Audio Engineering Society

Convention Paper 8872

Presented at the 134th Convention
2013 May 4–7 Rome, Italy

Software Techniques for Good Practice in Audio and Music Research

Luís A. Figueira, Chris Cannam, and Mark D. Plumbley

Queen Mary University of London

Correspondence should be addressed to Luís A. Figueira (luis.figueira@eecs.qmul.ac.uk)

ABSTRACT

In this paper we discuss how software development can be improved in the audio and music research community by implementing tighter and more effective development feedback loops. We suggest first that researchers in an academic environment can benefit from the straightforward application of peer code review, even for ad-hoc research software; and second, that researchers should adopt automated software unit testing from the start of research projects. We discuss and illustrate how to adopt both code reviews and unit testing in a research environment. Finally, we observe that the use of a software version control system provides support for the foundations of both code reviews and automated unit tests. We therefore also propose that researchers should use version control with all their projects from the earliest stage.

1. INTRODUCTION

The need to develop software to support experimental work is almost universal in audio and music research. Many published methods comprise both written material and software implementations. Yet the software part of the work often goes unpublished and, even if published, it is not always clear whether it works correctly. In our earlier survey of the UK audio and music research community [1], of the 80% of respondents who said they developed software, some 40% said they took steps towards reproducibility, but of those only 35% (i.e. less than 15% of the total) had actually published any code. In the same survey, 51% of the respondents who reported developing software said that their code never left their own computer. While there are many examples of active researchers who do publish their software, informal observations suggest that the lack of publication of research software is widespread in many areas of audio research.

Research papers typically carry out validation of the

implemented model, evaluating how closely it corresponds with observations from reality. But any requirement to verify the implementation, to establish that the model described in the publication is the same as that actually implemented, usually goes unmet.

We have previously argued [2] that a lack of quality or confidence in the implementation may be a cause of failure to publish code. Similarly, studies of data sharing in other fields have reported a relationship between willingness to share supporting data and quality of reporting in the paper [3]. A cause for lack of confidence with code may be the lack of formal training in software development for researchers working in academic fields outside computer science. A recent study [4] found a great deal of variation in the level of understanding of standard software engineering concepts by scientists, and found that in developing and using scientific software, informal self-study or learning from peers was commonplace.

The audio and music research community is particularly heterogeneous in terms of academic backgrounds, with researchers having very different origins: as well as those from computing and engineering fields, many researchers in this community come from backgrounds with no formal training in software, such as music, dance or performance [1]. Such a wide variety of backgrounds poses a big challenge, given that many of these researchers do not have the skills or desire to become involved in traditional software development practice or in publication and maintenance of code.

In the remainder of this paper we will look into the software development practices of industry and discuss how some of these can be adopted by the audio and music research community.

2. FEEDBACK CYCLES

In commercial software development, much of existing "best practice" consists of trying to shorten or simplify feedback cycles so that the developer learns about mistakes as soon as possible. This is made explicit in incremental software development methods such as the Agile process [5], where there is a strong emphasis on individuals and interactions and use of techniques such as pair programming, continuous integration and unit testing. These techniques are used to facilitate short development cycles, with regular and quick software releases, making development as flexible as possible.

Even in software companies that do not adopt Agile procedures, however, there are some techniques very widely used in commercial development—including in companies throughout the audio and music software industry—that are not widely adopted in academia. Fig. 1 sketches some of these techniques, making a hierarchy of feedback mechanisms operating at different speeds, from most immediate (cycle a—compiler errors) to least (cycle g—reports from users of published software).

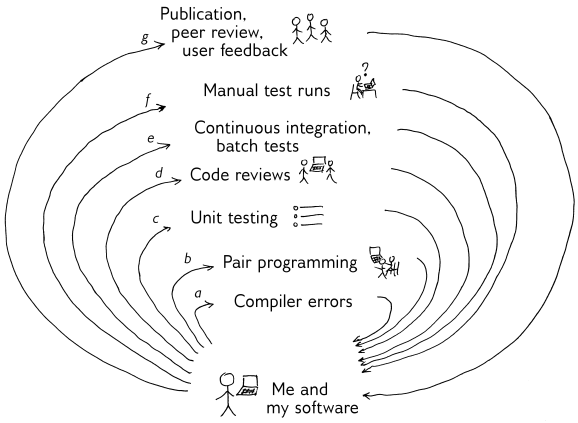

Fig. 1: Software development feedback cycles in industry

Researchers are familiar with a similar set of feedback processes in the development of written publications: collaborative review and editing with immediate peer researchers and supervisors, formal peer-review cycles, and so on. These processes provide new perspectives which help to produce quality results more quickly and help to remove potentially serious errors.

