Maintainability and Source Code Conventions: An Analysis of Open Source Projects

Michael Smit, Barry Gergel, H. James Hoover, and Eleni Stroulia
Department of Computing Science
University of Alberta
Edmonton, Canada
Email: {msmit,gergel,hoover,stroulia}@cs.ualberta.ca

***Abstract*—Maintainability is a desirable property of software, and a variety of metrics have been proposed for measuring it, all based on different notions of complexity. Although these metrics are useful, complexity is only one factor influencing maintainability. Practical experience in software development has led to a set of best practices and coding conventions that are believed to make source code easier to read, understand and maintain. Based on a survey of software engineers, we identify the relative importance of 71 coding conventions to maintainability. We propose a metric that offers a different perspective on maintenance, namely a "convention adherence" metric based on the number and severity of violations of these coding conventions. We examine the code repositories of four open-source Java projects to measure their adherence to coding conventions over the life of the project, based on both their self-identified conventions and those of the convention-adherence metric. Through our analysis, we discovered several interesting phenomena, including pre-release effort to bring new code in line with desirable conventions, effective usage of automated code convention checkers as part of the build process to improve adherence, variations in adherence over the software lifecycle, and a class of conventions consistently ignored in open source projects.**

I. Introduction

Maintainability is an important aspect of software quality. Software systems are part of the backbone structure of every modern organization and represent a significant and ongoing capital expense. Management is keenly interested in the maintainability of software deployed within their organizations, and how to reduce software maintenance costs while satisfying evolving user software requirements. More importantly, organizations face a challenge in the modern competitive landscape: speed and surprise are becoming required tools in the race for competitive advantage [1]. The accelerating pace of innovation and change leaves organizations with smaller windows of opportunity, and technology is reinforcing the importance of knowledge management within organizations [2]. Customers are becoming more fickle and demanding as they push for higher quality, more choice, and better service. This shrinks product life cycles and challenges the profitability of organizations [3]. The increasing uncertainty and unpredictability of this dynamic environment is forcing organizations to search for new tools and methods to cope with this accelerated pace of change.

Existing metrics for maintainability focus on complexity. While valuable, we suggest these metrics would benefit by including more factors believed to play a role in writing readable and maintainable code, namely, software development best-practices and code conventions that have evolved over time. Advocates of these conventions suggest that they produce better code, and software projects generally publish some set of conventions they adhere to, in order to keep the source code consistent. The intuition behind these conventions is plausible: hard-coded strings and numeric constants make code more difficult to update; well-formatted comments help new developers understand the code or use an API; a well-defined naming style that matches existing libraries helps associate syntax with semantic meaning.

To analyze the potential of code-convention adherence as a predictor of maintainability, we systematically examined all of the revisions of four open-source Java projects, checking for adherence to their self-imposed standards and to a set of conventions a panel of software engineers identify as important to maintainability. To support this work, we developed a set of tools and visualizations to semi-automate this process. We found that, when conscious of the conventions, *i.e.* when the project has an explicit code-convention adherence policy enforced by automated code-convention checkers, developers are willing to expend maintenance effort to improve adherence; when not as conscious of them, violations are prevalent. Some of the projects that make a conscious effort to improve the code quality in accordance with their conventions do so prior to a new software release. Automated code convention checkers appear to help projects improve adherence to their coding conventions. Finally, we observed that there is a distinct set of standard conventions that are consistently broken in the open source projects used in our study.

The remainder of the paper is as follows. Section II explores the background and related work that has motivated this paper. In Section III, we take a closer look at source code conventions and best practices. We describe the methodology we used to study the four open source applications in Section IV and present our results in Section V. Finally in Section VI and Section VII, we suggest possible future research directions and review our results.

II. Background & Related Work

The quality of a software system is an important factor to its success *i.e.*broad adoption and long-term use and evolution;

however, software-quality metrics remain difficult to evaluate as the various stakeholder groups often have varying agendas and concerns. Consequently, a variety of software-quality models have been proposed but no single model is universally recognized. One widely accepted quality model is defined in the ISO/IEC 9126-1 standard [4]. The model defines software quality using six attributes: functionality, reliability, usability, efficiency, maintainability, and portability.

Developers and users are inherently concerned about quality, yet view quality from two different perspectives. Users view quality characteristics that focus on attributes such as usability and performance. Tonella and Abebe define these characteristics as external qualities [5]. They are dynamic and generally measured at run-time. Conversely, internal qualities, such as maintainability, are more likely to be reflected in the static structures of the software [5], [6], and developers are very interested in these qualities. Mari and Eila grouped these two categories using alternative descriptive terms where internal qualities are defined as evolution qualities and external qualities are execution qualities [6]. In this work, we focus on maintainability, which is an internal static quality characteristic. The evolution quality view proposed by Mari and Eila is particularly descriptive of the evolving role that maintainability plays in software quality.

Maintainability has a significant impact on the success of a software application. The cost of maintenance through the life cycle is generally accepted to be between 50-70% of the total cost for the software. Unsurprisingly, maintainability has received significant attention from software engineers and developers. The ISO/IEC 25010-2011 standard defines maintainability as [7] "The capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications."

Multiple metrics have been proposed for measuring maintainability. The majority of these metrics focus on evaluating complexity. The Halstead complexity metric [8] and McCabe's cyclomatic metic [9] are two prominent complexity metrics. These metrics are combined with the number of lines of code to compute the maintainability index [10]. But complexity is only one factor affecting maintainability. Practitioners intuitively agree that, even complex software can become more maintainable when it is understandable, whether through good documentation or through improved code readability [11]. These properties are also important in evaluating the maintainability of software.

Software source code is a written language that codifies the design of the software. Implementation decisions made by developers, such as the use of magic numbers and hard coded strings, negatively impact the readability, the understandability and, ultimately, the maintainability of a software system by introducing brittleness that reduces modifiability. Source-code conventions are established to improve the maintainability of source code and capture best practices [12], [13]. Li and Prasad reported that although developers understood the importance of using code conventions, they did not follow them when development needed to be completed quickly [14].

III. Source Code Conventions

Source code conventions have co-evolved with programming languages, motivated by the assumption that code developed using consistent conventions on issues such as naming of programming elements, inlined documentation, and organization of the syntactic structures is easier to read, is likely to suffer less from careless mistakes, and is more likely to conform to best practices. As a result, such code is likely to be more maintainable and of higher quality, in general.

