Code Convention Adherence in Research Data Infrastructure Software: An Exploratory Study

Michael Smit
School of Information Management
Dalhousie University
Halifax, Canada
Email: msmit@dal.ca

***Abstract*—Science is rapidly evolving, incorporating technology like autonomous vehicles, high-throughput scientific instruments, high-fidelity numerical models, and sensor networks, all generating data with increasing frequency, variety, and volume. Scientists committed to open science are interested in sharing this data, which requires research data infrastructure (RDI). The software underlying RDI is often created and/or deployed by people who have not received formal training in software engineering, or at organizations with primary mandates that do not include software development. Our understanding of software engineering as a field and practice does not universally translate to this software. As RDI software is pushed to handle larger data sets, and used to share data more widely, it is important to understand the maintainability, the resilience of the development community, and other indicators of long-term software project health. While there is a body of research on scientific software, and on free and open source software, it is not known if existing approaches to assessing these properties are effective for RDI software. In this exploratory study, we calculate one proxy measure for maintainability (code convention adherence) for a popular ocean data management system, and compare the results with four open source projects, and with the apparent experience of users as captured in public mailing lists and an issue tracker. The results advance our limited understanding of this type of software, and inform hypothesis generation and future research design.**

***Keywords*-research data management; open data; FAIR principles; ocean data management; research data infrastructure; technical debt**

I. Introduction

Two important trends in research data management are converging. The first is scientists collecting a growing variety and volume of data focused on their research problem, and seeking solutions to more effectively manage, store, query, search, and retrieve this data [1][2][3]. The second is a growing interest in sharing this data, driven by personal motivation and/or funder, journal, or institutional requirements [4][5][6]. The increase in data availability and the growth of data sharing is happening at different speeds and in different ways for various disciplines, which has resulted in a pantheon of software tools for depositing, disseminating, and accessing this data. We thus have a widely varied *research data infrastructure* (RDI, defined by the Research Data Alliance as "digital infrastructure organized to promote data sharing and consumption in support of research efforts" [7]).

RDI software includes general, discipline-agnostic data repository software, where development is driven by information and data specialists (librarians, archivists) at organizations with primary mandates that do not include software development, or by open-source communities. It also includes domain-specific "homegrown" tools developed by scientists to meet a particular need within their domain, sometimes within their organization. These developers are often also users; they are adding features as they or their colleagues identify the need [8]. As this software is pressed into service to share larger volumes of data more widely, it is important to understand how it will respond to increased use, development, and feature expansion. For commercial software or free and open source (FOSS) projects, there are various approaches to assessing mintainability, potential longevity, technical debt, community resilience, and many other indicators of long-term health. Our understanding of software engineering as a field and practice does not always translate outside of organizations with a strong software engineering focus [9], so it is not known if existing approaches to assessing these metrics are effective for RDI software. This software, and its relationship to other types of software, is discussed in more detail in Section II.

Any developer can create source code that can be understood and executed by a computer to provide desired features. Software engineering is about undertaking development as a systematic, managed, and measured approach to the entire software lifecycle, including operating and maintaining software once developed. At its most simple, one might say that software engineering is about putting effort into things the computer does not understand: building teams of collaborators, managing the work effort of distributed teams, creating design documents and other formal artifacts that are human-readable not necessarily computer readable, and putting effort into writing source code that makes it easier for humans to edit. The last example is about establishing and adhering to conventions which suggest commonly-accepted best practices for naming variables, documenting code, limiting the length of functions, and many other development decisions. While these actions are not recognized or valued by compilers, they make working with a team to develop and maintain software easier. The extent to which software adheres to code conventions may be a coarse measure of to what extent software engineering principles influenced the development of a software system. In fact, some have suggested that code convention adherence is related to how maintainable a software system is [10][11][12], as further described in Section II-C.

This paper motivates and describes an exploratory study to examine how code convention adherence in RDI software compares to adherence in other free and open source software (FOSS). This comparison provides insight into how comparable RDI software is to a class of software that is widely studied in literature on software maintenance, and can also serve as a coarse proxy measure of the maintainability of RDI software. We choose a single software package from the discipline of ocean observation for this exploratory study, and compare its code convention adherence to a baseline for FOSS adherence established in previous work [12][13]. We then review the mailing list and Github issues for this project to compare the results of the automated review to the documented experience of developers and users. We seek to answer three questions: RQ1. Is this software system less adherent to generally accepted code conventions? RQ2. If less adherent, does that have a noticeable impact on users or contributors? RQ3. Does the quantitative analysis provide sufficient insight to justify incorporating additional metrics? The methods are described in Section III.

The preliminary results (Section IV) provide insight into the nascent software seeking to meet user needs at the intersection of the explosion of Big Data and the push for open, FAIR research data. In short, it's not always pretty, but it seems to work incredibly well for the people using it. While at this stage the results are not generalizable to other RDI software or the broader field of research data management, they do indicate we should continue to investigate the extent to which existing literature might be relevant, and they will inform hypothesis generation and future research design (Section V).

