# Code Convention Adherence in Evolving Software

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 desired property of software, and a variety of metrics have been proposed for measuring it, focusing on different notions of complexity and code readability. Many practices have been proposed to improve maintainability through code refactorings: improving the cohesion, simplification of interfaces, renamings to improve understandability. Code conventions are a body of advice on lexical and syntactic aspects of code, aiming to standardize low-level code design under the assumption that such a systematic approach will make code easier to read, understand, and maintain. We present the first stage in our examination of code-convention adherence practices as a proxy measurement for maintainability. Based on a preliminary survey of software engineers, we identify a set of coding conventions that most relate to maintainability. Then we devise a "convention adherence" metric, based on the number and severity of violations of a defined coding convention. Finally, we analyze several open-source projects according to this metric to better understand how consistent different teams are with respect to adopting and conforming to code conventions.**

## I. Introduction and Background

A modern organization depends on software to provide a competitive advantage. Its developers must be agile, able to quickly respond to opportunities and threats. Yet software assets must be long-lasting, partly so that the organization can recoup its investment, and more importantly so that developers can focus on new opportunities, rather than being employed in re-implementing the past. But new opportunities do not stand alone, they must integrate into an organization's existing body of software. Thus, software maintenance is about responding to change using both current and new software assets.

Software maintenance requires two key things: comprehension and design. To change software you have to understand it. Even when you understand it the design has to support the changes you desire. Therefore maintainability needs to be measured in context, relative to the kinds of changes you wish to make.

Multiple metrics have been proposed for measuring maintainability, most of which focus on evaluating complexity. The Halstead complexity metric [1] and McCabe's cyclomatic metric [2] are two prominent complexity metrics. These metrics are combined with the number of lines of code to compute the maintainability index [3]. But complexity is only one factor affecting maintainability, and the relationship may not be as linear as is assumed by complexity-driven metrics.

Practitioners intuitively agree that even complex software can become more maintainable when it is understandable, whether through good documentation or through improved code readability [4]. If this is the case, then good documentation and good code somehow contribute to comprehension of enough design to make (or abandon) the desired code changes. There is no precise metric for good, but in the case of code we have a potential proxy definition. Code is good if it conforms to the code conventions for an organization or project.

Advocates of code conventions suggest that they produce software that is less error-prone and easier to maintain. Furthermore, code conventions appear to support maintainability over a wide range of contexts, and thus benefit the entire organization. Software projects generally publish a set of conventions they adhere to in order to keep the 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. An unstated outcome is that good code makes it easier to understand the design.

This leads us to the central question of our research: To what extent do code conventions increase or reduce maintainability? The questions we answer in this paper are preliminary: How well do projects follow their code conventions over time? How well do they follow code conventions that are considered important for maintenance?

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

## II. Code Conventions

Code is king — it is the ultimate expression of 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. We use *code conventions* as a broad umbrella term that includes best practices around naming, syntactic and commenting style. Code conventions are repositories of rules and guidelines encompassing all concerns with regard to improving code quality.

Code conventions have co-evolved with programming languages. Some conventions are generally applicable while

others are specific to one language (*e.g.* Java) or to a paradigm (*e.g.* object-oriented programming). 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 [5]. Tools have been developed to enforce these conventions (for example, FindBugs, Checkstyle, and Jtest[1]). Given the broad range of concerns captured by these best practices, it is intuitive that not all of these conventions are equally relevant to code readability, understandability, and maintainability.

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 question, the rationale behind the convention was given, and the source identified where possible. The respondents were asked to answer on a 7-point Likert importance scale[2] 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".

We identified 32 code conventions as "Important" based on this input, *i.e.* conventions that were ranked as *Very Important* and *Important*. It should be noted that this set of results was not 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 *Neutral* down to *Very Unimportant*. 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.