3. FEEDBACK CYCLES IN RESEARCH

The three most typical feedback cycles in research software development are illustrated in Fig. 2: (a) compiler errors, or immediate feedback from an interactive development environment; (f) using the code to run experiments, examining the results, and judging whether or not they appear to make sense;

and (g) obtaining feedback from users following publication of the software.

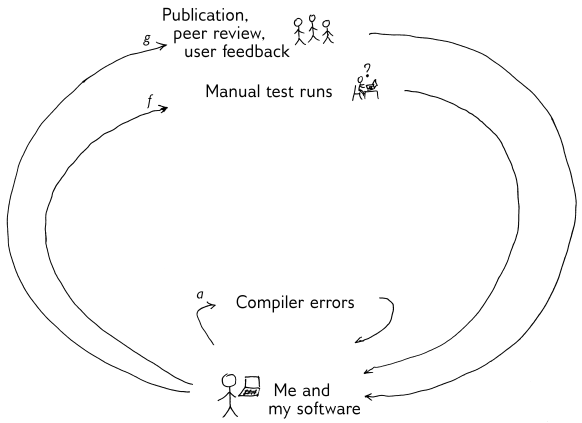

Fig. 2: Common feedback cycles in research software development

Compiler feedback, as depicted in Fig. 2, loop a, is highly immediate, but contains no information about program correctness other than basic syntax and type checking.

Manually running and inspecting the output, as represented in Fig. 2, loop g, is a fundamental part of any software development procedure, but it is hard to interpret the feedback it provides. If we don't know the expected output of an experiment, we may be unable to distinguish between genuine experimental results and artefacts caused by errors in the code. We also obtain no feedback on aspects of the code not formally evaluated in the experimental design.

The last loop, represented in Fig. 2, loop g, has its own limitations. It requires that the researcher publishes the code, which as noted in Sec. 1 does not always occur—often for reasons relating to the lack of applicable feedback during development [2]. Secondly, the speed at which feedback can be obtained and amount of feedback available are heavily dependent on publication times and uptake among users.

We believe that researchers can take better advantage of other available feedback mechanisms to give them more confidence in the quality of their code as quickly as possible. Specifically, we now introduce two of these techniques—peer code reviews and unit testing—and explain why and how they may profitably be applied even for small informal projects in an academic research environment.

4. CODE REVIEWS

A code review (Fig. 1, loop d) simply consists of having another person read one's code before running any significant experiments with it. This is analogous to copy-editing a paper publication.

4.1. Code reviews can be quick and informal

Studies have shown that the first hour of code review is the most productive. In [6] and [7], Dunsmore et al found that defects were found at a roughly constant rate through the first 60 minutes of inspection, then tailing off so that no further defects were discovered after the first 90 minutes. This result suggests that code reviews need not be too great a burden on time.

In [7], the same authors compared three methods of formal code review, which they identified as "checklist", "systematic" (or "abstraction") and "use-case" methods. The checklist method requires preparing and following a list of common sources of error in the language under inspection; the systematic method calls for the reviewer to reverse-engineer a specification for each function from its source code; and the use-case method examines the context in which code is used by tracing the flow of data, line by line, from likely inputs to expected outputs of each function.

The use-case method formalises the intuitive reading approach taken by a peer code reviewer familiar with the problem domain but not with the implementation: working from the likely inputs to each function, toward the expected outputs. Academic peers who are familiar with the field in which a program operates can apply this method informally by considering input data typical of the domain and tracking how it is transformed through the lines of code within a function. Although the use-case method did not perform as well as the checklist method in the Dunsmore et al study [7], the difference was small and we would suggest it as a good starting point for an informal code review process to be applied within a research group.

4.2. Reviewing code contributes to learning

Effective code reviews require some ability to read code. Reviewers reporting a high level of code comprehension have been shown to find more defects [8],

and training developers in "how to read code" makes a substantial difference [9] to the number of errors found during code review.

This suggests that learning to read code is a valuable part of learning to write and debug software, a principle supported by [10] and others. Formal training in code comprehension may be of benefit to the researcher-developer who wishes to review code, but in addition to any formal training, the exercise of code review is itself a learning opportunity.

Although the studies cited here show improvements in review performance due to improved code comprehension and review methods, a high level of comprehension is not a prerequisite for taking part in reviews: in [8] it is noted that some reviewers who did not fully understand the code nonetheless found around half of the defects in it.

