a singleton class, but it was changed in the newer AspectJ version to a non-singleton class, due to change in the AspectJ functionality. A new class `Ajde` was created that acted as a singleton wrapper of the old class and had a `Asm` field (that is why the `Asm.getHierarchy()` is called on the `Ajde.getModel()`

```

public AddBookmarkAction(Shell shell)
{
    super(WorkbenchMessages.getString("AddBookmarkLabel"));
    setId(ID);
    Assert.isNotNull(shell);
    this.shell = shell;
    setToolTipText
        (WorkbenchMessages.getString("AddBookmarkToolTip"));
    WorkbenchHelp.setHelp(this,
        new Object[] {IHelpContextIds.ADD_BOOKMARK});
}

public AddBookmarkAction(Shell shell)
{
    super(WorkbenchMessages.getString("AddBookmarkLabel"));
    setId(ID);
    Assert.isNotNull(shell);
    this.shell = shell;
    setToolTipText
        (WorkbenchMessages.getString("AddBookmarkToolTip"));
    WorkbenchHelp.setHelp(this,
--> IHelpContextIds.ADD_BOOKMARK);
}

```

Figure 3.7: Method change from Eclipse 1.0 to 2.0. Before the change there was a call to a deprecated method.

return value). In the newer version of AspectJ both classes are present (and all their methods, as well). Thus, a developer could still use the `Asm.getDef()` method and the compiler would not issue a warning. However in this case, the `Ajde` class needs to be used.

Now let us take a look at an example of a code smell detected by LAMARCK. In Figure 3.7 one can see a frequently occurring change between Eclipse 1.0 and 2.0. This change has been detected by LAMARCK through the following pattern:

```

O: setToolTipText(String) < setHelp(iAction, Object[])
D: setToolTipText(String) < setHelp(iAction, Object[])
A: setToolTipText(String) < setHelp(iAction, String)

```

What this pattern tells us is that the call to `setHelp(iAction, Object[])`



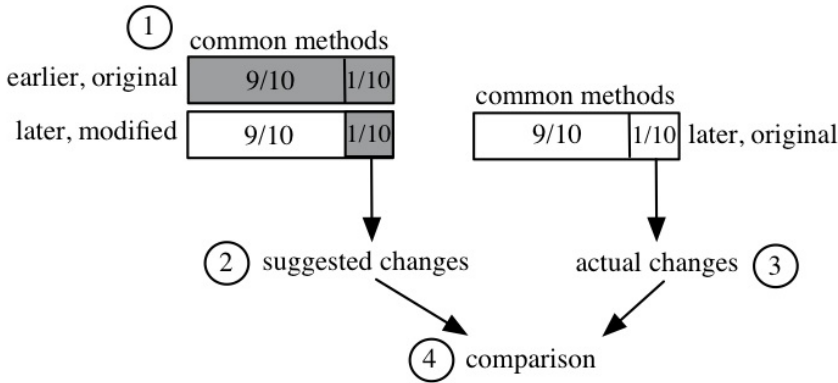


Figure 3.8: Evaluation Setting. In 1/10 of the code, we artificially revert changes and check whether LAMARCK is able to predict them after learning from the changes in the remaining 9/10.

has been deleted and substituted with a call to `setHelp(iAction, String)`. The support for this pattern is 33, which means that in 33 methods throughout Eclipse 2.0 this pattern has been followed. LAMARCK has detected violations of this pattern in Eclipse 2.0 in the sense that the developers continued to use the old `setHelp` method. After investigating the matter we found out that the old `setHelp` method was in fact a deprecated method in Eclipse 2.0 and shouldn't have been used. Thus, even though the program was not crashing, we classified this violation as a code smell, as the code is expressing unwanted and deprecated behavior.

In conclusion we can state that

*LAMARCK is able to find useful evolution patterns and use them to detect missing changes in project code.*

### 3.4.2 Preventing Errors

We already discussed a few of the patterns detected by LAMARCK and their usefulness when it comes to detecting missing changes in project's code. However, this is only one of the aspects of the usefulness of those patterns. The patterns as detected by LAMARCK can also be used for *preventing* errors. On top of that, the patterns themselves are an exact suggestion of how to fix a potential future defect location.

**Hypothesis 2:** *LAMARCK prevents missing changes and suggests how to fix them.*

Table 3.6: LAMARCK’s effectiveness in discovering inconsistently applied changes. The table summarizes the results obtained for 50 random splits of the input data. See Section 3.4 for details on the evaluation scheme.

Case study	# Inconsistencies			Precision		
	min	max	avg	min	max	avg
Eclipse 1.0 vs. 2.0	16	29	22	93%	100%	<b>98%</b>
Eclipse 2.0 vs. 2.1	14	24	19	90%	100%	<b>99%</b>
AspectJ 1.6.0 vs. 1.6.3	4	12	7	100%	100%	<b>100%</b>
Azureus 4.1.0.0 vs. 4.4.0.0	2	5	3	100%	100%	<b>100%</b>

### Evaluation Setting

In order to perform such evaluation in an unbiased manner, we designed the scenario sketched in Figure 3.8. The key idea is to *artificially revert changes* between versions and check whether LAMARCK is able to predict them. In this way we simulate a real-life scenario, when a given change pattern has been applied to only a few of the intended locations.

- ① We split the set of common methods in the two versions into two parts—9/10 and 1/10 parts. In the 1/10 part, we substituted the methods from the later version by the methods in the earlier version, effectively reversing the changes that occurred between those two versions. We thus simulated a situation in which 1/10 of the code in the later version would still be in need of update.
- ② We applied LAMARCK on the earlier version and the (modified) later version and had LAMARCK predict which locations in those 1/10 methods would be in need for update and what the update should accomplish.
- ③ From the (original) later version, we looked at the *actual changes* applied in the same 1/10 methods subset.
- ④ Comparing the *suggested* and the *actual changes* for the 1/10 part allows us to assess the accuracy (and hence the usefulness) of LAMARCK.