## III. Methodology

We examined four open-source projects (Table I) for their adherence with the Important code conventions from Section II, using a set of tools developed for this purpose. We assume that these projects have good maintainability, which is something to be verified. We examined each project's source code repository over a period of time. The projects were chosen to represent a wide range of activities within the open source community. They vary in size, overall architectural complexity, and the number of participating developers. Some projects use tools to enforce code conventions, while others do not. The selection of applications also represents software from a range of domains. Furthermore, each project is primarily Java and uses an Subversion (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 following 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. Meta data (relevant revision numbers, dates, committers, svn log messages) is collected.

**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 (all non-Java files and files only for testing are excluded[3]).

**Analysis of the change set.** We count the total source lines of code (SLOC) in each file in the filtered change set (using CLOC[4]). 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 the sets, for every revision of every file, is stored in XML. The code conventions used included project-specific standards and the Important standards identified previously.

**Analysis of code-convention violations.** We explore the cached convention violation data and SLOC metrics.

## IV. Early Results

This analysis produced significant amounts of data about adherence to code conventions; we present some of these results here, as a first step toward understanding the relationship to maintainability.

Fig. 1 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). Hadoop and JFreeChart, both Checkstyle users, eventual shows consistently low violation count. Ant, also a Checkstyle user, shows low violations eventually — it should be noted that the Checkstyle tool is in use *today*, but did not exist when Apache Ant first started. Sharp drops indicate intentional effort to increase adherence. 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 ($\rho$) 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 negatively correlated ($\rho = -0.455$) and JFreeChart is positively correlated ($\rho = 0.233$).

Next we examined how the correlation between lines of code and violations changes over the life of a project, using

[1] http://findbugs.sourceforge.net/, http://checkstyle.sourceforge.net/, http://www.parasoft.com/jsp/products/jtest.jsp?itemId=14

[2] 7 is Very Important, 6 Important, 5 Somewhat Important, 4 Neutral, 3 Somewhat Unimportant, 2 Unimportant, 1 Very Unimportant.

[3] 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.

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

| 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 I: Metadata for the open source projects included in this paper.

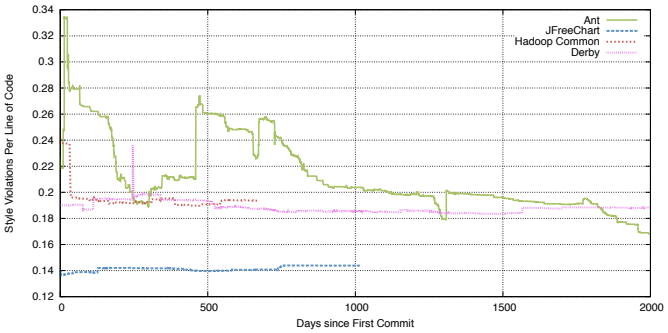
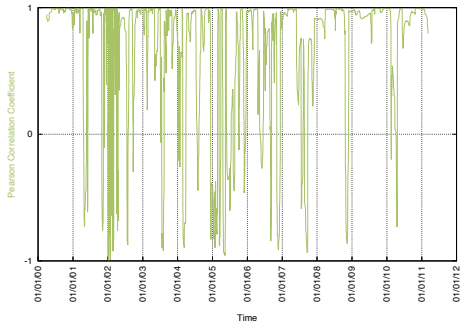
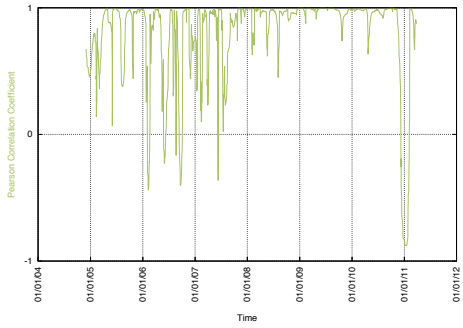



Fig. 1: Code convention violations per line of code, for conventions identified as Important.




(a) Apache Ant




(b) Apache Derby

Fig. 2: Pearson correlation coefficient for sliding 100-commit windows.

a sliding window of 100 commits (sliding-$\rho$). We suggest that a strongly positive correlation in a 100-commit window indicates consistent use 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, either positive or negative, indicates a mix of commits; some increasing (or decreasing) code convention adherence with no overall tread. A strongly negative correlation indicates effort (over all 100 commits) to increase adherence to conventions.