II. Background and Related Work

A. RDI software as scientific software

We hypothesize RDI software has much in common with software used directly in computational science ("scientific software"); in fact, some definitions of scientific computing would notionally include the data infrastructure. Scientific software has been extensively studied through the lens of software engineering, particularly during the earliest spike in interest in scientific software. What development methods do they use? Agile, kind of [14]. Do they use version control? Yes, but not what we're used to [15]. What metrics do they value? Good science over good software [16]. What testing do they do? It varies; it is often quite limited relative to SE best practices for a variety of reasons [17], but there are examples of rigorous software testing [18][19]. Does the software work to the extent that it produces accurate results? Mostly, yes [19]. Is the development efficient? Not really [20]. Does it age well? Not especially; maintenance effort grows quickly [21], though it's not clear if it's worse than other software.

Generally scientific software works as intended, in part because the developers are the users, and as domain experts well-suited to detect errors in output. There is some evidence that scientists don't distinguish between their theory and the code that implements the theory [19]. Yet there are high-profile examples of scientific software resulting in errors. In one data-related example, a piece of scientific software was found to have swapped two columns of data, significantly changing the outcome of the analysis and resulting in modification or retraction of five papers [22].

RDI software is also different from scientific software, in potentially meaningful ways. We expect that the increased expectations around sharing data has led to RDI being used in ways not originally anticipated when it was first created. Growing expectations of users and data managers have led to the development of advanced query/sorting features, new versioning features, and visualization features. More importantly, RDI is now used to provide data to end users who are not deeply involved in the creation of that data, meaning the quality control offered by developers being the primary users may be impaired. The core functions of RDI (depositing, managing, searching, and retrieving data) are not core to science in the same way that the functions of scientific software typically are. The typical software model is that expert software developers implement requirements provided by the user. The scientific software model is that users *are* the developers. With RDI, and the growth of data sharing, the developers are neither expert software developers nor are they the primary users of all features. The implications of this on software quality are unknown.

While scientists acknowledge the challenge of managing data effectively in relation to scientific software [23], and the importance of ensuring data quality, recent reviews of software engineering in scientific software suggest there has been no examination of the quality of RDI software [17][24][25]. The RDI community has not prioritized software maintenance/quality. The World Data System of the International Science Council and the Data Archiving and Networked Services combined their efforts to assess and review data repositories several years ago, creating the CoreTrustSeal. This certification is issued to complying repositories as an indicator that this repository is suitable for data deposit [26]. Their 30-page guidance on the requirements to be certified mention software only briefly: "The repository functions on well-supported operating systems and other core infrastructural software and is using hardware and software technologies appropriate to the services it provides to its Designated Community" [26]. More specific questions ask about a "software inventory" and if "community-supported software" is in use, with any answer being accepted as long as it is properly justified. There is little consideration of the maintainability of the software.

B. RDI software as open source community projects

Where scientific software is often (though not always) unique to the lab that developed it, RDI software is often open source, with a community of contributors and public interaction between users and the developers. The health of open source software often depends on the health of that community, and several decades of research have informed metrics and analyses for assessing this health. RDI software often originates through a pattern similar to scientific software, and is released as open source software to encourage adoption. We hypothesize that as RDI software matures, it more closely resembles open source communities. We have anecdotally observed phenomena in RDI software development that resembles archetypes in the literature – the "hero" who has unique and extensive knowledge of the software [27]; the long tail of small-scale contributors [28]; the influence of the for-profit companies that contribute source code [29]; the adjustment to providing support to a user community with high expectations [30]; and more.

Major risks to open source projects include abandonment [31] and the extent to which knowledge is concentrated with only a few individuals (sometimes referred to morbidly as the bus factor) [32], and we anticipate methods for measuring these (e.g. [31][33]) will be usable on RDI software projects.

C. Code convention adherence & software maintainability

While software is often assessed on its ability to meet functional and user-facing non-functional requirements (e.g. performance), maintenance for software begins very early in the project. Seminal work in 1980 reported that maintenance uses about 50% of development time, and includes correcting faults (corrective), responding to external changes (adaptive), and improving features and documentation (perfective) [34]. (More recent estimates of relative effort vary, but the original study remains a useful approximation.) ANSI/IEEE standard 729 defines maintenance as "the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment" [35]. The ease with which software can be maintained (its *maintainability*) is a question of significant interest. Related concepts include software complexity, intentional decisions to take shortcuts to save time in the short term (technical debt), and for free and open source software (FOSS) the health of the community. While maintainability is difficult to measure, various proxy measures have been proposed.

The majority of metrics proposed for measuring maintainability focus on evaluating complexity. The Halstead complexity metric [36] and McCabe's cyclomatic metic [37] are two prominent complexity metrics. These metrics are combined with the number of lines of code to compute a maintainability index [38]. But even complex software can be more maintainable if it is understandable, and software can be made understandable using documentation or by writing code in a style that is easier to read and modify [39].

