How Should We Measure the Relationship Between Code Quality and Software Sustainability?

Aseel Aldabjan
School of Computer Science
University of Manchester, UK

Robert Haines
School of Computer Science
University of Manchester, UK

Caroline Jay
School of Computer Science
University of Manchester, UK
caroline.jay@manchester.ac.uk

***Abstract*—Software sustainability has been proposed as a non-functional requirement of a codebase. The aim of sustainable research software development is to produce reliable code that supports reproducible results, and can be reused in future projects. At present, research software is often not developed in a sustainable manner, partly due to the funding environment within which it exists, but also because there is no concrete metric with which to measure software sustainability, nor any concrete guidance on how to achieve it. We propose that empirical studies determining the relationship between measurable aspects of a project, and its active life—a period we define using a metric of *software sustainment*—are a strong means of understanding how requirements encapsulating software sustainability should ultimately be defined. Here, we report the results of a sustainment analysis of projects in GitHub, and describe the opportunities and challenges of understanding the relationship between sustainment and code quality.**

I. Introduction

Sustainable research software is defined as software that can be reused, in whole or in part, in future projects[1]. It is a truism that good coding practices will lead to more sustainable software, a view which is supported by research software engineers [1], and it has been proposed that sustainability should be considered as a requirement of a project [2, 3]. However, there is currently no concrete definition of software sustainability, nor any concrete guidance on how to achieve it [4]. This research aims to contribute to an understanding of what makes software sustainable, by measuring the relationship between characteristics of a project's codebase and the project's active life, which we term *software sustainment*.

We collect data for this study from GitHub, selecting a subset of repositories based on their start date and the language they are written in. As GitHub repositories include a number of software engineering artefacts—such as an issue tracker, documentation wiki, web pages and collaboration data—we consider a repository as a proxy for a software project[2].

In Section II we define our software sustainment metric, and in Section III we report on the distribution of open source Java projects according to this metric. Section IV describes the metrics we will use to measure code quality, and finally Section V discusses the challenges of linking code quality to sustainment, to ultimately determine the characteristics of software that are key to fulfilling the requirement of sustainability.

[1]http://software.ac.uk/

[2]Data and analysis code are available here: https://github.com/hainesr/sustainment-analysis

II. A metric for software sustainment

We take an empirical approach to understanding what makes a project sustainable, by examining how aspects of a project—in this case its code quality—vary as a function of its sustainment, or active life. Our definition of software sustainment is the time period from the initial creation of the software in a repository—the first commit—through to the last commit in the original repository (see equation 1):

$$S = t_{last\_commit} - t_{initial\_commit} \tag{1}$$

where S is our software sustainment metric, measured in days. This measured difference reflects the period over which the project is actively maintained or developed.

We calculate S for the default branch of the repository, as indicated in the meta-data we mine from GitHub, as we recognise that not all repositories use the 'master' branch as their default. We are only considering the default branch of the original project when calculating our sustainment metric as simply picking the most recently updated branch, or fork, has a high chance of containing incomplete, untested and non-working versions of the code. Nevertheless, there is an argument that if the code lives on in subsequent forks it has been sustained, even if the original project has not, so we will consider forks—and how they relate to the sustainability of both the project and the code—in a future study.

III. The sustainment of Java projects in GitHub

Projects were mined from GitHub according to the following criteria: they were created between 1st January and 31st December 2009; they had at least one commit; the first commit occurred after 1st January 2009; they were written in Java. Projects were retrieved on 26th July 2016, so S was calculated for each project at that point in time.

Figure 1 shows the distribution of projects as a function of S, in days. Of 3113 projects in total, 22% (682) had an S value of 0, and 35% (1076) had an S value < 7, indicating that over a third of the projects were sustained for only a week. A cursory inspection reveals some of these projects to be quite large, so it is likely that in these cases the development period was longer than the calculated sustainment metric, and that the project was only put into Git version control some time after its real start date. After the steep drop off at around seven days, the curve gradually flattens over time.

