Defining Sustainability through Developers' Eyes: Recommendations from an Interview Study

Mário Rosado de Souza*, Robert Haines & Caroline Jay
School of Computer Science, University of Manchester, UK

July 21, 2014

Abstract

Defining 'software sustainability' in the context of research, and determining how best to support it, remains a considerable challenge. The research reported here approaches the question from a new direction by asking research software developers to provide their own experience of and opinions about sustainability. Two main themes emerged from the study: *intrinsic sustainability* – concerned with aspects of the software itself – and *extrinsic sustainability* – concerned with the environment in which it is developed. A key finding is that developers consider sustainability is less at the level of the research project that funds it, and more at the level of the software artefact itself, and ensuring that the functionality it represents can be re-used in some capacity in the future. We propose recommendations to improve sustainability, focusing in particular on community building, improving software quality and increasing motivation.

1 Introduction

Research software engineering projects are conducted in unusual development environments, particularly in terms of the way that they are funded: it is a period of time, rather than a set of features, that is resourced; software is often bespoke, and therefore constructed from scratch; and whilst there is an increasing expectation that it will continue to function after initial funding has ceased, it is difficult to define precisely how this will occur.

According to Raturi *et al.* (2014) 'Work at the intersection of information technology and sustainability often appears in the form of the "greening" of IT itself. This area, often called "Green IT", is "the study and practice of designing, manufacturing, using, and disposing of computers, servers, and associated subsystems ...efficiently and effectively with minimal or no impact on the environment". However, the software subsystem is often neglected.' [3]

Due to the experimental nature of research, much of the software produced is discarded, however there is a drive to ensure the software developed during this costly process has some additional value that may extend its life, by making it *sustainable*. But what exactly does this term mean to developers, and what processes are required to make it a reality?

The Software Sustainability Institute (SSI) defines sustainable software as follows: 'software you use today will be available – and continue to be improved and supported – in the future'[1]. We can therefore consider sustainable software as that which has longevity. However, 'there still is no concrete guidance for the different aspects of sustainability that are observable from the point of view of software engineering' [2].

This work aims to build a better understanding of what sustainability means in the context of research software engineering by interviewing developers and examining the extent to which they share a common understanding of sustainability. The results demonstrate that whilst there is some consensus as to the general meaning of the term, there are a variety of opinions about the best way of achieving it. By pooling and analysing the the interview data, we are able to provide a broad and comprehensive set of recommendations for how to improve the sustainability of software, based solely on the views of developers.

2 Study

Nine research software engineers (one female, eight male) from a large UK University were recruited for the study through purposive sampling. The participants work on a variety of projects within the same research group and have between 18 months and 20 years of software engineering experience. The semi-structured interviews, which were conducted either face-to-face or via video-conferencing, asked the following (with probing questions in italics):

1. From your point of view, what is sustainability in terms of software?
   What are the attributes or features of the software that lead you to believe that it is sustainable?

2. Regarding the software you've developed: was sustainability a consideration?
   If yes, at what point in time did it become a consideration? If no, why not?

3. Have you worked on any projects that were not sustainable?
   Were there any consequences of it not being sustainable?

The mean interview time was ten minutes and 12 seconds. The interviews were recorded, transcribed, and uploaded to the qualitative data analysis software, Dedoose

*mariorosadu@gmail.com

[1] www.software.ac.uk/about

4.12[2]. Transcripts were thematically analysed in an open coding fashion following established analysis methods: (1) familiarisation with the data; (2) generating the initial codes; (3) searching for themes; and (4) iteratively reviewing themes.

3 Results

The majority of the participants' definitions of sustainability approached that of the SSI, in the sense that software would continue to be both maintained and usable: 'I think it's keeping it well packaged and maintainable, documented.' (P2). '...it's sustainable if somebody else can pick it up...'(P1). 'sustainable software is software that someone can use now and can use going forward in the future' (P4).