Using the setting described above, we performed 50 random 1/10 vs. 9/10 splits on each of the four case studies.

## Results

Our results are presented in Table 3.6. The first column gives the projects and their versions used for the experiment. The second column gives the number of inconsistencies (with a pattern) detected in the 1/10 part of the methods, when the later version was modified as explained above—we show the minimum, maximum and the average number of reported inconsistencies over all the 50 experimental runs. The third column contains the *precision* of the reported inconsistencies, i.e. the percentage of the cases where the faulty location was indeed fixed by the developers exactly as LAMARCK recommended it to be fixed (for a definition of precision, please refer to Chapter 2). A high precision implies a low number of false positives, i.e. invalid recommendations.

The reported precision in Table 3.6 is accumulated over the total number of 50 random splits. As one can see from the table, our precision results are close to 100% in all cases. This means that almost all of the inconsistencies detected by LAMARCK actually were in need for update, and this in the exact way as predicted by the pattern. We consider this precision a good result, as this means LAMARCK is not only able to predict that some location will change, but is able to accurately predict how this location will change. Note that these results do not depend on the size of the split (if we consider all the changes applied in 90% of the code or less), as the size of the split would influence the amount of patterns detected, but not their defect prevention properties.

## Validation

In our experience, evaluation results like these are more likely to indicate a bug rather than a feature. We therefore manually took a look at one random split for each of the four experiments in order to re-verify that the location for which an inconsistency would have been reported was indeed changed the way our patterns say. Our findings were that in all cases when LAMARCK predicted that some code needs to be changed, this code was indeed changed and in fact it was changed exactly as predicted by LAMARCK. We classified predictions that were wrong as false positives.

As LAMARCK extracts the evolution patterns from the bytecode of the project versions, we also stumbled across evaluation artifacts—cases where our tool recommends a change, but this change is only in the type of the objects returned by some methods and passed to some other methods. The situation here is as follows: class *A* gets replaced by class *B*, and methods that used class *A* now use class *B*. Source code that does not use objects of class *A* explicitly, but just passes them around (as in `foo(bar())`), if `bar()` returns an object of type *A* and `foo()` accepts an object of type *A* does not need to be changed, but the bytecode will change after the project gets recompiled.

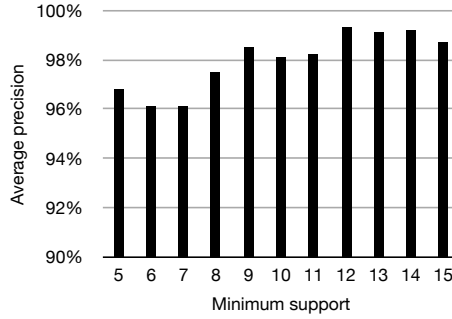


Figure 3.9: Influence of minimum support on LAMARCK’s effectiveness in the case of Eclipse 1.0 vs. 2.0 (minimum confidence fixed at 0.8)

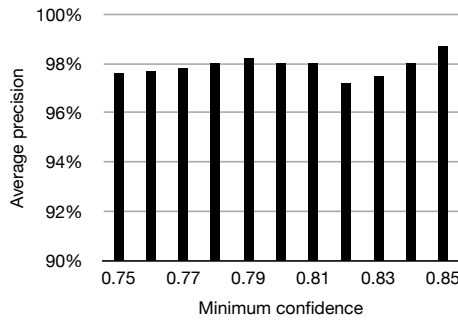


Figure 3.10: Influence of minimum confidence on LAMARCK’s effectiveness in the case of Eclipse 1.0 vs. 2.0 (minimum support fixed at 10)

Since in our evaluation we use old bytecode of the 1/10 part, LAMARCK reports such locations as needing an update. However, these are not faulty recommendations *per se*, because during real usage (and not in an artificial setting, as in our evaluation) after replacing classes and recompiling the code LAMARCK will not report such locations anymore.

After classifying all reported predictions into true positives, false positives and evaluation artifacts, our manual inspection reports precision ranging from 90% to 100%. We classified in total 45 predictions out of which 16 were evaluation artifacts, which we ignored. Thus, our manual inspection confirmed our high precision rate.

To further reduce the probability of a bug, we also took a look at the *recall* values and they were as close as up to 10%. Recall in this case would mean finding all

locations where a change is supposed to be applied (for a definition of recall, please refer to Chapter 2). Such low recall values are expected, as we do not aim at finding all changes that occurred between two versions of a given project; we aim at finding all *frequently* occurring changes—changes that follow certain patterns. The recall values are low, as most of the code changes occur only at a single location (or at least in less than 10 locations, which is our minimum support number). A bug in the evaluation, for instance a mix of training and testing sets, would have returned far higher recall values.

### Sensitivity Analysis

Finally, we investigated the sensitivity of our results to small changes of the minimum support and minimum confidence parameters. For this purpose, we redid our evaluation on Eclipse 1.0 vs. 2.0 for different minimum support and minimum confidence values. The results are shown in Figures 3.9 and 3.10. It turns out that LAMARCK is quite insensitive to small changes of its input parameters. What is more important, for minimum support of 5, LAMARCK finds on average 40 missing changes (compared to 22 when using the default minimum support value of 10), and still more than 95% of LAMARCK’s suggestions (on average) are followed by the Eclipse developers. Thus, users can tweak LAMARCK to find more missing changes, and still get very precise results.

All these results confirm our hypothesis:

*LAMARCK can detect missing updates with a precision close to 100%,  
giving precise fix suggestions.*

## 3.5 Applications

Let us now take a closer look at a few areas of application, LAMARCK can apply to.