Some conventions are generally applicable while others are specific to one language (*e.g.*Java) or to a paradigm (*e.g.*object-oriented programming). Tools can be used to enforce these conventions (for example, FindBugs[1], Checkstyle[2], and Jtest[3]). The term *code conventions* is used as a broad umbrella term that includes best practices around naming, syntactic and commenting style. Clearly, not all of these conventions are equally relevant as far as the code's readability, understandability and maintainability is concerned.

To identify the code conventions most important to maintainable code, we solicited input from a 'panel' of seven software engineers. Each had a Masters degree or higher, with many years programming experience and theoretical knowledge of coding conventions and best practices. All panel members have current or former associations with our research lab but are not involved directly in this project. A total of 71 different coding conventions were presented to the panel. The conventions and their descriptions were modified from the Checkstyle documentation (and so are automatically detectable), with the checks that were not specific or difficult to enforce excluded. For each, the rationale was provided (and the source identified where possible). The respondents were asked to answer on a 7-point Likert importance scale[4] how important they believed the convention was to "ensure the ability to change, adapt, or update source code to meet changing requirements or fix bugs". They were asked to provide a rating for both Per-project and Universal importance (that is, the relative importance if a project identifies this convention as one they intend to follow, versus the importance of this convention to **all** (or most) software projects).

The average scores for each code convention are shown in Table I. We've grouped the code conventions into three groups (Important, Minor, and Unimportant): values that are (or round up to) Important or Very Important are labelled Important; values that are (or round up to) Somewhat Important are Minor, and the remainder are labelled unimportant. Values with high standard deviation (> 1.5) are noted with an asterisk.

Some of the code conventions use "magic number" thresholds; *i.e.*, fixed numeric constants that are set based on some set of best practices. For these, the question used the default in Checkstyle; however, respondents were asked if they would suggest a different number. Their suggestions were incorporated into the set of Important and Minor standards that we use in our experiments. In particular, the maximum

[1]http://findbugs.sourceforge.net/

[2]http://checkstyle.sourceforge.net/

[3]http://www.parasoft.com/jsp/products/jtest.jsp?itemId=14

[4]7 is very important, 6 important, 5 somewhat important, 4 neutral, 3 somewhat unimportant, 2 unimportant, 1 very unimportant.

Convention	Universal	Per-Project
Public methods and constructors require javadoc-style comments.	6.8	7.0
Public classes and interfaces require javadoc-style comments.	6.8	7.0
String literals should not be compared using == or !=.	6.8	6.8
Each variable declaration should be its own statement and on its own line.	6.7	6.7
Classes that override Object.equals() must also override Object.hashCode().	6.5	6.7
Loop control variables must not be modified inside the for block.	6.5	6.5
The fields of a class should be declared private (exceptions: static final members, serialVersionUID).	6.3	6.8
Javadoc comments must be well-formed (per the requirements of the javadoc tool from Sun/Oracle).	6.3	6.8
Utility classes (classes that contain only static methods or fields in their API) should not have public constructors.	6.3	6.7
The default case should appear after all the cases in a switch statement.	6.3	6.5
Avoid .* imports.	6.2	6.7
Omit any unnecessary import statements.	6.2	6.5
Magic Numbers are numeric literals 'buried' in the code instead, and should be avoided (except -1, 0, 1, and 2).	6.2	6.3
Multiple occurrences of the same string literal within a single file should be refactored to a constant.	6.2	6.3
Limit the number of parameters for methods and constructors to 7 at most.	6.0	6.3
If any equals() method is defined, Object.equals(Object o) must be overridden.	6.0	6.2
Limit the number of methods in a class to 100 or fewer.	6.0	6.2
Even when optional (e.g. after while, if, for statements), curly braces should be used.	* 5.8	6.3
Checks for uncommented main() methods (debugging leftovers).	5.8	6.0
Cyclomatic complexity less than 10	5.8	6.0
Do not use nested 'free' blocks (not associated with control statement).	5.8	5.8
Exceptions should be immutable; that is, have only final fields.	5.7	6.3
Variables should not be assigned in subexpressions, such as in String s = Integer.toString(i = 2);	5.7	6.2
Catching (or throwing) java.lang.Exception, .Throwable or .RuntimeException is almost never acceptable.	5.7	5.8
Only one statement per line is permitted.	5.7	5.8
Follow the Java naming conventions.	5.5	6.5
Local variables that never have their values changed should be declared final.	* 5.5	* 5.8
Long constants should be defined with an upper ell, i.e. (2345L not 2345l).	5.5	5.7
Method parameters must not be assigned new values.	5.5	5.7
There should be no space between the identifier of a method definition and the left parenthesis of the parameter list.	5.5	5.7
Avoid overly complicated boolean expressions: (b == true), (!false), if (valid()) return true, etc.	* 5.5	* 5.5
Classes should rely on a maximum of 20 other classes	5.5	5.5
String literals being compared to String variables should be on the left side of an equals() comparison.	5.4	5.8
The finalize() method should never be used.	5.4	5.6
All switch statements must have a 'default' clause.	* 5.3	6.5
The Object.clone() method should not be overridden.	5.3	6.0
Empty blocks should be avoided.	5.3	5.7
Imports must be precisely ordered and grouped.	5.3	5.7
The parts of a class or interface declaration should appear in the Sun conventions order.	5.3	5.5
Limit lines of code to no more than 80 characters.	5.3	5.2
Private methods and constructors require javadoc-style comments.	5.2	5.3
Private classes and interfaces require javadoc-style comments.	5.2	5.3
The number of possible execution paths through a function should be limited.	5.2	5.2
Whitespace around the Generic tokens $<$ and $>$ should meet the Sun standard.	* 5.2	6.2
End of line comments should not be used.	5.2	5.8
References to instance variables and methods of the present object must be explicitly of the form this.varName *etc.*	5.2	5.8
Certain Java tokens should be preceded and followed by whitespace.	5.2	5.5
The number of instantiations of other classes within a given class should be less than 7.	5.0	5.4
Certain classes (e.g. Abstract) should not be used as types in variable declarations, return values or parameters.	* 5.0	5.3
Nested (internal) classes/interfaces should be declared at the bottom of the class after all method and field declarations.	5.0	5.3
Indent 4 spaces for each nested block, closing parentheses aligned with the statement starting the block.	* 4.8	* 5.2
A local variable or a parameter should never have the same variable name as a field in the same class.	* 4.7	5.5
All classes should declare constructors (i.e. never rely on a default constructor).	* 4.7	* 5.2
Interfaces are designed to describe a type and should therefore define at least one method (and not just constants).	* 4.7	* 5.2
There should be no spaces immediately following an open parenthesis or following a closing parenthesis.	4.7	5.2
Files should end with a new line.	* 4.7	* 5.0
Array-type definitions should be Java-style: String[] args and not C-style: String args[].	4.7	* 4.8
Methods are limited to 50 SLOC, classes to 1500, and files to 2000.	* 4.5	* 5.0
Restrict the number of number of &&, etc. in an expression.	4.5	4.8
Restrict throws statements to 1 Exception.	* 4.3	* 5.7
A local variable should not have the same name as a field, EXCEPT in the constructor and in Setter method parameters.	4.3	5.0
Array initializations that span multiple lines should contain a trailing comma.	* 4.3	4.7
Method/constructor/catch block parameters must be final.	4.3	4.7
The order of modifiers should follow the suggestions of the Java Language specification.	4.3	4.7
Opening curly brace {: end of control statement line. Closing curly brace }: end of its own line after the block.	* 4.2	5.0
If, For, and Try statements should each have at most additional statement nested inside.	* 4.2	* 4.5
Classes should be 'designed for extension': No code is permitted in public methods of extensible classes.	4.0	5.0
Avoid redundant modifiers.	* 4.0	* 4.2
Avoid unnecessary parentheses.	* 4.0	* 4.0
Avoid inline conditionals like: expression ? true : false.	* 3.7	* 4.2
The number of return statements in a method should be limited (Default: 2). Exception: all equals() methods.	* 3.7	* 4.0