Four participants said they had considered it at the beginning of a project, and the remaining five said it was considered after some time with that time ranging from two months to two years. 'I don't think it was a consideration at the start, I think at the start it was more about getting things done, getting things ready, so yeah it's more of a thing that's come about as the project has come along.' (P3). 'I've worked here for quite a few years so I'm used to this whole funding cycle and, you know you're paid for three years and then after two and a half years. Things start getting a bit hairy and you're hoping for more funding, so you know, with that in mind, we try and keep the software sustainable' (P2). The majority of participants (seven) had worked on software that was not sustainable for various reasons.

The precise way that developers considered sustainability varied. Whilst P4 and P8 viewed software as sustainable if it continued to be used by, or work for, the initial user group (the *project* is sustainable), other developers took a broader view, considering the possibility that it may be used in a different capacity by other developers, and potentially another user group (the *software artefact* is sustainable). In all cases the developers agreed that the codebase must be actively maintained. The results suggest that to achieve this, it is necessary to break the concept down into two forms: *Intrinsic Sustainability* and *Extrinsic Sustainability*.

3.1 Intrinsic Sustainability

This theme categorises characteristics of the software artefact itself that were perceived to be important to sustainability. These characteristics are listed in categories below, but it should be noted that they are often interrelated:

Documented (8 participants)

Participants largely agree that the code must be well documented. For sustainability to be possible: '...there are some additional steps that you have to do, like you make sure you have documentation, you make sure that the source code is in one place and things like that' (P8). 'It needs to be well-documented' (P5).

Tested and testable (5 participants)

Several of the developers felt that including tests is important. 'It's a lot of test automation and continuous integration testing, and I think that helps a lot with keeping it sustainable' (P2). 'Software tests as well. Yes, absolutely' (P7).

Easy to read and understand (5 participants)

Developers spend a lot of time refactoring and making their code readable, hoping this will make it more sustainable in the future. There was a general belief that if code is easy to read it will be more sustainable, because it will be more straightforward for someone else to pick up:

'...if he finds my code, and found that the effort of learning to use my code is going to be more difficult than the actual benefit it gave him, he'd probably throw away and write his own stuff' (P1).

Modular (2 participants)

Modularity is a very powerful feature as it reduces the complexity of the software, and makes it easier to reuse: '...so obviously you need to write nice modular code. Well I think that's a given in any kind of software development.' (P3). 'It turned out that the software was impossible for anyone to actually deploy in full, and it would only work if all the pieces were deployed. Funny, that didn't work.' (P4).

Uses third party libraries (2 participants)

Two developers made it clear that reinventing the wheel should be avoided, particularly when support is often good for libraries that have a large user base. It's important to '[use] technologies that people generally understand, reusing as much as you can, so don't write your own things, [when] there's good solutions already.' (P6).

Useful (2 participants)

If the software is fulfilling its purpose in an effective way, people will be motivated to sustain it. '...it's coupled to the software doing something useful, which either there isn't an alternative for, or that it is much better in its niche than the alternatives.' (P4). Usefulness could also be thought of at a broader level. If the code itself isn't quite right, 'they think, "OK, I will take the idea, but I will write my own stuff" ' (P3).

Scalable (1 participant)

Making code scalable was thought to help future-proof it. This ensures '...it's also going to be usable long term, 'cause if it's just the simple cases, people go, "yeah that's a really nice idea", and then as soon as they start using it in anger, a lot blows up because it doesn't scale or anything like that.' (P1).

3.2 Extrinsic Sustainability

This second theme considers the environment in which the software is developed and/or used, as opposed to the software artefact itself. The factors can be separated into the following broad categories, which are again interrelated:

Openly available (6 participants)

If research software is for use in a distributed project then sharing it in an open repository during the initial development stages is of great value. This is also a key factor for sustainability after the project end, as it increases the chance

[2] www.dedoose.com

it will be found and re-activated or reused. 'Usually I would look online in a repository for libraries and I would see when it was last updated... if it's in version control then it's a good start – it's in GitHub or [a] social coding website – and that's usually a sign that it's at least accessible by everyone and it's open and it's not going to be [a] closed book.' (P9).