### 3.5.1 Detecting Errors

As it is evident from Section 3.4.1, LAMARCK is able to detect missing changes in the source code. What this means to software developers and managers is that running LAMARCK against the source code of their software project will lead to probable detection of defect locations. From the times needed for running the tool, presented in Table 3.5, one can also see that running LAMARCK is highly time-efficient and can be used at any point of the project development process, without the need for a special time planning. We believe that LAMARCK can be of great use in the field of defect

detection and its usage can possibly lead to reduction of process costs usually related to testing, debugging and fixing of software defects.

### 3.5.2 Preventing Errors

As one could see from Section 3.4.2, LAMARCK proved to be extremely efficient at preventing errors from appearing in the source code. If the developers have the tool run in the background of their IDE<sup>8</sup>, it can greatly assist them in their coding practice. LAMARCK can not only be used to warn experienced developers against repeating mistakes of the past, but it can also be used as a valuable tool for newly joined developers, who are still unfamiliar with the project and its project-specific patterns. Even though we have not conducted extensive empirical studies regarding the ease of usage and usefulness in the above mentioned developer groups, we believe that LAMARCK will be a great asset to any software development project.

### 3.5.3 Threats to Validity

As any other, our study is prone to threats to validity.

**External validity.** We investigated seven versions of three different open-source projects of different maturity, size and domain. However it is possible that the results we acquire on them do not generalize to other arbitrary projects. For example, closed-source projects, due to differences in the internal processes, might have very different properties.

**Construct validity.** Our approach might be prone to mistakes. The external tools we use might also be defective. However, we hope that we have eliminated this threat to a big extent as the Colibri [21] and the JADET [39] tools are publicly available<sup>9</sup> and besides the validation in Section 3.4.2, we ourselves have performed a cross-check of our source code to eliminate any possible mistakes on our side.

**Internal validity.** The presented evaluation of the usefulness of LAMARCK when used as an errors preventing tool is a combination of automatic and manual inspection of 50 random splits of the common methods for two versions of a project's methods. It might be the case that 50 splits are not enough. It might also be the case

---

<sup>8</sup>IDE stands for integrated development environment and is a software application that provides facilities to computer programmers for software development.

<sup>9</sup>This is the online implementation of the latest JADET tool version: <http://www.checkmycode.org/>

that, as we are not well acquainted with the analyzed projects, our manual classification results would not be the same if classified by a real developer of that project. Another possibility is that, apart from the evaluation artifacts mentioned in Section 3.4.1, other missing changes reported by LAMARCK would also be found by a compiler (due to missing types, etc.). However, based on our evaluation in Section 3.4.2, we think that the likelihood of *all* reported missing changes being of this type is very small.

## 3.6 Related Work

Our approach is unique in that it combines specification mining with mining source code archives. LAMARCK is based on the JADET [39] static analysis tool, but the same technique could be used to enrich any other single-versioned programming rules mining tool like PR-Miner [20] or GrouMiner [31].

To the best of our knowledge, the presented work is the first to define API evolution patterns as a set of change properties derived from temporal properties. However, there are many other approaches that learn from existing code in order to learn about the software evolution or detect code defects.

### 3.6.1 Learning Evolution Rules

A large body of work has been done in the area of looking for API evolution changes and the way they should be deployed to API clients. Several techniques and tools [19, 4, 40, 42] have been developed to discover the refactorings that a software system has undergone by analyzing two versions of the evolved software project. Dig and Johnson [5] found out that 84%–97% of all API breaking changes, i.e. changes that are extremely disruptive in the development life cycle of component-based applications, are in fact refactorings (e.g. class or method renaming). LAMARCK is also able to detect change patterns based on refactorings, but is as well able to detect much more complicated patterns and thus find non-refactoring based defects.

Nguyen et al. [29] developed the LibSync tool, which helps developers migrate from one library version to another. LibSync has a knowledge base of API adaptation patterns for each library version and given a client system and the desired library version, the tool finds the locations in the code that are associated with the changed API version. In comparison, even though both tools report similar precision rates, LAMARCK is much more light-weight and time-efficient and is able to detect both external API, as well as project-specific change patterns.

Based on how a framework adapts to its own changes, Dagenais and Robillard [2] developed a recommendation system that suggests replacements for framework elements accessed by client programs. Dagenais and Robillard designed a tool, called *SemDiff*, that explores code locations that used an API method, which was later deleted from the API. *SemDiff* mines the new API methods that are being used instead and comes up with a set of method calls that substituted the call to the deleted API method. In comparison to *SemDiff*, LAMARCK is also able to perform such kind of detection, but in addition can also offer recommendations in the cases where an entire piece of code was substituted with a method call (e.g. our `checkWidget()` example) or the method (or even a class) remained unchanged, but the usage pattern of the method (or the class) changed (e.g. our *Ajde* example).

Fluri et al. [6] looked for context changes of method invocations as moving an existing method invocation into the `then` or the `else`-part of an `if`-statement. Our approach is not restricted to such context changes and can detect any type of context change thanks to the *original* change properties representing the context of a change.

Kim and Notkin [17] grouped code changes that form change patterns with the help of their *LSdiff* tool. *LSdiff* infers systematic structural code differences as logic rules from the difference of two sets of predicates, representing the two versions of a program. *LSdiff* was built as a tool for assisting developers when making a `diff` between two revisions of a file.

### 3.6.2 Learning from Project History

*FixWizard* [30] is a tool that identifies recurring bug fixes by comparing the changes that happened between two version control revisions of a project. Apart from using a completely different algorithm for identifying frequently occurring changes, LAMARCK and *FixWizard* also interpret the context of a code change differently. *FixWizard* is restricted to infer and offer recommendations only from and to code peers that match in naming convention or ancestor classes. LAMARCK on the other hand interprets context as any method that meets the same “starting conditions” (described by the original temporal properties) of a pattern, thus addressing a much larger number of methods.