4.3. Reviews are effective

In a well known article, Fagan [11] shows that code reviews can remove 60−90% of errors before the first test is run [12]. Comparative studies of code review techniques and reviewers support the proposition that code review is generally effective, showing for example that "median" reviewers found 60−70% of defects [8] or that reviews exposed all but one defect in the sample code [7].

4.4. Recommendations

We therefore suggest code reviews as appropriate for academic software development for three reasons:

1. Code reviews can be carried out quickly and informally in a peer setting such as a research environment;
2. Carrying out reviews gives the reviewer and author practice in reading code and in writing code that can be read, both of which are valuable in improving software development ability;
3. Code reviews are more effective at finding errors than any other commonly used technique.

5. UNIT TESTING

Unit testing is a second technique, widely used in non-academic settings, for providing software developers with feedback (Fig. 1, loop c).

A unit test is a means of automating and repeating tests of the individual parts of a program. It consists of code that calls a function in the program, gives it some input, and tells the developer whether it returns the expected result. If all non-trivial functions have tests, and if those tests provide a reasonable coverage of both likely inputs and edge cases, this provides a baseline assurance that the components of a program work as expected, even if the program as a whole is complex or changing.

5.1. Testing research software can be hard

Much research software is written to perform novel work: potentially complex experiments with previously unknown outputs. How do we test such software, given that the authors are not able to predict its output?

Unit testing goes some way toward helping with this. For code to be testable through unit tests it must be written in sufficiently small functions, with clearly defined inputs and outputs, for each to be tested mechanically. At this level it should indeed be possible to calculate the expected output of a function, given sufficiently simple choices of input. Further, code written in this way is easier to read and reuse.

Unit tests written during software development also provide early sanity-checking for code. A program to read data from a file and calculate a result might fail, not only because the basic algorithm is wrong, but because the input is read wrongly or the code fails to deal with unusual cases such as short or empty datasets or missing values.

Although unit testing cannot guarantee that a program produces the correct results, ensuring that it is built from correct components is a useful step toward ensuring it actually implements the method described in the publication.

5.2. Unit tests help avoid regressions

In software development, a regression is an error introduced accidentally while modifying software—something which used to work but has since become broken, such as a new bug introduced while fixing something else.

Regressions are common: in [13] Robert Glass identified 8.5% of errors identified after delivery of software from a substantial commercial aerospace

project as regressions. Regressions may also be dangerous, because they change behaviour that may be relied upon or that has been already published; and regressions are dispiriting, because they are examples of unambiguously wrong behaviour that suggest the developer is making negative progress.

Although regression testing is especially important when working in a team, such errors also occur in solo projects, and regressions can also arise from factors outside a developer's own control such as platform dependencies or changes between versions of a third-party library.

Automated tests are vital in avoiding regressions, because regression testing is simply too difficult and tedious to perform manually. It would require testing every use case by hand after every code change, which is not feasible. Unit tests can be run quickly after every build, and can identify with some precision the part of the code that has been broken by a change.

5.3. Test-driven development may be a useful analytical tool

A fundamental discussion around unit testing concerns when to write the tests.

Some developers, e.g. [14], advocate test-driven (or test-first) development, a process in which the unit tests are written before the rest of the code. The act of writing tests constructs a contract which the implementation is then written to satisfy. In principle, this ensures the code will be correct as it is first written.

Erdogmus et al. [15] found that those developers who adopted a test-driven strategy wrote more tests; that the developers who wrote more tests tended to be more productive; and that the quality of software produced was roughly proportional to the number of tests provided. A further survey [16] found that the quality of tests in a test-driven approach was often higher than that of those written after the implementation. However, we are unaware of any studies that show a direct link between the test-driven approach and higher quality or productivity, controlling for other factors.

Nonetheless, what makes test-driven development interesting for research software is the opportunity it affords to approach a problem from the opposite perspective to that usually used when simply writing code. Rather than begin by considering how to implement a solution and then think about how to test whether it works, the test-driven approach begins by considering what would be necessary to establish whether a solution was correct or not, before filling in the implementation. We believe this approach may be a mental tool worth considering when approaching hard-to-implement problems, regardless of whether it ultimately produces better code or not.

5.4. Recommendations

We propose unit testing as part of a practical, research-driven development environment for the following reasons:

1. Research software can be hard to perform high-level testing for: unit tests are a relatively easy way to obtain any assurance of correctness;
2. Unit testing is a good way to defend against regressions, a common class of error which can be hard to discover by manual testing;
3. A form of unit test development, *test-driven* development, can provide an additional analytical perspective when solving difficult problems.