Software source code is a written language that codifies the design of the software, whether that design is formal or simply a programmer's vision. Some decisions make software brittle and hard to modify, like using "magic numbers" without explanation, or by using literal strings with the same value repeatedly but separately. Other decisions obscure the intent of the developer, like how to name variables or how long to make functions. This individual decision is sometimes referred to as a programming style, and individual preferences vary. The purpose of a code style convention or best practice is to improve the maintainability of source code and capture best practices [40], [41]: adopting a *lingua franca* for a development team. 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.

Source code conventions have co-evolved with programming languages, and using consistent style within a project is generally valued by development teams. Yet adherence takes effort, and there are times when that effort is not expended. For example, 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 [42]. To support developers, tools can be used to enforce these conventions (for example, FindBugs[1], Checkstyle[2], and Jtest[3]).

In previous work that sought to use code convention adherence as one proxy measure that impacts maintainability [12][13], we recognized that not all of these conventions are equally relevant as far as the code's readability, understandability and maintainability is concerned. 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 had associations with our group, but were not involved in the code convention adherence project. A total of 71 different coding conventions

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

were presented to each member of the panel independently. 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 scale 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).

For this paper, we focus on the set of conventions deemed important to the maintainability of most software; this list and the survey results are described in detail in a technical report [13]. Each item on this list is also part of the original code conventions documented by Sun [41], which are widely used as de facto conventions by software projects.

III. Methods

A. Software for exploratory study

ERDDAP "is a data server that gives you a simple, consistent way to download subsets of gridded and tabular scientific datasets in common file formats and make graphs and maps[4]" using the DAP protocol, and is widely used by NOAA and the ocean observation community[5]. Originally developed by a single developer in NOAA to meet local needs, it was released as an open source Java project with several contributors, bug tracking, and an active mailing list. It is an example of discipline-specific data repository software, with a small development community. It appears to be experiencing the transitions described in the introduction. The most recent release added features generally understood by software engineers to frequently introduce faults: threading, caching, more aggressive memory management to improve handling of Big Data, sophisticated string processing to produce date/time values, and advanced query/sorting features[6]. The primary developer is more frequently adding features requested by others. There has not been an apparent increase in the number of software errors.

Through other research projects, we are connected to the developer and worldwide user community of ERDDAP. After extensive assessment, we unequivocally recommended ERDDAP for use in an oceans-specific data repository in Canada [43]. This research is approached as committed users of ERDDAP and believers in its value, seeking to dispassionately examine markers of long-term maintainability of this essential tool.

[4]https://coastwatch.pfeg.noaa.gov/erddap/index.html
[5]https://github.com/IrishMarineInstitute/awesome-erddap
[6]https://coastwatch.pfeg.noaa.gov/erddap/download/changes.html

The primary developer of ERDDAP is Bob Simons, a NOAA employee. He discusses the use of code conventions in the documentation[7]: "I try to write very clean code. I write Java Doc comments. I write comments in the code. I chose variable names carefully. I follow the Java formatting guidelines. All of this is an effort to make the code more readable, for other programmers who want to understand and/or change it, and for me, because, in a year or two, I will have forgotten the details of how and why the code was written the way it was." When we assess adherence to the project's own style, we assume that "Java formatting guidelines" refers to the Sun Java coding conventions[8]. For one person to be primarily responsible for developing, documenting, and maintaining such an essential tool is remarkable, though not unheard of in FOSS projects.

ERDDAP has several important differences from the software previously assessed that require changes to the previous methodology [12].

- It is primarily maintained by a single developer who has written most of the code.
- Like many modern FOSS projects, it is hosted on GitHub; all of the tests in 2011 used SVN.
- The distribution does not include a build script; the recommended approach to building is to compile TestAll.java after verifying your classpath includes the files listed in the documentation[9]. Indeed, many releases of ERDDAP include defunct code with errors, with a comment early in the file indicating the file is no longer used[10]. Using addition / deletion / updates as in the previous paper won't be accurate as code may effectively be deleted but live on for many commits.
- Code is committed mostly at release time, with long periods of time in between commits. For example, the initial release candidate for version 2.0 was committed almost 7 months of no commits, and included 313 changed files with 73,827 additions and 33,543 deletions[11]. In total, there have been 46 commits over the 7 year history in the GitHub repository. This analysis will tell us less about the ongoing development of the project than the previous study, where commits were frequent and smaller.

B. Approach to analysis

We identified all commits up to the curent release (v2.02), and checked out each commit in sequence. We used the documented approach to build from source, and then identified every .java file included in the build. These were processed

[7]https://coastwatch.pfeg.noaa.gov/erddap/information.html#sustainable
[8]https://www.oracle.com/technetwork/java/codeconvtoc-136057.html
[9]https://coastwatch.pfeg.noaa.gov/erddap/download/setup.html#programmersGuide
[10]https://github.com/BobSimons/erddap/issues/23
[11]https://github.com/BobSimons/erddap/commit/0f527064baf4967640178e4ef87ee9d27f06c5d9

using the 2011 checkstyle configuration files for assessing code convention adherence (both important and project-specific), cloc[12] for counting lines of code, and lightly modified versions of the tools and visualizations used in 2011 to tie it all together [13]. (These tools were implemented in Perl and used gnuplot and various visualization libraries). The working directory was wiped between each checkout operation.