Livshits and Zimmermann [22] also mine patterns and their violations from software repositories. Their *DynaMine* tool can detect a pattern of method calls, but only if the method calls are used in the same transaction. We on the other hand, look at the project as a whole and extract our patterns based on the entire project’s code. This allows us to produce more general patterns and to find patterns and possible defect locations different from the ones *DynaMine* detects.

Kim et al. [18], similarly to Livshits and Zimmermann [22], also operate on version system transaction level and look for patterns on bug fixes. Our approach can, like



BugMem, detect changes that were due to a bug fix, but is not restricted to that. For example, we are able to detect a change due to the addition of new code, which is not the case with BugMem. We also have a much higher true positive rate, when comparing our best defect detection results (62%) against their best results (38.7%) on the Eclipse project (see Section 3.4.1).

Williams and Hollingsworth [41] mine source code repositories to look for commonly fixed bugs. We do not rely on version repositories, which are not always easy to find, but simply on two versions of the same project. Our method is not restricted to patterns of bug fixes.

## 3.7 Summary

To reduce the risk induced by software evolution, it is necessary that changes be applied *consistently* across a project. By characterizing the impact of change on involved method calls, their temporal ordering, and their dataflow, our tool LAMARCK learns how software has changed in the past. As it comes to *preventing* errors, LAMARCK's recommendations are very precise. An average false positive rate of  $< 2\%$  implies substantial benefits at low costs and low requirements. We therefore recommend usage of LAMARCK or similar approaches in all projects that care about minimizing the risks of inconsistent software evolution. On top of that, LAMARCK can also be helpful for *detecting* errors in existing code, uncovering complex API usage changes with a true positive rate of 33%–62%.

The contributions of our work are as follows:

- The first approach to study how object usage changes over time;
- The first approach to combine specification mining with mining source code archives;
- A novel approach to detect missing and incomplete changes, based on object usage;
- A novel approach to detect bugs due to inconsistent API usage, based on API evolution.



# Chapter 4

## Assessing Modularity

*“Every program is a part of some other program and rarely fits.”*  
– Alan J. Perlis, Epigrams on Programming.<sup>1</sup>

Good program design strives toward modularity of its components, that is, limiting the effects of changes to the rest of the code. We assess the modularity of software modules by mining change histories: *the more a change to a module implementation affects its usage in the client code, the lower its modularity* and vice versa—the less a change in the modules’ implementation affects its client code, the higher its modularity. Preliminary results of the work presented in this chapter are published at PASTE’11 [27].

### 4.1 The Problem

A software project is in a constant change mode—features are being added, defects are being fixed, or code is being refactored. Modular software design attempts to limit the effect of those 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 work we define *modularity* as

---

<sup>1</sup>“Epigrams on Programming” is an article by Alan Perlis published in 1982, for ACM’s SIGPLAN journal. They are a series of short, programming language neutral, humorous statements about computers and programming.

Table 4.1: Evaluation Subjects.

Case study	# Modules		median loc
	all	changed	
Eclipse 3.4.2	17350	59	30
Eclipse 3.5.2	18531	-	-
AspectJ 1.6.2	1940	17	35
AspectJ 1.6.3	1952	-	-

**Definition 11 (Modularity)** *A module has high modularity if the changes to the module’s implementation do not lead to changes in its usage. A module has a low modularity if the changes to the module’s implementation lead to changes in its usage.*

In this study, we investigate the extent to which a change to a module implementation requires changing client code—that is, code that uses elements of the module.

As an example, consider the Eclipse class `CompletionProposal` on Figure 4.1. 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 is; the lower the modularity, the higher the likelihood of future changes to induce unwanted effects.

The approach presented in this chapter is concerned with the influence of change in a module’s implementation on the module’s client code. As a *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. 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. Details on how we construct those patterns can be found in the following sections.

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

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

Figure 4.1: Change in the *usage* of the `CompletionProposal` class that occurred in the Eclipse code between versions 3.4.2 and 3.5.2.

## 4.2 Collecting Usage and Implementation Changes

To assess modularity we analyze two variables—the number of module’s implementation changes done 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.

### 4.2.1 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 both empty and comment lines. Any other line that was changed we count as changed.<sup>2</sup>

Table 4.1 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.

### 4.2.2 Collecting Usage Changes

In order for us to detect how a specific module and its elements are being used, we use our tool LAMARCK [24] to collect and extract the module’s usage information.

LAMARCK receives as an input the binary code of two versions of the project we want to analyze and looks for the evolution patterns that occurred between those two versions (for a definition of an evolution pattern, please refer to Section 1.2).

LAMARCK goes through four basic stages, illustrated on Figure 4.2, in order to produce those evolution patterns (for more detailed explanations, please refer to Chapter 3).

---

<sup>2</sup>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.

**Mining object usage models.** For each statically identifiable object used in each of the project’s methods, an object usage model is created. Here we refer to an *object* as defined in object-oriented programming, i.e. a particular instance of a class. Previous research has demonstrated the potency of object usage models in expressing the behavior of an object [39, 24]. An example of an object usage model can be seen on Figure 3.3.

**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  $b$ . Examples of an event are calling a method on an object, type casting or returning a value [24] (more details on event types can be found in Section 3.2.1). In this second stage, LAMARCK extracts temporal properties per method per analyzed project versions (i.e. separately for each version).

**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, where “A:” stands for added, “D:” stands for deleted and “O:” stands for originally present (which allows us to track the context of a change). 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.

**Mining evolution patterns.** LAMARCK takes all the sets of annotated temporal properties and, with the help of concept analysis [7, 9], 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 4.3 shows the evolution pattern<sup>3</sup> that corresponds to the code change from Figure 4.1.

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 module’s methods). To collect the module’s usage changes we count the number of evolution patterns, of which the module is part of. One 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 modules clients, module’s code

---

<sup>3</sup>The evolution pattern has been slightly modified to fit the page format. LAMARCK works with the full signatures of the methods.