TABLE I: Average importance scores for 71 code conventions, partitioned into "Important", "Minor", and "Unimportant". Scores with high standard deviation are denoted with *.

Name	Start SLOC	End SLOC	Committers	Start Date	End Date	Δ Time	Δ SLOC	SLOC/Day
Ant	3849	106547	47	13/01/2000	11/03/2011	4075	102698	25.20
Derby	235485	353398	36	11/08/2004	28/03/2011	2420	117913	48.72
Hadoop	37636	68531	29	18/05/2009	24/03/2011	675	30895	45.77
JFreeChart	82434	100354	2	19/06/2007	30/03/2010	1015	17920	17.66

TABLE II: Metadata for the open source projects included in this paper.

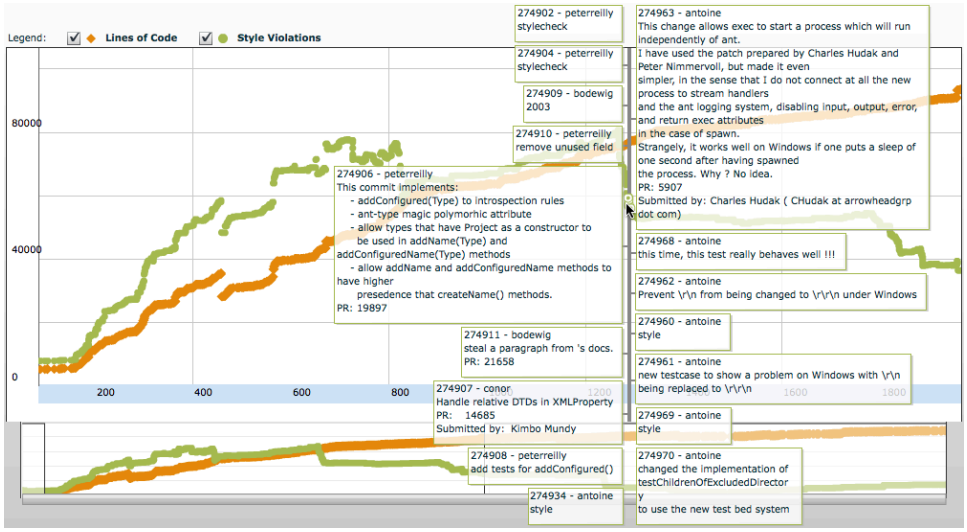


Fig. 1: Screen capture of tool for interactive exploration of lines of code, code violations, and svn commit messages for Apache Ant (code violations scaled 10x).

number of methods in a class was decreased from 100 to 75 (best design practices suggest a significantly lower limit; 75 was chosen as a level respondents agreed was definitely bad).

It should be noted that this set of results does not offer a consensus. For all but one convention, at least one respondent answered Important or Very Important. For all but the top 20, at least one respondent entered 4 or lower; for one-third of the conventions, at least one respondent entered a 2 or lower. We recognize that our identification of "Important" conventions, as agreed upon by our "expert panel" will not be universally accepted. From conversations with the respondents, it is clear they believe that every good convention has exceptions, and adherence to conventions in general is secondary to compliance with functional specifications. In the future, we plan to conduct a broader survey of the software engineering community at large, to examine whether a clearer consensus might be reached on the relative importance of these conventions; nevertheless, we believe that through our panel's answers we have a good initial view.

IV. Methodology

We examined four open-source projects (Table II) for their adherence with code conventions over time. The projects were chosen to represent a wide range of activities within the open source community. They vary in size, in terms of physical size, overall complexity, and the number of participating developers. Some of projects, such as Ant, use tools to enforce coding convention standards, while others do not. The selection of application also represents software from a range of domains. Furthermore, each project is primarily Java and uses an SVN repository. With the data from each project, we examined only the trunk of the svn repository, which is typically the base of development for the project. Branches were ignored until their code was merged into the trunk. Our study consisted of the folowing steps.

Identification of relevant revisions. Every commit to an SVN repository increments the revision number. Some commits do not modify the trunk source tree and can be safely ignored[5]. We used the `svn log` command on the path to the project of interest to obtain a set of all relevant commits. Meta data (relevant revision numbers, dates, committers, svn log messages) is collected and stored.