We then reviewed the types of violations flagged for the current release, and examined examples of each in the codebase.

Both sets of results were compared with the results from the previous study, with a particular focus on JFreeChart because of the small size of the development team and the fact that it provides a checkstyle configuration file for developers to adhere to, and Apache Derby because it is also data-focused and follows the Sun conventions without using checkstyle or a similar tool.

Finally, we reviewed all open and closed issues on Github, commit logs, and the history of the ERDDAP Google Group mailing list[13] (135 threads, 576 messages) to examine the lived experience of developers and users. This is in addition to a technical assessment of the functionality of the software previously completed [43].

IV. Results & Discussion

The methods for this paper included a tool-based quantitative analysis and an informal qualitative analysis, each described in the following two subsections. The final subsection discusses the answers to the research questions posed in the introduction.

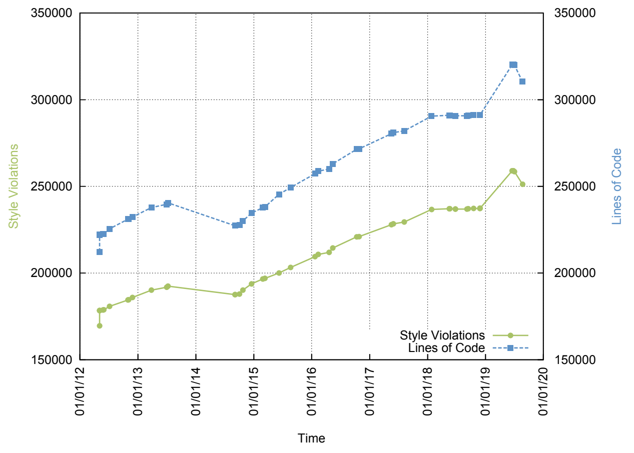
A. *Checkstyle results*

There are 129,378 lines of comments and 310,434 lines of Java code in the latest release, in approximately 500 class files. Checkstyle reports 103,537 deviations from our 'important' conventions, and 251,195 deviations from Sun conventions. The number of deviations grows linearly with the number of lines of code (LOC) over time ($\rho = .999$ for important conventions), which is similar to other projects we've examined that do not use tools to support code convention adherence (Figure 1 shows the change over time for self-imposed conventions).

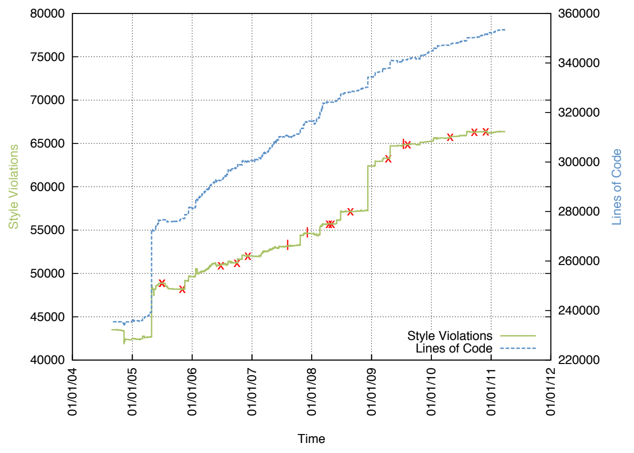
We next examined the ratio of deviations from code conventions to the size of the project (lines of code), over time. The maximum possible ratio is not fixed; types of deviations vary based on the code, and some deviations relate to comments. There is no objective measure of what the ratio of violations to source lines of code should be; however, relative comparisons can be made. The number of deviations per LOC in ERDDAP (.33) is notably higher than any of the FOSS we examined ($< .20$). There is a slight but

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

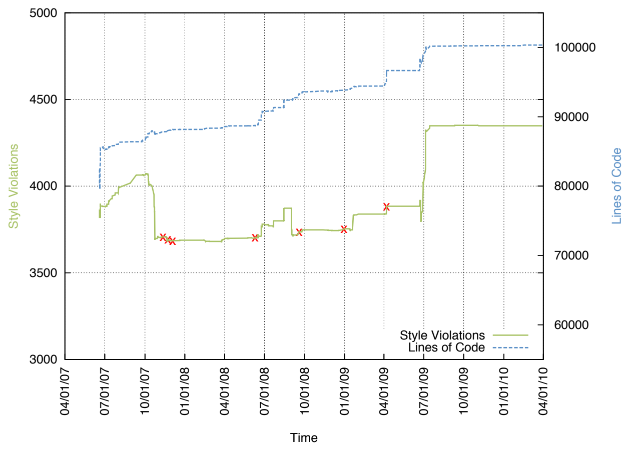
[13]https://groups.google.com/forum/#!forum/erddap



(a) ERDDAP



(b) Apache Derby



(c) JFreeChart

Figure 1: Violations of self-imposed code standards (left axis) and SLOC (right axis); release points shown by 'x' for the latter two. Note the y-axes do not start at 0.

consistent decrease over time from .36 to .33, as shown in Figure 2.