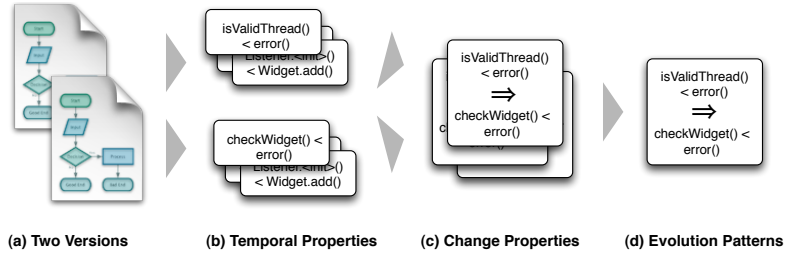


Figure 4.2: The four basic stages of LAMARCK.

```
O: CompletionProposal.setCompletion() <
  CompletionProposal.setRelevance()
D: CompletionProposal.setCompletion() <
  CompletionProposal.setRelevance()
A: InternalCompletionProposal.setCompletion() <
  InternalCompletionProposal.setRelevance()
```

Figure 4.3: The evolution pattern corresponding to the code change indicated on Figure 4.1. The pattern shows what was the context of the change (indicated by the “O” properties) and what was deleted and added (indicated by the “D” and “A” properties).

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.

## 4.3 Evaluation

We examined the test subjects presented in Table 4.1 to assess the modularity of their modules based on the existence of implementation and usage changes.

### 4.3.1 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. An illustration how this data looks like can be found on Figure 4.4(a). We also collected the number of evolution patterns each module was part of (an illustration of this data



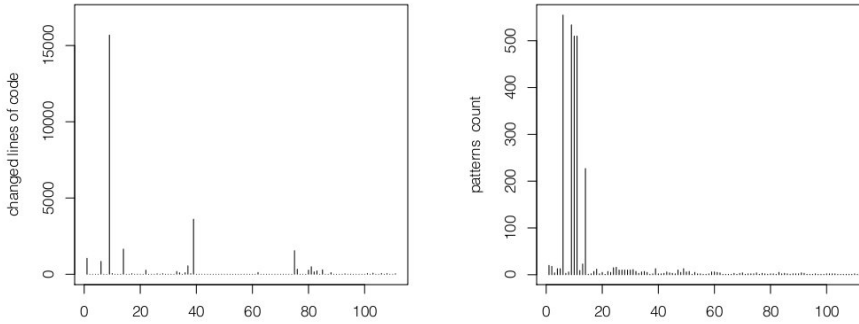


Figure 4.4: (a) the number of changed lines of code for each of the Eclipse modules in the transition from version 3.4.2 to version 3.5.2; (b) the number of evolution patterns each Eclipse module was part of in the transition from version 3.4.2 to version 3.5.2.

can be seen on Figure 4.4(b)).

After collecting both the number of changed lines of code and the number of evolution patterns per module, we examined the correlation between them. Figure 4.5 presents the correlation between the modules' implementation and usage changes in Eclipse for the transition from version 3.4.2 to version 3.5.2. Figure 4.6 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**.

### 4.3.2 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

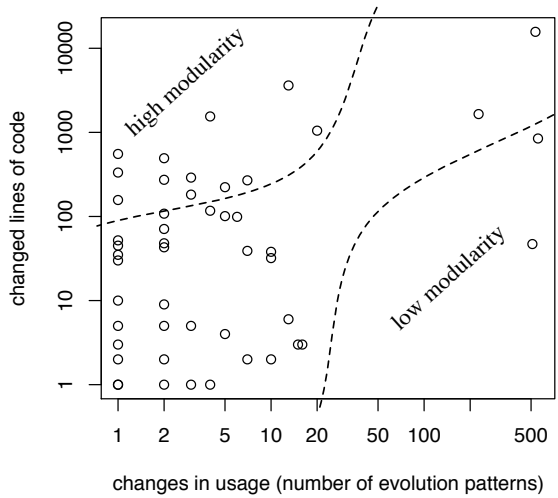


Figure 4.5: Correlation between implementation changes and usage changes for the case of Eclipse 3.4.2 to 3.5.2. Each point represents a module.

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 4.1). 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 Eclipse class `StyledText`. 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.

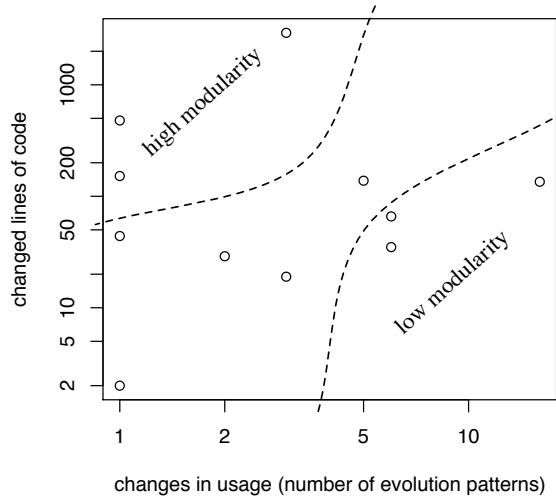


Figure 4.6: Correlation between implementation changes and usage changes (AspectJ 1.6.2 to 1.6.3). Each point represents a module.

## 4.4 Applications

The knowledge of which modules are more modular and which are less modular has multiple applications. Let us take a look at a few such applications, which by all means are not an extensive list. Our belief is that this technique can be used in multiple scenarios and can be further extended and adjusted to the needs of its user. Part of our future work includes analyzing the application of modularity in the optimization of the software development process.

### 4.4.1 API producers

An API, as any other piece of software, has to deal with problems like resource distribution, user complaints, and software processes. The approach we presented in this chapter can serve the API producers to locate the API modules with the lowest modularity and direct their efforts towards refactoring them. Refactoring a module with low modularity is important as it speaks for the quality of the API and can thus be a major

factor in the success or lack of such of the API.