Identification of change sets. We then iteratively check out each of the relevant revisions and obtain a list of added, deleted, and updated (including merged) files. These change sets are subsequently filtered for relevance; the following files are excluded: all non-Java files, all files that appear to be only for testing[6] (JUnit or otherwise), and code identified as non-core by manual examination of the SVN repository.

Analysis of the change set. We count the total source lines of code (SLOC) in each file in the filtered change set (using CLOC[7]). The complete output for every change set is stored in XML format. We then use the Checkstyle tool to test adherence to code conventions; every violation of all three sets,

[5]This especially relevant for Apache projects, as all projects are in the same SVN repository; only 2000-6000 of the 1,100,000 commits are relevant to the projects of interest.

[6]Arguably code conventions that are relevant to the core source are just as important to the test cases; however, our results showed substantially more code-convention violations in the testing classes in most projects.

[7]http://cloc.sourceforge.net/

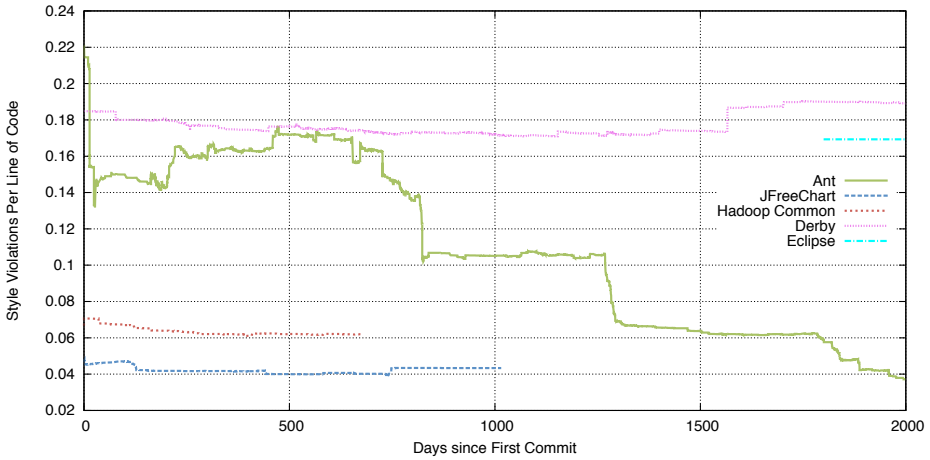


Fig. 2: Code convention violations per line of code, based on self-imposed standards.

Project	Conventions
Ant	checkstyle configuration file
Derby	"The Derby community has not approved a common body of coding standards"[10]; developers are pointed to Sun's conventions. Javadoc is mandatory (not enforced).
Eclipse	Mostly Sun's[11].
Hadoop	checkstyle configuration file
JFreeChart	checkstyle configuration file

TABLE III: Style guidelines self-imposed by projects.

for every revision of every file, is stored in XML.

1) *Their own standards* as identified by their own checkstyle configuration file (where applicable, otherwise we created a checkstyle configuration file for them based on their documented coding standards).
2) *Important standards* as identified by our expert panel.
3) *Minor standards* as identified by our expert panel.

Analysis of code-convention violations. We explore the cached convention violation data and SLOC metrics using three approaches. The first is a set of tools for extrcting select information from the XML and generating visualizations using Gnuplot. The second uses the Google Charts API to produce interactive visualizations with zoomable time windows; mouse-over a data point produces dates and exact values. The third is built on an open-source extentions [15] of the time series graph in the Google Charts API. Shown in Figure 1, it offers the same zoomable time windows, but shows svn commit message metadata when hovering the mouse over any point. By zooming into anomalies in the violations and examining the svn commit logs, we can gain insight into what was happening. For example, in the figure we are examining a sharp drop in violations. We see two users making a number of commits specifically intended to improve adherence, and note they are doing so in the midst of adding new features to the code and implementing new tests.

V. Results

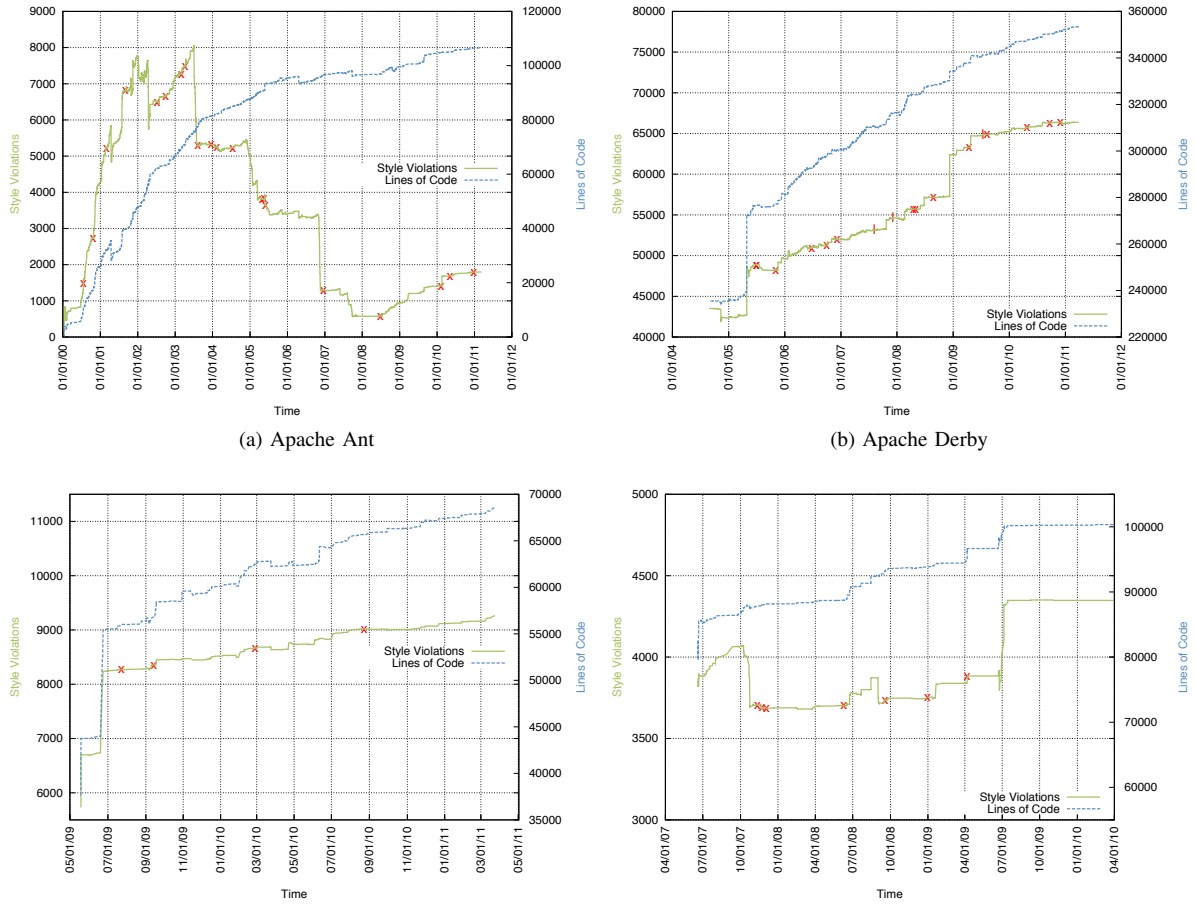
The following sections review some of our findings based on our detailed examination of four open-source projects over time. We focus on adherence to self-imposed standards and adherence to those conventions labelled as important by our panel. The "minor" conventions are excluded since their sheer number requires separate treatment in future work (at the latest revision we examined, Ant: 53,000 (0.5 per line of code); Derby: 361,270 (.55); Hadoop: 26,116 (.38); JFreeChart: 12,296 (0.12)).