Shared/co-owned (3 participants)

If the software is developed by a team, this increases the chances of it remaining active. '[It's important] that there is some community around it. You need to have more than one person involved, right? If it's a one man project and that guy is hit by a bus or just decided to do something else, work at google or something, then it just dies. And then by that point even if it's all open source then it's a bit too late to get involved because you don't know the inside workings and anything like that.' (P8).

Resourced (6 participants)

This is one of the aspects that developers were most concerned about. There needs to be a motivation to keep software active. Employment is the most common. '...a lot of our projects are coming to an end and we need to make a plan for them to be maintained in the future.' (P3). 'Somebody writes it and then when their funding runs out they just kind of abandon it and if someone else finds it, fine. And then it's a take it as you want it kind of thing. You see it in a lot of research, I mean the [removed for anonymity] stuff I did – completely just gone. The minute I left it, still sitting on GitHub but no one even looked at it. So that's it.' (P1).

Actively maintained (6 participants)

Developers are wary of software that isn't in current usage, due to the potential for out-of-date dependencies and modules that don't work any more because the platform has evolved. Support is less likely to be available. 'Physically the software lies there... you find software to do something, [you think] OK that looks good, and then you look – last updated three years ago. Most people won't touch it.' (P1). 'So it's about having this kind of momentum to the project, so that it keeps moving. That you have further development, even if you have maintenance mode – that is, not many new buttons being added – but at least there is someone [keeping] it alive. That could be out of self interest, because there is another organisation that depends on it, could be because they like having it as a hobby' (P2).

Independence from infrastructure (1 participant)

Sustainability can be related to where the software runs; if the infrastructure is not maintained, is the software capable of running outside that environment? '[Removed for anonymity] did most of that, and it's one of these things that will probably stay alive for as long as the server that it's on stays alive, and if that server crashes they will probably not bother rebuilding it into another machine.' (P1).

Supported (4 participants)

Sustainable software usually has some sort of user facing support from the team who is developing or maintaining it, which is helpful to both external developers and end-users. This is directly related to the project being active. 'So this tends to mean things like e-support, or automated tools of various kinds.' (P4).

4 Discussion

Evidence from the developers in this study indicates that the SSI's view that sustainable software 'will be available – and continue to be improved and supported – in the future' is, to a large extent, understood by developers, and considered to be meaningful. It is not clear that the working definition used by most developers is an exact match, however. To end users, the SSI's definition essentially means a software application that they can continue to use, but most developers considered sustainability to be a much broader concept. In particular, there is a significant focus for many developers on trying to ensure some aspect of the code itself is usable in the future, regardless of whether that use occurs in the same application, or contributes to a different one.

Although there is no concrete guidance on how to achieve sustainability in terms of software engineering [3], or indeed, not a concrete definition of research software sustainability itself [5], many of the factors that developers consider to be important for sustaining research software (particularly those relating to quality) are also important for both sustaining software products [4] and successful FLOSS development [1]. We therefore suggest that from the perspective of research software, a broad view is helpful. In simple terms sustainability can be considered at the level of a software product delivered by a particular project. In the absence of infinite resources, however, projects – and the software they produce – are going to remain of a fixed term nature. In this case, the route to sustainability therefore becomes reuse of the software, or aspects of it, in future projects.

5 Recommendations

Based on this broad understanding of sustainability, we can articulate developer-defined recommendations for improving it as follows:

Follow good development practices

If software is to be sustained over the long-term, it has to be learnable, readable and usable by more than one person – and preferably by anyone. Developers were clear that producing tested/testable, well-documented software and sharing ownership of code were key to achieving this. Given the importance of software quality, it may be possible to use tool-based measures of this as a proxy for intrinsic sustainability at the time of development.

Actively maintain the software

Writing good software is only the first step towards sustainability; at any given point, software only remains sustainable if it is being actively sustained. This applies to many software products, but is particularly an issue for research software which, due to its continuously evolving nature, often has dependencies that are continuously evolving themselves; software that is not updated decays when its

dependencies are updated. The technologies used for development in particular evolve rapidly: 'mainly because I don't think the technologies are that mature, you know, I think we're still learning. I mean the web moves at what kind of pace? The web changes constantly so it's hard to keep up.' (P6).