The type of deviation is an important consideration. It is difficult to compare the count of deviation types, as some deviations can occur for every variable declared (e.g. the Final Local Variable test) while others by definition can occur for only some of the classes (e.g. Class Fan Out Complexity). We have listed the deviations identified for the latest version of ERDDAP in Table I, along with their ordering when sorted by number of incidents. We then include the ordering for that issue on each of 4 other FOSS projects, and focus on the deviations that appear more or less prevalent than the comparator group (rank shown in bold).

The most common deviations across all projects included the Final Local Variable test, which looks for instances where a local variable is declared and assigned but not modified, and reports a deviation if it was not declared final. This is somewhat controversial. 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 maintainers 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 any of these projects.

Another common pair of deviations 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 must be changed in (at least) two places. (It should be noted that ERDDAP defines a `String2` class, for Strings requiring features not provided by the standard Java library. `String2` literals would not be detected by this test.) 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.

ERDDAP is notably better at providing Javadoc-compatible comments on public methods and classes, which is a very common deviation in the comparator projects. It is also makes meticulous use of whitespace: a top 10 deviation for one project, but only two issues in 300,000 lines of code for ERDDAP.

ERDDAP and Apache Derby both have a high occurrence of missing *Braces*. The compiler considers braces optional for single-line blocks, but best practices suggest that they be included anyway to make the block explicit. Users accustomed to this best practice may add a line following the existing single line intending for it to be part of (for example) an `if-else` block.

Unexpectedly high are *Illegal Catch* and *Illegal Throws*. The former are overly broad `catch` statements that catch `java.lang.Exception` or other superclasses instead of the specific exception they are trying to handle. The issue is while the handling code might handle `IOException`, the catch statement might catch other unanticipated exceptions, like `NullPointerException` or `OutOfMemoryError`. Alternatively, maintainers might add code that throws a different type of exception and not notice it is not handled. The short version of this convention is "don't catch something you aren't prepared to handle" . The latter are the inverse: constructors or methods that are defined as throwing an overly broad exception, like `java.lang.Throwable`. This makes handling the exception unlikely, as calling methods cannot predict what exceptions might be thrown. There is indication that some of these are deliberate, with the intent being to display the full stacktrace so they can contact the lead developer[14]. This works as long as the user is themselves a developer or ERDDAP administrator, but isn't an ideal user experience for other users.

Notable is the 23 Covariant Equals deviations, which was not detected at all in the other four projects. This is flagged because it is almost always a bug: it usually indicates times when a programmer intended to override `Object.equals(Object o)`, but instead of accepting an `Object` parameter, it accepts a covariant (e.g., itself, or `Comparable`). For example, `gov.noaa.pmel.util.Dimension2D` is a simple wrapper class "to encapsulate a width and a height" (Javadoc). It defines a method for testing the equality of two dimensions: `.equals(Dimension2D d)`, which properly compares the width and height fields to see if they are equal. However, this method does not override `Object.equals(Object o)`, which means there are two `.equals()` methods, one which does an appropriate equality test based on the values and one which will only return true when compared to the exact same `Object` in memory. As the checkstyle documentation says, "this kind of bug is not obvious because it looks correct, and in circumstances where the class is accessed through the references of the class type (rather than a supertype), it will work correctly. However, the first time it is used in a container, the behavior might be mysterious[15]."

In summary, while some convention deviations may not impact maintainability, there is evidence of generally accepted conventions not being adhered to, as well as some potentially dangerous violations. Some of these more dan-

[14]https://groups.google.com/forum/#!topic/erddap/aPu6wczCP54
[15]https://checkstyle.sourceforge.io/config_coding.html#CovariantEquals

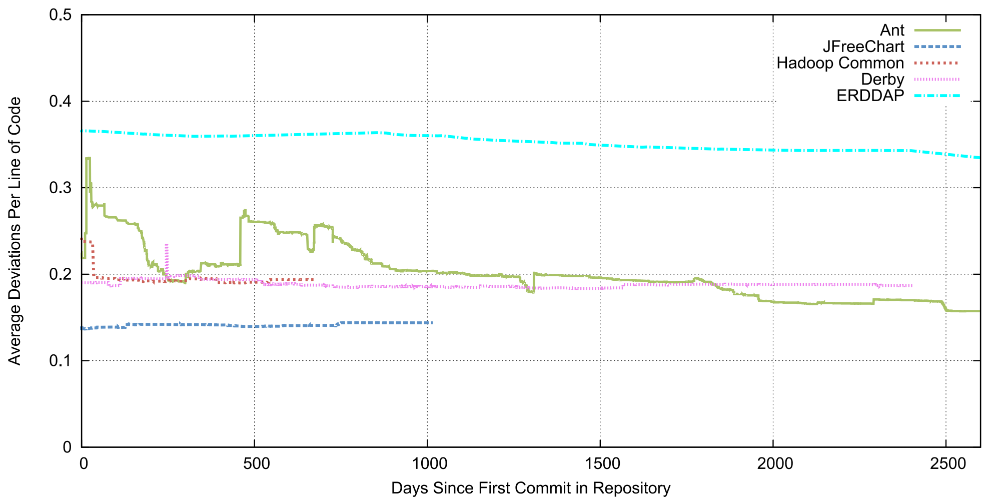


Figure 2: Code convention violations per line of code, based on 'important' code conventions.

gerous violations occur with higher frequency in ERDDAP than in the FOSS comparators.