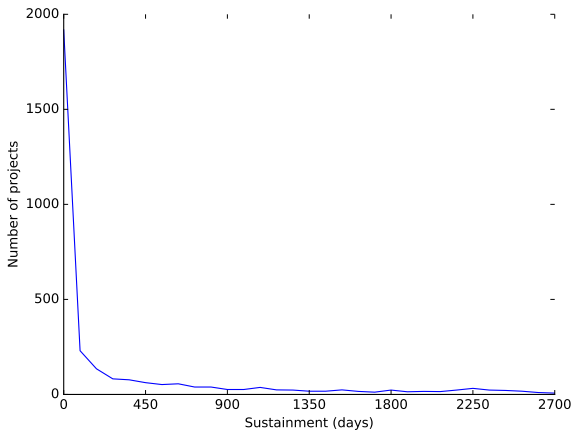

Fig. 1. Projects in GitHub as a function of their software sustainment in days.

IV. Code quality metrics

We hypothesize that the following static analytic metrics [5] are related to sustainability:

Lines of Code (LOC) is an indication of class size, where a higher value means longer and potentially more complex code. It is advisable to treat this metric in relative, rather than absolute terms, as lines of code may vary with programming language, or the individual style of a programmer.

Number of Local Methods (NOM), an indicator of interface complexity, measures the number of methods locally declared in a class. As the interface grows, the class becomes more complex, and more difficult to test. The optimum value for this metric is considered to be between 3 and 7. If there are fewer than 3, the class might simply be a data holder; if there are more than 7, the class might be in need of decomposition.

Depth of Inheritance Tree (DIT) calculates the complexity of a software entity based on the distance between a node and its root down the inheritance tree. As the code goes down the inheritance tree, testing becomes more difficult as the control flow becomes more complicated. A value between 0 and 4 is generally considered to indicate an adequate balance between complexity and the use of inheritance.

Coupling Between Objects (CBO) calculates the complexity of a class through its dependencies: a class is considered well designed when it is loosely coupled. Classes with a large number of dependencies are more difficult to maintain and test. The reusability of classes is limited by high levels of coupling because if a class depends on other classes, it is difficult to reuse it in another system. A value of CBO greater than 4 is generally considered undesirable because it indicates a high number of dependencies.

Improvement of Lack of Cohesion in Methods (ILCOM) provides a measure of class cohesion, by calculating the number of connected components in a class. High cohesion is a desirable characteristic within a class in object oriented languages, as it is usually harder to test classes that do not have cohesion between their components. A value of zero in the ILCOM metric indicates a lack of methods in the class, while a value of one represents a high level of cohesion. A value greater than one indicates cohesion is low, and the class may benefit from being divided into separate classes.

Lack of Documentation (LOD) was chosen as an interesting metric that considers comments in the code, with at least one comment per method and one per class as a minimum target. Comments often make the purpose of methods and classes clearer, increasing maintainability and facilitating the reuse of the code. Comments in Java code can also be used to automatically build API documentation for a project, so one might expect well maintained code to include at least one comment per method and per class for this purpose. A caveat is that the content of the comments is not considered.

V. Linking code quality to sustainability

We have proposed a simple metric with which to measure sustainment, and suggested a static analytic approach to quantifying code quality. Although these measures provide values that can be compared quantitatively, there remain considerable challenges in determining the relationships between them:

- *Refining the sustainment metric.* At present S only accounts for the time that activity on the project occurs in GitHub. Should we filter projects further, to ensure S is a true representation of the lifetime of the project?
- *Reconciling units of assessment.* How should we compare class-level software quality metrics with a project-level sustainment metric? For example, should we use median/mean values as input to the analysis, or look at the proportion of classes that meet a certain criteria?
- *Determining the appropriate point for assessment.* Is the final/current state of the code enough to draw conclusions? If not, should we combine metrics from various points in the project's history or monitor changes? How do we select those points in the project history?
- *Determining the appropriate statistical procedures for assessment.* Several of the code quality metrics are non-linear in terms of their optimal values, so a simple correlation may not be the best way to assess their relationship with sustainability. What is the best approach to take in these cases?

References

[1] M. R. de Souza et al., "Defining Sustainability through Developers' Eyes: Recommendations from an Interview Study," in *WSSSPE 2*, 2014.

[2] C. Venters et al., "The blind men and the elephant: Towards an empirical evaluation framework for software sustainability," *JORS*, vol. 2, no. 1, 2014.

[3] R. Chitchyan et al., "Sustainability design in requirements engineering," in *ICSE 2016*, 2016.

[4] C. Venters et al., "Software sustainability: The modern Tower of Babel." in *RE4SuSy*, 2014.

[5] R. Lincke and W. Löwe, "Compendium of Software Quality Standards and Metrics," 2005. [Online]. Available: http://www.arisa.se/compendium/