The presented approach can be used by the API producers to draw a correlation between the internal code changes of the API modules and the changes in their users' code. We would like to point out though that our technique could just as well be used much earlier in the process in order to see how a change will affect the users. Such an observation, though important, can be avoided on an earlier stage and can thus prevent the low modularity to spill over to the users's side. The API producers can use the presented technique on their own code to see if their changes affected their own user modules within the API itself. Even though we have not performed the corresponding experiments, we believe such a step will greatly improve the modularity of the API modules on a much earlier stage and will ensure the API to provide a good product to its end users.

A usage, as the one described in the previous paragraph, can be applied to any piece of software. Using our approach, any existing software system can self-analyze the modularity of its modules by correlating it with its own changes in the other internal modules that are dependent on the analyzed module. Thus we can generalize the usage of the presented technique to any software system.

#### **4.4.2 API users**

As any user, an API user would like to be using the best product possible. The users always aim at using the best product available, e.g. API, that will fit their needs as perfect as possible. The users on an API are software developers, who are hoping that the API they chose will help them do their job better and faster.

Imagine a scenario where the user of an API has decided to put his/her trust into this API and has incorporated the usage of its modules into his/her own source code. The external API works perfectly and the developer is satisfied and moves on to further tasks. After some time the API producers decide to change the implementation of some of the API modules and deploy this change to their users by releasing a newer version of the API. API users are always urged to use the latest API version available, as the old versions are not supported anymore and the newer versions might bring new and useful extensions. In this scenario however, some of the API modules have been changed in such a way that the code of the API user becomes faulty and/or does not compile anymore. In this case the API user has to drop his/her current task, go back to the old task, try to remember what the code using the low modularity API module(s) was supposed to do and change it accordingly. As the reader can imagine, such a scenario is not too pleasant for the user and if it repeats often the user is very likely to decide to stop using this API and to move to another one.

The presented technique can serve the API users in selecting an API that has a good

design and has modules with high modularity. We believe that it is important to keep the API users informed of the quality of the APIs they are to use and thus to be able to make a better choice for their system.

### 4.4.3 Defect prediction techniques

In the literature exist many techniques that deal with trying to predict possible defect locations in the form of faulty classes, methods or software processes. The work that we introduced in this chapter can be used to expand and improve such existing techniques. One example would be adding the knowledge of modularity to existing algorithms for finding security vulnerabilities [46, 8].

There are also plenty of techniques developed to compute complexity of code changes [10] and the way that influences the system. If one adds a modularity metric to existing code complexity metrics [14, 28], we believe this could further improve the presented results in these research works.

## 4.5 Threats to Validity

As any other empirical study, our study is prone to threats to its validity.

**External validity.** We have investigated four versions of two different open-source projects, coming from different maturity, size and domain. However, we do not claim that our results generalize to other arbitrary projects.

**Approach is applicable only to JAVA projects.** As the tools we use to detect the modules' usage changes are designed for JAVA projects, our approach and tools are currently applicable only to such projects. This limitation however comes only from the tools we use and does not generalize to the the approach itself.

**The implementation may have errors.** A final source of threats is that our implementation could contain errors that affect the outcome. To control these threats we did a careful cross-check of the data and the results and to best of our knowledge it is free of errors.

## 4.6 Related Work

Martin Robillard [34] 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 [3] also recognized that a change in the implementation of an API might lead to problems in the client code. Their SemDiff tool recommends replacements for API methods that were deleted during the evolution of an API.

The approach developed by Hovemeyer and Pugh [15] 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 uses such a module would be a subject to potential code defects.

*Change impact analysis* [33] 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* [47] 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* [45] relate the likelihood of a component failure to the number of dependent clients. Again, we want our modularity measure to be independent of a particular context.

## 4.7 Summary

In this chapter we discussed the concept of modularity and how modularity can be defined through usage. Having available such usage and modularity knowledge, one can direct the project's resources toward the weakest modules in the project. Resource distribution has always been of vital importance in the software development process and our approach can be easily plugged-in the resources decision making.

Besides assisting the API project managers in decision making, the presented approach can also assist the API users in deciding which API is recommendable to use. As low modularity is usually a sign of an inadequate design, which more often than not leads to defective or inefficient code—modularity information points to the quality of the API and users usually prefer making informed decisions regarding the quality of the technology they are using.

The presented approach can also be used to expand and improve existing defect detection techniques.

## Chapter 5

# Conclusions and Future Work

*“If you follow reason far enough it always leads to conclusions that are contrary to reason.”*

– Samuel Butler.<sup>1</sup>

The field of mining software archives and analyzing project’s evolution is vast. What characterizes this field is the enormous amount of information available that offers the possibility for exploring trends and patterns within this information body. This dissertation has addressed the following open problems:

**Estimating Quality** This has always been a critical issue for each software project. A lot of research and work has been done in estimating which parts of the project are of high quality and which ones still need attention.

My research has contributed to estimating the quality of APIs and modules through the usage of evolution trends and patterns. Chapter 2 presents one of the most extensive study done so far on the usage and quality of software libraries. The collection of usage data can not only be used for estimating the current quality of a product, but to also estimate the impact of a change and thus predict future quality and support issues. Similarly, we have also developed an approach for evaluating the modularity and thus the quality of modules, which is presented in Chapter 4.

AKTARI: <http://www.st.cs.uni-saarland.de/softevo/aktari.php>

---

<sup>1</sup>Samuel Butler (1835—1902) was an iconoclastic Victorian author who published a variety of works.

**Defects Detection** One of the main problems in software are the defects in the software. Software companies each year invest enormous amount of money and time in testing the product and identifying existing, undetected defects.

Chapter 3 presents the first technique to combine the two worlds of specification mining and mining software archives. This lead to the development of a defect detection tool that is able to detect a collection of code defect, no other tool developed so far could find.