A. Self-imposed Standards

Software projects typically identify a set of coding conventions to be followed by their developers. We examined adherence to the standards set by the open-source projects shown in Table III. We include the adherence of a 2010 snapshot of the entire Eclipse repository to get a sense of results for a multi-million line project. Three of the projects we examined use a checkstyle configuration file to specify required code conventions; targets are defined in their build script to report on violations. The Derby project does not have a set of standards agreed upon by the community, though they do require javadoc comments and point developers to the Sun conventions, so we evaluated adherence to the Sun conventions. The other two document their standards but do not include tool support or adherence notification. We created Checkstyle configuration files based on their documented standards. In all cases, we excluded all white space checks.

We first examined the ratio of violations to the size of the project (lines of code), over time. The maximum possible ratio is not fixed; types of violations vary based on the code, and some violations relate to comments. There is no objective measure of what the ratio of violations to source lines of code should be; however, relative comparisons can be made.

Fig. 2 shows how violations per line of code changed over time, measured by days since the first commit we examined. This generally indicates the date on which the source code was made public (*e.g.*, committed to a public repository like Apache or Sourceforge), but with the exception of Apache Ant is not the first day of development. The time window is truncated at 2000 days, which mostly affects Ant. Eclipse is shown as a line, but is actually only a single data point. Hadoop and JFreeChart, both Checkstyle users, show consistently low results. Ant, also a Checkstyle user, shows low



(a) Apache Ant

(b) Apache Derby

(c) Apache Hadoop Common

(d) JFreeChart

Fig. 3: Violations of self-imposed code standards (left axis) and SLOC (right axis); release points shown by 'x'.

results eventually – it should be noted that the Checkstyle tool is in use *today*, but did not exist when Apache Ant first started.

Ant shows a particularly interesting pattern, but all four exhibit noticeable spikes and drops. A closer examination of the four projects is shown in Fig. 3; the violations are shown on their own axis to better show how the violation count changes in relation to the source code. The lines are annotated with release points (though it is not always clear exactly which SVN revision was packed for release, these points are close to accurate). The varying axes across all four subfigures (and the fact that four different standards were enforced) make across-project comparisons difficult for anything but the relationship between the growth of the code and the number of violations.

Checkstyle was first introduced to the Ant project in early 2002, which led to the first sharp drop – before that, violations had been increasing in step with the size of the code. In subsequent years, the recurring pattern is growth of the code and the number of violations, followed by a few weeks or even months of deliberate effort to reduce these violations. These sharp drops appear to immediately precede notable releases. Most convention-related fixes have happened in the maintenance phase of the software, where the pace of new additions to the code has slowed down. JFreeChart exhibits a similar though less pronounced pattern in the first few years: effort to "clean up" the code immediately prior to release. However, this cleaning effort hasn't happened recently.

Derby and Hadoop do not exhibit this intentional effort; the growth of the code base has a strong positive correlation with the increase in violations. We calculated the Pearson correlation coefficients (ρ) for the SLOC and violation counts; $+1$ is a perfect increasing linear relationship between the two; -1 is a perfect decreasing linear relationship. While Derby and Hadoop have $\rho = 0.955$ and $\rho = 0.979$ respectively, Ant is slightly negatively correlated ($\rho = -0.455$) and JFreeChart is slightly positively correlated ($\rho = 0.233$).

B. Important code conventions

We now examine adherence to code conventions identified as "Important" by our panel. We begin with Fig. 4, showing the violations per line of code. As in the graph for the self-imposed standards (Fig. 2), we use days since the first commit we examined as a timeline, truncated at 2000 days (Ant descends to .16 over 2000 more days; Derby continues unchanged for 400 days). Eclipse is shown as a line for visibility, but is a single data point.

Comparing to the results for self-imposed standards, the projects using Checkstyle configuration files show an increased