B. *The experience of developers and users*

Developers and users are typically enthusiastically positive about ERDDAP in public-facing messages and meetings. The lead developer responds promptly and thoroughly, and there is robust discussion. Most of the messages are seeking support or requesting features, rather than reporting issues. ERDDAP is trusted to serve thousands of datasets globally, by leading ocean research organizations.

There are anecdotes of users encountering issues related to these code convention deviations. For example, a user reported an issue where an error message about a missing environment variable didn't match the name of the variable the code was looking for, and indicated this could be resolved by using a single String constant instead of multiple string literals[16]. There is also indication that some of the code convention deviations are intentional, for example displaying complete stacktraces to end users.

There is also evidence that ERDDAP is being pushed toward Big Data, based on the ripple effects seen in mailing lists: memory use issues[17], challenges with interacting with Amazon S3 storage[18], and interest in federating search across multiple (30+) instances of ERDDAP[19].

Finally, there are indications of a clear commitment from the community to further develop ERDDAP, including a code sprint led by the lead developer to introduce more developers to the source code[20].

[16]https://github.com/BobSimons/erddap/issues/1
[17]https://groups.google.com/forum/#!topic/erddap/OaX7JjV18pg
[18]https://groups.google.com/forum/#!topic/erddap/_GZK5VEKeVc
[19]https://github.com/IrishMarineInstitute/search-erddaps
[20]https://groups.google.com/forum/#!topic/erddap/xAHPX-Q2YaE

C. *Addressing the research questions*

RQ1. Is this software system less adherent to generally accepted code conventions? Yes, it is less adherent than the four comparator projects, both to the code conventions identified as important and to the ones identified by the lead developer as their standard practice. There are limits to this comparison, and larger comparator groups should be sought. The code convention deviations mimic the pattern of small development teams that don't use formal style checkers.

RQ2. If less adherent, does that have a noticeable impact on users or contributors? We did find anecdotal evidence that users encountered bugs that would not exist if code conventions were followed, and some of the deviations were of a type that suggests latent bugs exist. However, users remain enthusiastic about the project and it is deployed widely to an engaged community. It is difficult to assess how contributors react to these conventions, as the vast majority of contributions are from the original developer.

RQ3. Does the quantitative analysis provide sufficient insight to justify incorporating additional metrics? There are indications that these metrics are useful, but any further work will need more extensive grounding in the lived experience of developers and users, including interviews. Additional RDI software should be examined, a broader array of metrics should be explored, and more information should be gathered to help understand why this project is different from the comparator group.

V. Conclusion

RDI software is critical to supporting Big Data for open science. This exploratory study suggests that there is merit in examining code quality metrics, as long as these metrics are triangulated using the lived experience of developers and users. The initial indication is that RDI software is more

Table I: The types of code convention deviations detected in ERDDAP, along with their relative frequency rank compared to four other FOSS projects. Notable deviations are in bold.

Checkstyle Test	**Count**	**Rank**	**Ant**	**JFC**	**Hadoop**	**Derby**
Final Local Variable	20928	1	1	1	2	2
Magic Number	19255	2	3	2	3	4
Multiple String Literals	17436	3	4	4	4	8
Need Braces	13499	4	9	13	5	3
Javadoc	8263	**5**	2	3	1	1
Visibility Modifier	3638	6	5	8	6	6
Naming	2307	7	6	7	7	5
Illegal Catch	1446	**8**	8	14	12	12
One Statement Per Line	1213	**9**	20	20	16	25
Illegal Throws	991	**10**	23	–	25	27
Multiple Variable Declarations	915	11	13	12	17	19
Parameter Assignment	860	12	10	10	8	10
Cyclomatic Complexity	773	**13**	7	5	9	9
Avoid Star Import	470	14	11	22	10	16
Inner Assignment	291	15	14	16	11	14
Parameter Number	179	16	19	9	20	13
Avoid Nested Blocks	103	17	18	26	13	15
Class Fan Out Complexity	70	18	15	11	18	17
Hide Utility Class Constructor	44	19	17	19	14	20
Equals Hash Code	40	20	26	6	29	28
Uncommented Main	36	21	21	17	23	26
Method Count	27	22	22	18	26	21
Covariant Equals	23	**23**	–	–	–	–
Modified Control Variable	14	24	16	21	15	24
Simplify Boolean Expression	14	25	28	24	24	18
Mutable Exception	13	26	24	–	22	22
Redundant Import	11	27	25	25	21	11
Simplify Boolean Return	5	28	27	23	28	23
Default Comes Last	3	29	–	–	30	29
Whitespace	2	**30**	12	15	19	7

vulnerable to maintenance issues than the FOSS comparator group.