6. VERSION CONTROL

A version control system is a tool that tracks the history of all files in a software project, logging changes and different versions, thus allowing researchers to easily compare different versions of the same file. Version control systems also help developers to share and merge changes, making projects easier to manage when several researchers are working together on them. Commonly used version control systems include git[1], Mercurial[2], and Subversion[3].

Besides making general management of code easier and more pleasant, a version control system provides a necessary foundation for effective code reviews and unit testing.

In allowing its user to see the differences between respective revisions of a project, a version control

[1] http://git-scm.com/
[2] http://mercurial.selenic.com/
[3] http://subversion.apache.org/

system makes it possible to isolate a set of changes and be sure that a code review is covering everything that has been changed in a particular piece of work. It also provides a mechanism for sharing code between developers, such that they can be sure they are looking at the same version of the same software.

In the context of unit testing, especially for regression testing, a version control system provides the necessary support to compare two versions of code and identify the source of a change in behaviour.

7. RECOMMENDED ACTIONS FOR A RESEARCH GROUP

7.1. Code reviews

As noted above in Sec. 4, in a research environment we advocate the adoption of an informal style of "use-case" code review. Here the reviewer follows data paths through the code, from an imagined input to the result output, with the original developer available to explain what each step is for if necessary. This technique can be conducted without much preparation, because the reviewer does not need an in-depth knowledge of the specifics of the implementation and there is no requirement to make checklists or other aids prior to the review process.

Wilson et al [17] propose the use of "pre-merge" code reviews, in which reviews are a requirement for code to be included into a shared repository. When using a modern distributed version control system, this could be set up either by committing to a user's own branch and then merging to a main branch after review, or else by having the developer commit only on their local machine and get the code reviewed in person by another researcher before pushing to a central repository. In either case, it is important to review before performing experiments using the code.

7.2. Unit testing

Unit tests should be small and as simple as possible. The aim is to exercise the code in unexpected ways, focusing on tricky small cases rather than easy large ones. Studies show that areas of code where bugs are found are more likely to be fragile in general [18, 19] and bugs that have already been found are relatively likely to reappear as regressions, so more time should be focused on testing areas that have already proven problematic or where bugs have been found in "finished" code.

Be careful to avoid exact comparisons when testing outputs from inexact calculations or those using floating-point arithmetic. The error margin for a calculation is part of the test, just as the expected result is.

Because it can be hard to get started in writing unit tests, we propose adopting the simplest available unit test framework. In some cases, it may even be easiest to write unit tests without a framework: call a function; complain if the result is wrong. More usually the simplest method will be to adopt a standard test framework, such as Nose for Python[4]. Other authors [17] advise using a test framework for all unit testing.

7.3. Version control

A version control system should be used from the very beginning of any project, regardless of its complexity. Recent reports from other fields show encouraging trends in adoption of such systems in an academic setting [20]. In general, the right system to choose is whichever is available in the research group or in use by peer researchers. In the absence of a local standard, we recommend the use of a modern distributed version control system such as git or Mercurial.

Similar guidelines apply to the choice of hosting services. Several free solutions are available on-line, such as GitHub[5] for git, or Bitbucket[6] for both git and Mercurial projects with good support for private work. In addition to these generic hosting sites, there are subject-specific sites such as the SoundSoftware project site[7] for UK-based researchers in the audio and music field.

8. CONCLUSIONS

Throughout this paper we argue that code reviews and unit testing are valuable techniques that can and should be adopted in academic contexts. We introduce these techniques in the context of a set of feedback loops that give the researcher-developer timely information about problems in their code.

[4] https://nose.readthedocs.org/en/latest/
[5] https://github.com/
[6] https://bitbucket.org/
[7] http://code.soundsoftware.ac.uk/

We have also given some advice and pointers to how to adopt such techniques in a research environment, such as:

- apply an informal style of "use-case" code review technique, calling on peer researchers to review code before using it in substantial experiments;
- adopt the simplest available unit testing regime, keep unit tests small and concise, and apply a standard test framework for unit tests on larger pieces of code;
- support these techniques with the use of a version control system from early in a project.

Techniques such as these rely on becoming comfortable with the idea that others will be looking at one's research code. We believe that this is an important step, and one that makes documentation and unit testing easier, code reviews possible, and publication and peer review of software less scary.