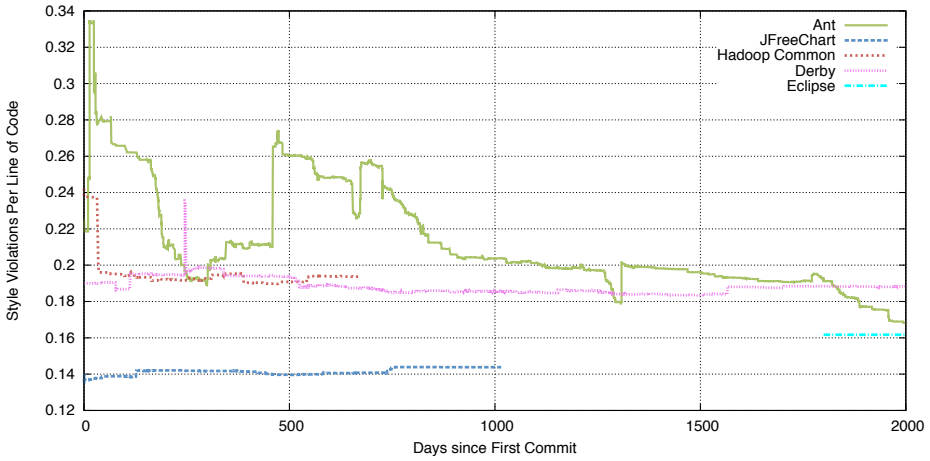
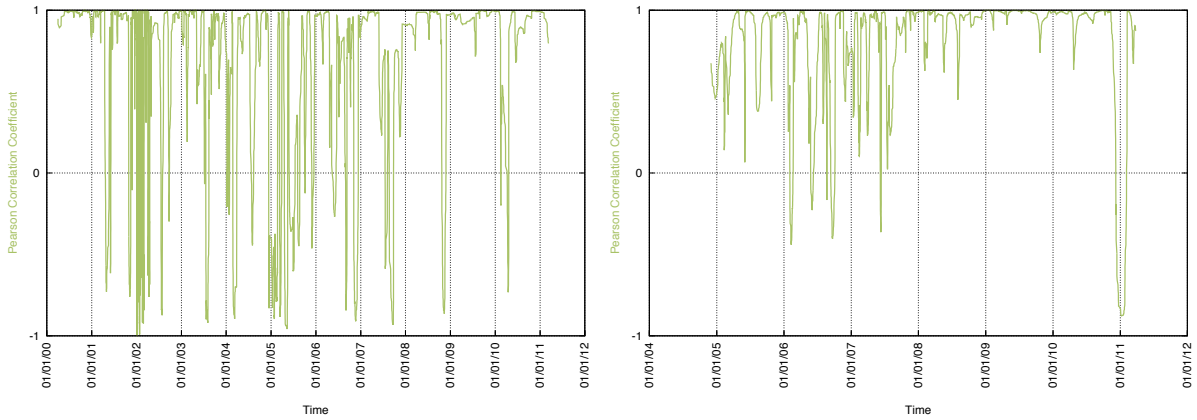


Fig. 4: Code convention violations per line of code, for conventions identified as Important.



(a) Apache Ant

(b) Apache Derby

Fig. 5: Pearson correlation coefficient for sliding 100-commit windows.

ratio of violations per lines-of-code, by a factor of approximately 3. In contrast, the projects held to the (apparently more exacting) Sun conventions exhibit little movement in the ratio. Again with the exception of Ant, the ratio for all the projects remained nearly constant over the life of the project. This is in contrast to our expectations: we expected to see fluctuation around the time of the release points as developers cut corners to produce working code. It is possible this phenomenon exists but is hidden by the large number of violations (*e.g.*, 50,000+ for both Ant and Derby), or that the projects employ a more relaxed release strategy.

There are spikes throughout the Ant line and occasionally in the Derby line. Examination of the commit log indicates there was a check-in of contributed code from outside the regular development team with enough violations to skew the ratio; shortly thereafter, a regular developer cleans up the check-in code. There is intentional effort to keep the ratio from growing. Inspection of commit logs suggests that Ant accepts the most code from outside contributors, though this has not been empirically validated. The Ant ratio peaks after the initial commits of code, in the first few months. However, by the end of the first year of the project, the ratio is at its lowest point ever: a reduction of 36%. This effort to "clean" their code predates their use of the Checkstyle tool. Also note that projects using the checkstyle tool were more compliant with their self-identified standards, but when held to our single standard did not exhibit substantially lower ratios of violations to lines of code.

Next we examine how the correlation between lines of code and violations changes over the life of a project, using a sliding window of 100 commits (sliding-ρ). We suggest that a strongly positive correlation in a 100-commit window indicates consistent use (or misuse) of conventions: that is, that as lines of code are added, roughly the same proportion of convention violations are added, consistently over time. This does not say anything about the slope of the linear relationship – that is, how many violations are introduced per line of code – only that the two grow together. A weak correlation indicates a mix of commits, some increasing (or decreasing) SCAA substantially and some continuing the usual trend. A strongly negative correlation indicates effort (over all 100 commits) to increase adherence to conventions.

The sliding-ρ graph for JFreeChart and Hadoop is not shown here – they were both fairly flat at or near +1; Hadoop shows

one valley into a small negative correlation. In the case of JFreeChart, this may be explained by the small size of the development team. Sliding-ρ for Ant and Derby is shown in Fig. 5. Ant shows periodic 100-commit windows with strong negative correlation. Though new commits increase the number of violations at a higher rate than the other projects, this periodic intentional effort to "clean up" the code decreases the per-line ratio to normal levels (.2) at the current state of the project. Derby shows varying levels of adherence to standards in the commits; only the last valley is deep enough to potentially indicate cleaning effort.

C. Specific Violations

Finally, we examined the types of violations reported (the data presented here is from only the latest revision of each project). The two most common violations were *commenting* and *final local variable* violations: together they accounted for around two-thirds of the violations reported in all of the projects. For the former convention, we counted only missing or incomplete Javadoc-style comments *on **public** types and methods only*. JFreeChart is well-documented; the others less so. The latter convention suggests that if a local variable is declared and assigned but not modified, it should be declared final[12]. The rationale is that this allows the compiler to enforce the fact that the variable never changes – this can avoid bugs that may occur when maintenance efforts add code that changes the value of a variable that other code did not expect to change. It is self-documenting code in that it is a clear explanation of the intent of the field. Detractors of this convention point out that "good" design limits the length of methods, reducing the chances of unanticipated changes. The final keyword can also cause confusion: an array or an object reference can be declared final, but the elements of an array and the state of an object can still change. Whether it has merit or not, it is apparent that the convention is not strictly adhered to in these projects.

The results (Fig. 6) show first a pie chart comparing the number of the top two violations to all of the other violations, then show the details of all the other violations in a histogram. The next two problems occurring most frequently in each project (except Apache Derby) are *Magic Numbers* and *Multiple String Literals*; that is, in-line numeric values are used instead of appropriately named numeric constants, and String literals are repeated instead of using a common constant. The latter category is especially troubling for maintenance, as when one of these String literals changes it has to be changed in (at least) two places. Using common constants also enables intent-checking: if you use the value 3.24 without assigning it to a named value, you might have mistyped π, or might mean a different number. Derby has a high occurrence of missing braces (the compiler considers braces optional for single-line blocks, but best practices suggest that they be included anyway in case additional statements need to be added to the block).