While this exploratory study included only one example of RDI software, it is an interesting case. ERDDAP is at an important stage in its evolution as it is called upon to do more. For example, Canada is establishing a National Data Services Framework, defined as part of the digital research infrastructure supporting the research enterprise [44]. It includes national-, institution-, and discipline-specific repositories. One such discipline-specific repository is the Canadian Integrated Ocean Observing System, which is built using ERDDAP [45], and is also connected to the global ocean observing system (GOOS). As this software plays an increasing role in data infrastructure, its maintainability becomes more important.

Further work is required to propose specific research objectives. Software quality is a complex construct; a first step will be a thorough review of using static analysis to assess maintainability (including technical debt); sustainability; and code smells. Starting points include the Linux Foundation-sponsored CHAOSS project, which develops metrics and tools to assess the health of open source communities coupled with a systematic review focused on specific publication venues which have a strong practical lens, like the IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM). Once candidate measurements are identified, they should be applied to a variety of case studies and a control group.

To triangulate these results with the lived experience of user communities and developers, we suggest methods based on the ethnographic methods employed by Easterbrook and Johns [19] when assessing scientific software. Theirs was some of the earliest work to acknowledge that atypical development processes could still yield high-quality results. Drawing on their approach, we suggest semi-structured interviews with users and developers, and travel to observe meetings of the developer group and various user groups, coupled with rigorous content analysis of project documentation and digital records, including the mailing lists, commit comments, and issue trackers.

Good science needs good data; there is both the desire and mandate to improve data management and sharing in research. While RDI is somewhat fractured at present, the long-term vision is a global network that links research data, big and small, across disciplines and borders. FAIR data is only achievable if we have a research data infrastructure that is maintainable, reliable, and useable, which requires that we have the tools and metrics we need to assess RDI software. Better metrics will help users make good software adoption decisions, help developers improve RDI software, and build confidence in research data repositories.

Acknowledgment

This research is supported in part by the Ocean Frontier Institute and was enabled in part by support provided by ACENET and Compute Canada. The author is grateful to the co-authors of the 2011 paper leveraged extensively by this project: Barry Gergel, H. James Hoover, and Eleni Stroulia. Thank you to Bob Simons and NOOA for creating such a useful open source tool.

References

[1] M. J. Costello and E. V. Berghe, "'Ocean biodiversity informatics': a new era in marine biology research and management," *Marine Ecology Progress Series*, 2006.

[2] D. Akmon, A. Zimmerman, M. Daniels, and M. Hedstrom, "The application of archival concepts to a data-intensive environment: working with scientists to understand data management and preservation needs," *Archival Science*, vol. 11, no. 3, pp. 329–348, November 2011.

[3] A. H. Poole, "How has your science data grown? digital curation and the human factor: a critical literature review," *Archival Science*, vol. 15, no. 2, pp. 101–139, 2015.

[4] K. S. Baker, S. J. Jackson, and J. R. Wanetick, "Strategies supporting heterogeneous data and interdisciplinary collaboration: Towards an ocean informatics environment," in *Proceedings of the 38th Annual Hawaii International Conference on System Sciences*. p. 219b–219b, 2005.

[5] J. Vertesi and P. Dourish, "The value of data: considering the context of production in data economies," in *Proceedings of the Conference on Computer supported cooperative work*, 2011, pp. 533–542.

[6] Y. Shorish, "Data information literacy and undergraduates: A critical competency," *College and Undergraduate Libraries*, vol. 22, no. 1, pp. 97–106, 2015.

[7] Research Data Alliance DFT working group, "Data foundation and terminology vocabulary," https://smw-rda.esc.rzg.mpg.de/exports/tedt30.pdf, 2018.

[8] V. R. Basili, J. C. Carver, D. Cruzes, L. M. Hochstein, J. K. Hollingsworth, F. Shull, and M. V. Zelkowitz, "Understanding the high-performance-computing community: A software engineer's perspective," *IEEE software*, vol. 25, no. 4, p. 29, 2008.

[9] J. Segal, "Scientists and software engineers: a tale of two cultures," in *Proceedings of the 20th Annual Meeting of the Psychology of Programming Interest Group*, 2008.

[10] Y. Guo, "Measuring and monitoring technical debt," Ph.D. dissertation, University of Maryland, Baltimore County, 2016.

[11] Z. Wang and J. Hahn, "The effects of programming style on open source collaboration," in *Proceedings of the International Conference on Information Systems (ICIS)*, 2017.

[12] M. Smit, B. Gergel, H. Hoover, and E. Stroulia, "Code convention adherence in evolving software," in *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2011, pp. 504–507.

[13] M. Smit, B. Gergel, H. J. Hoover, and E. Stroulia, "Maintainability and source code conventions: An analysis of open source projects," University of Alberta, Tech. Rep., 2011.

[14] J. Carver, R. Kendall, S. Squires, and D. Post, "Software development environments for scientific and engineering software: a series of case studies," in *Proceedings of the 29th International Conference on Software Engineering*, 2007, pp. 550–559.

[15] D. Matthews, G. Wilson, and S. Easterbrook, "Configuration management for large-scale scientific computing at the uk met office," *Computing in Science & Engineering*, vol. 10, no. 6, p. 56, 2008.