Use third-party libraries and embrace standards

If software is intended for reuse, it should be built on a solid foundation of well-tested, well-documented software itself. Technologies evolve rapidly, but where standards exist, using them will make it easier for others to extend or repurpose software in the future.

Make software available and discoverable

Ensuring that software is high quality does not in itself guarantee it will continue to be used. Software must be made available in open-source repositories, and it must be discoverable. Academic knowledge is promoted through publications, which are available in searchable digital libraries. 'So in some ways, I hate writing papers. But... Papers are probably around longer than the software is.' (P1). Whilst papers provide one means of publicising software, they are produced over a relatively long timescale, and do not evolve with the software. They also tend to focus on a scientific project, rather than software features, which may be useful to developers working in an entirely different domain. Promoting the functionality of the software itself will help to address this gap.

Rethink resourcing

Building and maintaining software requires resources. At present, funding is focused on resourcing scientific projects for finite periods of time, rather than developing scientific software. Historically, this made sense: software developed for a particular study was often prototypical or uni-purpose, with no requirement for it beyond the project end. The relationship between science and software is now changing, however, and researchers are coming to rely on software tools. In the same way that businesses employ developers to work on open source software that serves a purpose for them, research funding may need to go into supporting software, as a form of facility or equipment maintenance. One suggestion for measuring the value is the size of the user base: 'Well it needs to have a reasonably large user base. Well that could be anything from 50 to 1000 people. [Then] there is some community around it.' (P8).

Build a community around the software

In order for software to remain active, it requires a community of current users and developers able to maintain it (who may in some circumstances be the same people). Building a community of interested parties is important to demonstrate the value of the software, and ensure there is a critical mass of people willing and able to continue development.

Build a research software engineering community

The life of research software can continue beyond the project in which it was developed, and beyond the purpose for which it was originally built. Libraries, features, algorithms, operations and user interface design can all be reused, reducing waste and spreading good practice. To achieve this, software needs to start being visible beyond its immediate user and development group. A key goal should be sharing not just within the application domain, but within the discipline of research software engineering. The UK Community of Research Software Engineers[3] is one such example of a community being built to specifically support research software engineering practitioners.

6 Future work

The results of this study provide interesting insights into research software developers' views on sustainability, and how, as a community, we should move forward to consider sustainability in broad terms. A limitation of this study is that it spoke to a single group of developers within a single institution. The next step in this work is to widen the sample to include further institutions, both to validate the findings of this study, and to build a more comprehensive set of recommendations with a firm grounding in the experience of people who actually engineer research software themselves.

Acknowledgements

Mário Rosado de Souza is a visiting student from the Federal University of Lavras (UFLA), funded by Brazil's Science without Borders programme.

References

[1] K. Crowston, K. Wei, J. Howison, and A. Wiggins. Free/libre open-source software development: What we know and what we do not know. *ACM Comput. Surv.*, 44(2):7:1–7:35, Mar. 2008.

[2] B. Penzenstadler and H. Femmer. Towards a definition of sustainability in and for software engineering. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*.

[3] A. Raturi, B. Penzenstadler, B. Tomlinson, and D. Richardson. Developing a sustainability non-functional requirements framework. In *Proceedings of the 3rd International Workshop on Green and Sustainable Software*, 2014.

[4] R. C. Seacord, J. Elm, W. Goethert, G. A. Lewis, D. Plakosh, J. Robert, L. Wrage, and M. Lindvall. Measuring software sustainability. In *Proceedings of the International Conference on Software Maintenance*, ICSM '03, 2003.

[5] C. Venters, L. Lau, M. Griffiths, V. Holmes, R. Ward, C. Jay, C. Dibsdale, and J. Xu. The blind men and the elephant: Towards an empirical evaluation framework for software sustainability. *Journal of Open Research Software*, 2(1), 2014.

[3] www.rse.ac.uk