Two violations surprisingly high in count were violations of *naming conventions* and *missing or incorrect visibility modifiers*. We previously considered these to be generally accepted conventions – not necessarily the most important to maintenance, but widely accepted in general.

Particularly troubling were lingering *uncommented* `main(String[] args)` *methods* (Derby for instance has 37), public constructors in classes with only static fields and methods (Hide Utility Class Constructor), control variables being modified inside `for` loops, and overly broad `catch` statements that catch `Exception`. The first two make it more difficult for new users and new maintainers to identify the correct entry points to the software (first) and how to use utility classes (second); the first can be useful for debugging but the second is vestigial code. The third is a dangerous practice; maintainers will likely not expect the control variable to be modified outside the control statement. The fourth can result in maintainers adding code to a `try` block that throws (for example) an `IOException` without realizing they have done so, and failing to add appropriate handling code to the `catch` block resulting in exceptions being quietly ignored.

In summary, while some convention violations are of questionable gravity, there is evidence of generally accepted conventions not being adhered to, as well as some potentially dangerous violations. There is support for our suggestion that the number, ratio, and type of convention violations has bearing on the maintainability of a software project.

VI. Future Work

The initial collection of data and the preliminary analysis presented in this paper are the first step in a broader examination of code-convention adherence practices and their impact on software maintainability. We intend to conduct further analysis of the data collected – *e.g.*, how individual contributors impact adherence, how individual conventions or categories of conventions change over the life of the project, how individual files change as they mature, anything related to the minor violations, and weighting the types of violations based on the scores assigned by the panel. We have also collected the same adherence data for more software projects and are continuing to grow the dataset. The need for further analysis motivates additional tools and visualizations; for example, a heat map showing source code files in proportional size, coloured according to their adherence.

In addition to expanding our analysis of code-convention violations and maintainability, we are also interested in understanding the interplay of these violations with the notion of "technicval debt". Technical debt is a metaphor used to describe the practice of sacrificing long-term goals in exchange for the [cheap,fast] achievement of short-term goals (*e.g.*, [16]). We believe that one indicator of growing technical debt is the growing deviation from code conventions and best practices; for example, when a deadline looms, it may be faster to use a literal string than it is to define and document a new constant variable. We have not yet quantified or proven this relationship. Dijkstra references the Buxton Index [17], describing it as the length of time over which an organization or an individual makes plans. We hypothesize that technical debt – and with it, code convention adherence – is related to this notion; that sacrifices of long-term goals

[12]For our test, we excluded method parameters (see Parameter Assignment).

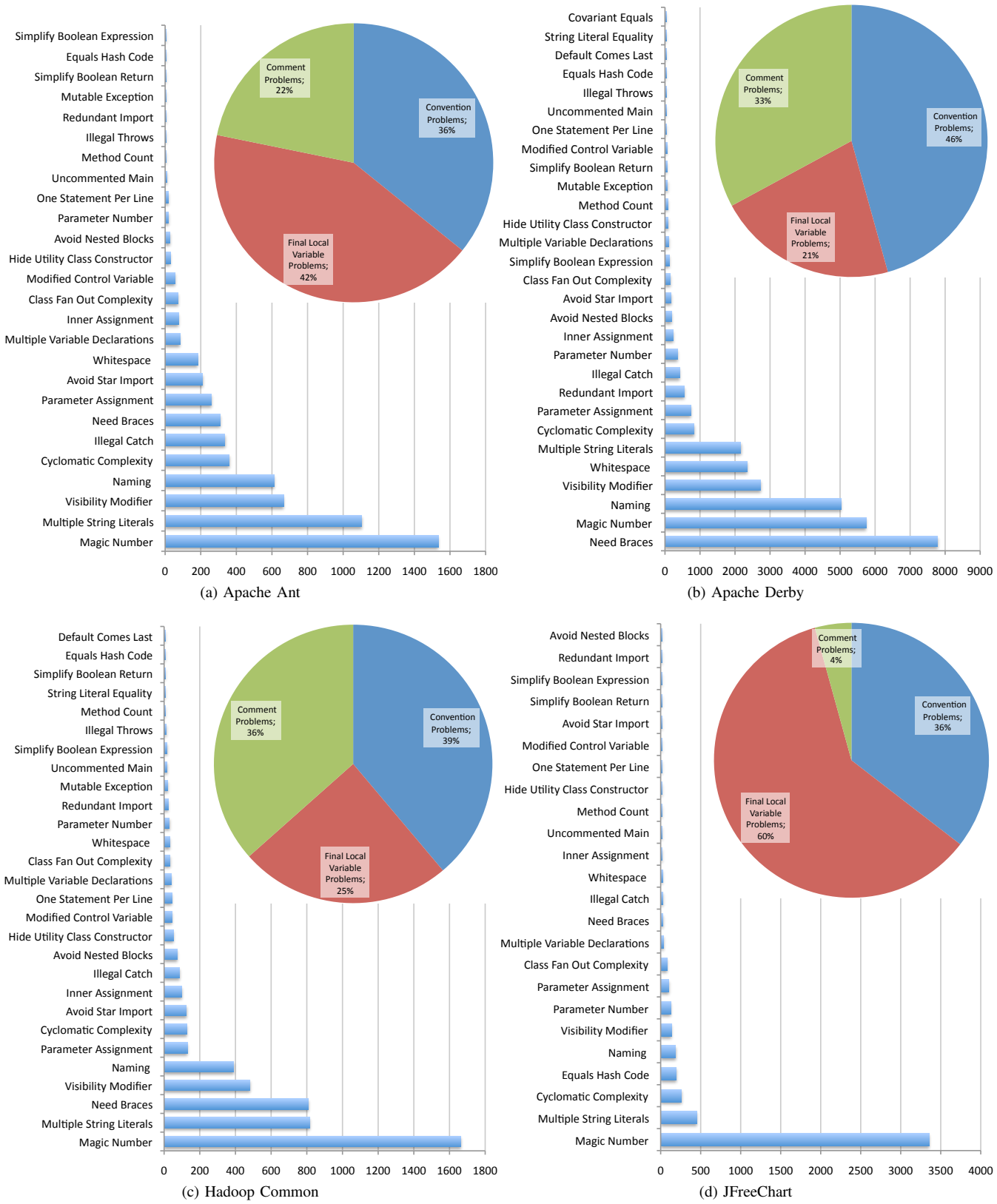


Fig. 6: The type of code convention violations; the pie chart compares final local variable, comment, and other code convention problems; the histogram is details of the other code conventions.