[16] R. Kendall, J. C. Carver, D. Fisher, D. Henderson, A. Mark, D. Post, C. E. Rhoades, and S. Squires, "Development of a weather forecasting code: A case study," *IEEE software*, vol. 25, no. 4, pp. 59–65, 2008.

[17] D. Heaton and J. C. Carver, "Claims about the use of software engineering practices in science: A systematic literature review," *Information and Software Technology*, vol. 67, pp. 207 – 219, 2015.

[18] F. Shull, "Assuring the future? a look at validating climate model software," *IEEE software*, vol. 28, no. 6, pp. 4–8, 2011.

[19] S. M. Easterbrook and T. C. Johns, "Engineering the software for understanding climate change," *Computing in Science & Engineering*, vol. 11, no. 6, pp. 65–74, 2009.

[20] S. Faulk, E. Loh, M. L. Van De Vanter, S. Squires, and L. G. Votta, "Scientific computing's productivity gridlock: How software engineering can help," *Computing in science & engineering*, vol. 11, no. 6, pp. 30–39, 2009.

[21] R. Sanders and D. Kelly, "Dealing with risk in scientific software development," *IEEE software*, vol. 25, no. 4, pp. 21–28, 2008.

[22] G. Miller, "A scientist's nightmare: Software problem leads to five retractions," *Science*, vol. 314, no. 5807, pp. 1856–1857, 2006.

[23] C. A. Mattmann, D. J. Crichton, N. Medvidovic, and S. Hughes, "A software architecture-based framework for highly distributed and data intensive scientific applications," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 721–730.

[24] T. Storer, "Bridging the chasm: A survey of software engineering practice in scientific programming," *ACM Comput. Surv.*, vol. 50, no. 4, pp. 47:1–47:32, Aug. 2017.

[25] U. Kanewala and J. M. Bieman, "Testing scientific software: A systematic literature review," *Information and Software Technology*, vol. 56, no. 10, pp. 1219 – 1232, 2014.

[26] CoreTrustSeal, "Core trustworthy data repositories extended guidance," https://www.coretrustseal.org/wp-content/uploads/2017/01/20180629-CTS-Extended-

Guidance-v1.1.pdf, June 2018.

[27] F. Ricca and A. Marchetto, "Are heroes common in floss projects?" in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2010, p. 55.

[28] G. Pinto, I. Steinmacher, and M. A. Gerosa, "More common than you think: An in-depth study of casual contributors," in *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2016, pp. 112–123.

[29] B. J. Birkinbine, "Conflict in the commons: Towards a political economy of corporate involvement in free and open source software," *The Political Economy of Communication*, vol. 2, no. 2, 2015.

[30] R. P. Bagozzi and U. M. Dholakia, "Open source software user communities: A study of participation in linux user groups," *Management science*, vol. 52, no. 7, pp. 1099–1115, 2006.

[31] C. M. Schweik, "Sustainability in open source software commons: Lessons learned from an empirical study of sourceforge projects," *Technology Innovation Management Review*, vol. 3, no. 1, 2013.

[32] V. Cosentino, J. L. C. Izquierdo, and J. Cabot, "Assessing the bus factor of git repositories," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 499–503.

[33] G. A. Avelino, E. Constantinou, M. T. Valente, and A. Serebrenik, "An empirical investigation of the abandonment and survival of open source projects," *Empirical Software Engineering and Measurement*, 2019.

[34] B. P. Lientz and E. B. Swanson, *Software maintenance management*. Addison-Wesley Longman Publishing Co., Inc., 1980.

[35] N. F. Schneidewind, "The state of software maintenance," *IEEE Transactions on Software Engineering*, vol. 13, no. 3, pp. 303–310, 1987.

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

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

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

[39] D. Posnett, A. Hindle, and P. Devanbu, "A simpler model of software readability," in *Proceedings of the 8th working conference on mining software repositories*. ACM, 2011, pp. 73–82.

[40] P. W. Oman and C. R. Cook, "A taxonomy for programming style," in *Proceedings of the 1990 ACM annual conference on Cooperation*. ACM, 1990, pp. 244–250.

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

[42] 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.

[43] M. Smit, R. Kelly, S. Fitzsimmons, S. Bruce, C. Bulger, B. Covey, R. Davis, R. Gosse, D. Owens, and B. Pirenne, "Canadian integrated ocean observing system: Cyberinfrastructure investigative evaluation," Fisheries and Oceans Canada, 2017.

[44] D. Baker, E. Barsky, J. Burpee, M. Leggott, J. Moon, M. Sinatra, B. Spencer, and L. Gerlitz, "Canadian National Data Services Framework: Discussion document," https://doi.org/10.5281/zenodo.2536558, 2019.

[45] A. Stewart, B. DeYoung, M. Smit, K. Donaldson, A. Reedman, A. Bastien, B. Carter, R. J. Kelly, E. Peterson, B. Pirenne *et al.*, "The development of a Canadian integrated ocean observing system (CIOOS))," *Frontiers in Marine Science*, vol. 6, pp. 431–440, 2019.