LAMARCK: <http://www.st.cs.uni-saarland.de/models/lamarck/>

**Defects Prevention** Fixing a defect, once present in a system, is time and resource-consuming. That is why a lot of effort is being spent on detecting possible weak-spots in a project and preventing the introduction of defects into the system.

The technique presented in Chapter 3 can as well be used, with a precision of 90% to 100%, to predict a possible defect location and thus prevent the introduction of defects in the source code. No other tool in the literature has so far reported such high precision values.

**Distributing Resources** Resources, like people, time and money, are always scares when it comes to developing a software and the quality of the final product can very much depend on the correct distribution of these resources. That is why existing resources need to be distributed correctly to insure quality maximization and the costs minimization.

The techniques presented in this thesis can serve as a way for developers and managers to determine, which components of the software are in need of refactoring, testing or redesign. The technique in Chapter 2 can be used to determine API components that need more testing; the technique in Chapter 3 points to locations that are in need of refactoring; the techniques in Chapter 4 can be used to point to components that are in need of redesign.

The above mentioned research problems are vast and even though this thesis does not claim that all those problems have been completely solved, my belief is that the contributions of this thesis push the frontier of their solutions further to their complete resolution.



## 5.1 Future Work

One can step on the developed techniques in this thesis and develop new approaches and extend the existing ones. Here are a few directions, in which this work can be extended and used.

**Trends Analysis** This thesis gives a baseline of approaches for analyzing evolution and usage trends. More work however can be done in the direction of classification of the trends and the forecasting the future behavior of a trend. Some techniques that could be used have been discussed in Chapter 2.

**Evolution of evolution patterns** In Chapter 3 was introduced the concept of evolution patterns. One could further examine those patterns and see how they evolve over time. This means extracting information from more than two versions of a project. Such knowledge will contribute to the better understanding of how objects evolve and will also improve on the amount and quality of patterns that can be detected.

**Evolution of Components** This dissertation has been mainly concerned with the evolution of libraries, library versions, objects and modules – a range of different in size and domain software components. The presented techniques however can be applied to an even broader set of software components. One big area, that could be addressed is the area of testing and test suits, where analyzing the existing trends in test suit changes can greatly contribute to the improvement of those suits and thus to the end quality of the software.



# Bibliography

- [1] DAGENAIS, B., AND ROBILLARD, M. P. Recommending adaptive changes for framework evolution. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering* (New York, NY, USA, 2008), ACM, pp. 481–490.
- [2] DAGENAIS, B., AND ROBILLARD, M. P. Recommending adaptive changes for framework evolution. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering* (New York, NY, USA, 2008), ACM, pp. 481–490.
- [3] DAGENAIS, B., AND ROBILLARD, M. P. Semdiff: Analysis and recommendation support for api evolution. In *Proceedings of the 31st International Conference on Software Engineering* (Washington, DC, USA, 2009), ICSE '09, IEEE Computer Society, pp. 599–602.
- [4] DIG, D., COMERTOGLU, C., MARINOV, D., AND JOHNSON, R. E. Automated detection of refactorings in evolving components. In *ECOOOP* (2006), pp. 404–428.
- [5] DIG, D., AND JOHNSON, R. The role of refactorings in API evolution. In *Proceedings of the 21st IEEE International Conference on Software Maintenance* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 389–398.
- [6] FLURI, B., ZUBERBÜHLER, J., AND GALL, H. C. Recommending method invocation context changes. In *RSSE '08: Proceedings of the 2008 international workshop on Recommendation Systems for Software Engineering* (New York, NY, USA, 2008), ACM, pp. 1–5.
- [7] GANTER, B., AND WILLE, R. *Formal concept analysis: Mathematical foundations*. Springer, Berlin-Heidelberg, 1999.

- [8] GEGICK, M., ROTELLA, P., AND WILLIAMS, L. Predicting attack-prone components. In *Proceedings of the 2009 International Conference on Software Testing Verification and Validation* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 181–190.
- [9] GÖTZMANN, D. N. Formale Begriffsanalyse in Java: Entwurf und Implementierung effizienter Algorithmen. Bachelor thesis, Saarland University, 2007. Publication and software available from <http://code.google.com/p/colibri-java/> (accessed 6 April 2010).
- [10] HASSAN, A. E. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering* (Washington, DC, USA, 2009), ICSE '09, IEEE Computer Society, pp. 78–88.
- [11] HENKEL, J., AND DIWAN, A. Catchup!: capturing and replaying refactorings to support api evolution. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering* (New York, NY, USA, 2005), ACM, pp. 274–283.
- [12] HOLMES, R., AND WALKER, R. J. Informing eclipse api production and consumption. In *eclipse '07: Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange* (New York, NY, USA, 2007), ACM, pp. 70–74.
- [13] HOLMES, R., AND WALKER, R. J. Informing Eclipse API production and consumption. In *eclipse '07: Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange* (New York, NY, USA, 2007), ACM, pp. 70–74.
- [14] HOLZ, W., PREMRAJ, R., ZIMMERMANN, T., AND ZELLER, A. Predicting software metrics at design time. In *Proceedings of the 9th International Conference on Product Focused Software Process Improvement* (June 2008), pp. 34–44.
- [15] HOVEMEYER, D., AND PUGH, W. Finding bugs is easy. *SIGPLAN Not.* 39 (December 2004), 92–106.
- [16] KERNER, S. M. Apache Maven Goes Commercial. <http://www.serverwatch.com/news/article.php/3784681>, November 2008.
- [17] KIM, M., AND NOTKIN, D. Discovering and representing systematic code changes. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 309–319.