The sliding-$\rho$ 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-$\rho$ for Ant and Derby is shown in Fig. 2. Ant shows occasional 100-commit windows with strong negative correlation. Though new commits increase the number of violations at a higher rate than the other projects, this intentional effort to "clean up" the code decreases the per-line ratio 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 indicate cleaning effort.

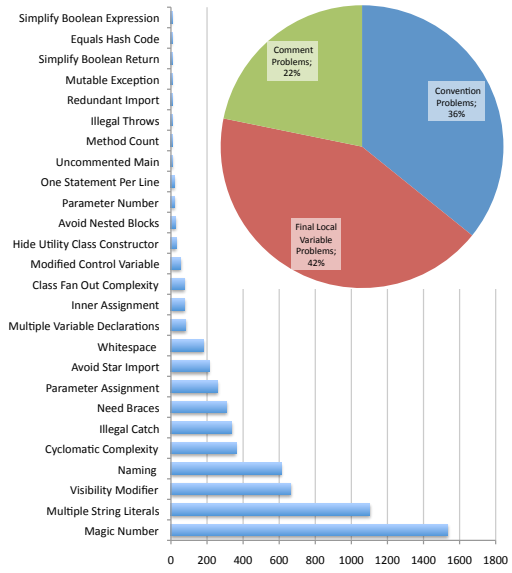
Finally, we examined the types of violations reported (the data presented here is from only the latest revision of each project). The two most common violations were *commenting* and *final local variable* violations: together they accounted for around two-thirds of the violations reported in all of 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 if a local variable is declared and assigned but not modified, it should be declared final[5].

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

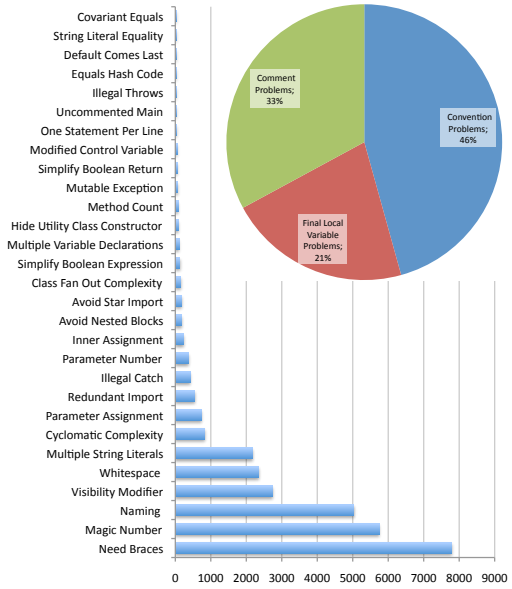
The results (Fig. 3) for Ant and Derby show first a pie chart comparing the number of the top two violations to all of the other violations, then show the details of all the other violations in a histogram. Missing braces, the use of magic numbers, and repeatedly hard-coding the same string literal are surprisingly common. 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. 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.

## V. Conclusion and Future Work

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.




(a) Apache Ant




(b) Apache Derby

Fig. 3: 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.

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.

With the preliminary questions answered, we turn our attention to the relationship between code convention adherence and maintainability. Planned future work includes examining how individual contributors impact adherence, how maintainable these projects actually are, 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, weighting the types of violations based on the scores assigned by the panel, and assessments of maintainability. We have collected adherence data for more software projects and are continuing to grow the dataset.

We are also interested in understanding the interplay of these violations with the notion of "technical 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.*, [6]). 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. There is limited support for this hypothesis in [5], 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. We are particularly interested in projects with more constrained release schedules, projects with greater separation between architects and developers, and projects that have been deemed 'unmaintainable'.

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.

## 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.

## References

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

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

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

[4] 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.

[5] 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.

[6] 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.