As part of our ongoing UK-based SoundSoftware project[8], we have developed several materials to add to this subject, such as handouts with practical guidance, video tutorials, and slides. You can freely access these on the project's website[9].

9. ACKNOWLEDGMENTS

This work was supported by EPSRC Grant EP/H043101/1. Prof. Mark D. Plumbley is also supported by EPSRC Leadership Fellowship EP/G007144/1.

10. REFERENCES

[1] I. Damnjanovic, L. A. Figueira, C. Cannam, and M. D. Plumbley, "SoundSoftware.ac.uk Survey Report." http://code.soundsoftware.ac.uk/documents/17, 2011.

[2] C. Cannam, L. A. Figueira, and M. D. Plumbley, "Sound software: Towards software reuse in audio and music research," in *Proceedings of the IEEE 2012 International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2012)*, pp. 2745–2748, IEEE Signal Processing Society, March 2012.

[3] J. M. Wicherts, M. Bakker, and D. Molenaar, "Willingness to share research data is related to the strength of the evidence and the quality of reporting of statistical results," *PLoS ONE*, vol. 6, p. e26828, 11 2011.

[4] J. E. Hannay, C. MacLeod, J. Singer, H. P. Langtangen, D. Pfahl, and G. Wilson, "How do scientists develop and use scientific software?," in *Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, SECSE '09, (Washington, DC, USA), pp. 1–8, IEEE Computer Society, 2009.

[5] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, "Manifesto for Agile Software Development," 2001.

[6] A. Dunsmore, M. Roper, and M. Wood, "Object-oriented inspection in the face of delocalisation," in *Proceedings of the 22nd international conference on Software engineering*, pp. 467–476, ACM, 2000.

[7] A. Dunsmore, M. Roper, and M. Wood, "Practical code inspection techniques for object-oriented systems: An experimental comparison," *IEEE Software*, vol. 20, pp. 21–29, 7–8 2003.

[8] A. Dunsmore, M. Roper, and M. Wood, "The role of comprehension in software inspection," *Journal of Systems and Software*, vol. 52, pp. 121–129, 2000.

[9] S. Rifkin and L. Deimel, "Applying program comprehension techniques to improve software inspections," in *In Proceedings of the 19th Annual NASA Software Engineering Laboratory Workshop*, pp. 3–8, 1994.

[8] http://soundsoftware.ac.uk
[9] http://soundsoftware.ac.uk/resources

[10] L. E. Deimel and J. F. Naveda, "Reading computer programs: Instructor's guide and exercises," Tech. Rep. CMU/SEI-90-EM-3, Software Engineering Institute, Carnegie Mellon University, August 1990.

[11] M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal*, vol. 15, no. 3, pp. 182–211, 1976.

[12] R. L. Glass, "Inspections - some surprising findings," *Commun. ACM*, vol. 42, no. 4, pp. 17–19, 1999.

[13] R. L. Glass, "Persistent software errors," *IEEE Trans. Software Eng.*, vol. 7, no. 2, pp. 162–168, 1981.

[14] K. Beck, *Test-Driven Development By Example*, vol. 2 of *The Addison-Wesley Signature Series*. Addison-Wesley, 2003.

[15] H. Erdogmus, M. Morisio, and M. Torchiano, "On the effectiveness of the test-first approach to programming," *IEEE Transactions on Software Engineering*, vol. 31, pp. 226–237, 2005.

[16] B. Turhan, L. Layman, M. Diep, F. Shull, and H. Erdogmus, "How effective is test driven development?," in *Making Software: What Really Works, and Why We Believe It* (G. Wilson and A. Orham, eds.), OReilly Press, 2010.

[17] G. Wilson, D. A. Aruliah, C. T. Brown, N. P. C. Hong, M. Davis, R. T. Guy, S. H. Haddock, K. Huff, I. M. Mitchell, M. D. Plumbley, B. Waugh, E. P. White, and P. Wilson, "Best practices for scientific computing," *Computing Research Repository (CoRR)*, vol. abs/1210.0530, 2012.

[18] A. Endres, "An analysis of errors and their causes in system programs," in *Proceedings of the international conference on Reliable software*, (New York, NY, USA), pp. 327–336, ACM, 1975.

[19] B. W. Boehm and V. R. Basili, "Software defect reduction top 10 list," *IEEE Computer*, vol. 34, no. 1, pp. 135–137, 2001.

[20] K. Ram, "git can facilitate greater reproducibility and increased transparency in science," *Source Code for Biology and Medicine*, 2013.