- [18] KIM, S., PAN, K., AND E. E. JAMES WHITEHEAD, J. Memories of bug fixes. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of Software Engineering* (New York, NY, USA, 2006), ACM, pp. 35–45.
- [19] KIM, S., PAN, K., AND WHITEHEAD, JR., E. J. When functions change their names: Automatic detection of origin relationships. In *Proceedings of the 12th Working Conference on Reverse Engineering* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 143–152.
- [20] LI, Z., AND ZHOU, Y. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of Software Engineering* (New York, NY, USA, 2005), ACM, pp. 306–315.
- [21] LINDIG, C. Mining patterns and violations using concept analysis. Tech. rep., Saarland University, Saarbrücken, Germany, June 2007.
- [22] LIVSHITS, V. B., AND ZIMMERMANN, T. Dynamine: Finding common error patterns by mining software revision histories. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (September 2005), ACM, pp. 296–305.
- [23] MCKEAN, E., Ed. *The New Oxford American Dictionary*, 2nd ed. Oxford University Press, Oxford, 2005.
- [24] MILEVA, Y., WASYLKOWSKI, A., AND ZELLER, A. Mining evolution of object usage. In *ECOOP 2011 – Object-Oriented Programming* (2011), M. Mezini, Ed., vol. 6813 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 105–129.
- [25] MILEVA, Y. M., DALLMEIER, V., BURGER, M., AND ZELLER, A. Mining trends of library usage. In *IWPSE-Evol '09: Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops* (New York, NY, USA, 2009), ACM, pp. 57–62.
- [26] MILEVA, Y. M., DALLMEIER, V., AND ZELLER, A. Mining api popularity. In *TAIC PART'10: Proceedings of the 5th international academic and industrial*

- conference on Testing - practice and research techniques* (2010), Springer-Verlag, pp. 173–180.
- [27] MILEVA, Y. M., AND ZELLER, A. Assessing modularity via usage changes. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools* (New York, NY, USA, 2011), PASTE '11, ACM, pp. 37–40.
  - [28] MISIRLI, A. T., MURPHY, B., ZIMMERMANN, T., AND BENER, A. B. An explanatory analysis on eclipse beta-release bugs through in-process metrics. In *Proceedings of the 8th International Workshop on Software Quality* (September 2011).
  - [29] NGUYEN, H. A., NGUYEN, T. T., WILSON, JR., G., NGUYEN, A. T., KIM, M., AND NGUYEN, T. N. A graph-based approach to API usage adaptation. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications* (New York, NY, USA, 2010), OOPSLA '10, ACM, pp. 302–321.
  - [30] NGUYEN, T. T., NGUYEN, H. A., PHAM, N. H., AL-KOFAHI, J., AND NGUYEN, T. N. Recurring bug fixes in object-oriented programs. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering* (New York, NY, USA, 2010), ACM, pp. 315–324.
  - [31] NGUYEN, T. T., NGUYEN, H. A., PHAM, N. H., AL-KOFAHI, J. M., AND NGUYEN, T. N. Graph-based mining of multiple object usage patterns. In *ES-EC/FSE '09: Proceedings of the the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering* (New York, NY, USA, 2009), ACM, pp. 383–392.
  - [32] PERKINS, J. H. Automatically generating refactorings to support api evolution. *SIGSOFT Softw. Eng. Notes* 31, 1 (2006), 111–114.
  - [33] REN, X., RYDER, B. G., STOERZER, M., AND TIP, F. Chianti: a change impact analysis tool for java programs. In *Proceedings of the 27th international conference on Software engineering* (New York, NY, USA, 2005), ICSE '05, ACM, pp. 664–665.
  - [34] ROBILLARD, M. P. What makes APIs hard to learn? Answers from developers. *IEEE Software* 26, 6 (2009), 26–34.
  - [35] ROGERS, E. M. *Diffusion of Innovations*. The Free Press, 1962.

- [36] SCHULER, D., AND ZIMMERMANN, T. Mining usage expertise from version archives. In *MSR '08: Proceedings of the 2008 international working conference on Mining Software Repositories* (May 2008).
- [37] SUROWIECKI, J. *The Wisdom of Crowds*. Anchor, 2005.
- [38] THUMMALAPENTA, S., AND XIE, T. Spotweb: detecting framework hotspots via mining open source repositories on the web. In *MSR '08: Proceedings of the 2008 international working conference on Mining Software Repositories* (New York, NY, USA, 2008), ACM, pp. 109–112.
- [39] WASYLKOWSKI, A., ZELLER, A., AND LINDIG, C. Detecting object usage anomalies. In *Proceedings of the 11th European Software Engineering Conference held jointly with 15th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (September 2007), pp. 35–44.
- [40] WEISSGERBER, P., AND DIEHL, S. Identifying refactorings from source-code changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 231–240.
- [41] WILLIAMS, C. C., AND HOLLINGSWORTH, J. K. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Trans. Softw. Eng.* *31*, 6 (2005), 466–480.
- [42] XING, Z., AND STROULIA, E. Refactoring detection based on umldiff change-facts queries. In *Proceedings of the 13th Working Conference on Reverse Engineering* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 263–274.
- [43] ZHONG, H., XIE, T., ZHANG, L., PEI, J., AND MEI, H. MAPO: Mining and recommending API usage patterns. In *ECOOP 2009: Proc. 23rd European conference on Object-Oriented Programming*, Lecture Notes in Computer Science 5653. Springer-Verlag, Berlin, 2009, pp. 318–343.
- [44] ZIMMERMANN, T. Fine-grained processing of CVS archives with APFEL. In *Eclipse '06: Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange* (New York, NY, USA, 2006), ACM, pp. 16–20.
- [45] ZIMMERMANN, T., AND NAGAPPAN, N. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th international conference on Software engineering* (New York, NY, USA, 2008), ICSE '08, ACM, pp. 531–540.

- [46] ZIMMERMANN, T., NAGAPPAN, N., AND WILLIAMS, L. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation* (April 2010).
- [47] ZIMMERMANN, T., WEISSGERBER, P., DIEHL, S., AND ZELLER, A. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering* (May 2004), IEEE Computer Society, pp. 563–572.