are made unconsciously by developers, who (quite rightly) are looking into only the immediate future. The architects who take a longer view would not make the same choices. There is limited support for this hypothesis in [14], where Li reported that though developers recognized the importance of code conventions to code quality, they did not follow them in practice when meeting deadlines. We are currently formulating a user study that asks users to rate code readability or perform a maintenance task on code with varying levels of convention adherence. This will quantify the relationship between code convention adherence and maintainability. Adherence to certain types of code conventions over time may also be a predictor of technical debt; this relationship should be explored. We are particularly interested in projects with more constrained release schedules and in projects with greater separation between architects and developers.

Finally, combining this static analysis with other forms of analysis could be revealing. For instance, a combination with dynamic analysis of code hot spots (or analyzing high-maintenance files from repositories) could be used to place greater importance on code convention violations in high-maintenance or heavily-used code. Combining with analysis of bug tracking data could quantify the relationship between bugs and code conventions – a relationship we expect exists but which has little empirical evidence.

VII. Conclusion

Existing metrics for measuring maintainability focus primarily on the complexity of the source code. We examined adherence to programming best practices and code conventions as a potential proxy measure for maintainability. We collected data on four open-source Java projects by mining their source code repositories and running an automated code convention adherence checker. We first examined adherence to their own agreed-upon community standards. We then identified the code conventions that a panel of software engineers and developers considered important to writing maintainable code, and examined adherence to those. We briefly described a set of tools we developed to semi-automate this type of analysis.

We found that projects that use an automated best practices checker had better adherence to their self-imposed standards, but did not have better adherence to our set of code conventions important to maintainability. We found two projects that exhibited intentional effort to decrease the number of convention violations as the project matured, particularly right before release points. In the absence of such efforts, the number of violations grows in a linear relationship with the length of the code (almost perfectly positively correlated) – even for well-respected software projects. The project with the smallest development team had the best convention adherence. We found contributions from outsiders were likely to increase the number of convention violations disproportionately, but that in at least two projects regular contributors made an effort to resolve these violations.

When examining the types of violations, we found problems with basic Javadoc-style comments – typically one of the top two violations, and also the most important convention identified by our panel. Also prevalent were instances of numeric and string literals hard-coded into source code, and missing braces. We identified hundreds of occurrences of programming practices known to be confusing to new developers.

There is support for our suggestion that the number, ratio, and type of convention violations has bearing on the maintainability of a software project. Further work is required to quantify the relationship between the various types of code conventions and maintainability.

Acknowledgements

Our thanks to the software engineering research lab at the University of Alberta for their input to this process. Special thanks to Nikolaos Tsantalis, Marios Fokaefs, Dave Chodos, Ken Bauer, Camilo Arango, Ricardo Sanchez, and Ken Wong. We would also like to thank Ray Patterson from the School of Business at the University of Alberta and Erik Rolland from the A. Gary Anderson Graduate School of Management at the Univerity of California, Riverside for their valuable input and insightful perspective.

References

[1] V. Sambamurthy, A. Bharadwaj, and V. Grover, "Shaping agility through digital options: Reconceptualizing the role of information technology in contemporary firms." *MIS Quarterly*, vol. 27, no. 2, pp. 237–263, 2003.

[2] S. Mathiyalkan, N. Ashrafi, W. Zhang, F. Waage, J.-P. Kuilboer, and D. Heimann, "Defining business agility: an exploratory study," in *Information Resources Management Conference*, 2005, pp. 848–849.

[3] G. Fliedner and R. J. Vokurka, "Agility: competitive weapon of the 1990s and beyond?" *Production & Inventory Management*, vol. 38, no. 3, pp. 19–24, 1997.

[4] "Information technology - software quality characteristics and metrics - part 1: Quality characteristics and sub-characteristics," p. 21, 1996.

[5] P. Tonella and S. L. Abebe, "Code quality from the programmer's perspective," in *Advanced Computing and Analysis Techniques in Physics Research*. Proceedings of Science, November 2008, pp. 1–11.

[6] M. Mari and N. Eila, "The impact of maintainability on component-based software systems," in *Euromicro Conference, 2003. Proceedings. 29th*, Sept 2003, pp. 25 – 32.

[7] "Systems and software engineering — systems and software quality requirements and evaluation (square) — system and software quality models," 2011.

[8] M. H. Halstead, *Elements of Software Science (Operating and programming systems series)*. New York, USA: Elsevier Science Inc., 1977.

[9] T. J. McCade, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308–320, 1976.

[10] D. Coleman, D. Ash, B. Lowther, and P. Oman, "Using metrics to evaluate software system maintainability," *Computer*, vol. 27, pp. 44–49, 1994.

[11] D. Posnett, A. Hindle, and P. D. Vanbu, "A simpler model of software readability," in *Working Conference on Mining Software Repositories (MSR-11)*. Waikiki, USA: To Appear, May 2011.

[12] P. W. Oman and C. R. Cook, "A taxonomy for programming style," in *Proceedings of the 1990 ACM annual conference on Cooperation*, ser. CSC '90. New York, NY, USA: ACM, 1990, pp. 244–250.

[13] "Sun/Oracle code conventions for the java programming language," http://www.oracle.com/technetwork/java/codeconvtoc-136057.html.

[14] X. Li and C. Prasad, "Effectively teaching coding standards in programming," in *Proceedings of the 6th conference on Information technology education*, ser. SIGITE '05. New York, NY, USA: ACM, 2005, pp. 239–244.

[15] B. Meutner, "Google finance with Flex code," http://www.meutzner.com.

[16] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, and N. Zazworka, "Managing technical debt in software-reliant systems," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*. NY, USA: ACM, 2010, pp. 47–52.

[17] E. W. Dijkstra, "On the fact that the Atlantic Ocean has two sides," in *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag, 1982, pp. 268–276.