Improving software quality in bioinformatics groups through temawork

This manuscript (permalink) was automatically generated from ferenckata/SQSeminarPaper@ad05117 on November 6, 2023.

Authors

- **Katalin Ferenc** ✉
  0000-0002-3006-4297 · ferenckata
  Centre for Molecular Medicine Norway (NCMM), Nordic EMBL Partnership, University of Oslo, 0318 Oslo, Norway · Funded by Grant XXXXXXXX

- **Ieva Rauluseviciute**
  0000-0001-9253-8825 · ievarau
  Centre for Molecular Medicine Norway (NCMM), Nordic EMBL Partnership, University of Oslo, 0318 Oslo, Norway

- **Ladislav Hovan**
  0000-0001-8847-9295 · ladislav-hovan
  Centre for Molecular Medicine Norway (NCMM), Nordic EMBL Partnership, University of Oslo, 0318 Oslo, Norway

- **Vipin Kumar**
  · princeps091-binf
  Centre for Molecular Medicine Norway (NCMM), Nordic EMBL Partnership, University of Oslo, 0318 Oslo, Norway

- **Anthony Mathelier** ✉
  0000-0001-5127-5459
  Centre for Molecular Medicine Norway (NCMM), Nordic EMBL Partnership, University of Oslo, 0318 Oslo, Norway; Department of Medical Genetics, Institute of Clinical Medicine, University of Oslo and Oslo University Hospital, Oslo, Norway

✉ — Correspondence possible via GitHub Issues or email to Katalin Ferenc <k.t.ferenc@ncmm.uio.no>, Anthony Mathelier <anthony.mathelier@ncmm.uio.no>.

Abstract

Introduction {.page_break_before}

Bioinformatics and computational biology are continuously gaining importance in biological research. These fields are heavily relying on inventions of computer science and software engineering. As Goble pointed out in 2014, about 90% of researchers used or relied on results produced by scientific software [1]. Goble, and others also highlighted that one implication of poor software engineering practices is that incorrect software results in invalid scientific findings [1,2]. Beyond incorrect software, even when the software performs as intended, researchers spend significant amount of time building software using suboptimal practices which can further increase the necessary time investment in the future [3].

Good software development practices to mitigate the risk of incorrect software solutions has been established in other software-heavy endeavours. However, bioinformaticians lack formal education in computer science and related subjects [2,4,5]. Such lack of theoretical and practical foundations hinders the adoption of good practices (e.g. continuous integration, unit tests, code reviews, planning of software architecture). Indeed, unknown unknowns may include most of the software quality attributes defined by the International Organization for Standardization [6] (Figure 1).

Figure 1: ISO25000

Figure 1: ISO25000

Beyond the limits of individual education, many of these practices rely on redundancy of knowledge within team members, supported by practices such as pair programming, daily stand-up meetings and code reviews. One main feature of academic research groups is the research projects conducted by a single PhD student or post-doc. In fact, academia prides itself on enabling individual achievements, and generally measures the success of individual effort while disregarding team effort. This often results in very few people maintaining the software product, which might be used by thousands of researchers worldwide [5,7,8,9]. Lack of funding contributes to the poor maintenance status of much of scientific software [1]. The lack of funding is being recognized and addressed by few agencies, one of which is the Chan Zuckerberg Initiative Essential Open Source Software for Science fund [10].

The concept of a team is therefore different in a science project from a software project. While group members help each other with scientific suggestions, most often there is a single responsible person for the design and implementation of the code base. As official guidelines are rarely available on coding practices, the actual craft of software engineering is treated as secondary and up to individual judgment. These guidelines may naturally emerge in larger groups, following them requires a form of team organization not intrinsic to academic groups. In line with this, Hannay et al. [4] found that researchers tended to rank software engineering concepts higher if they worked in a team. Russel et al. [8] showed that high-profile code bases feature larger development teams than low-profile ones, and their activities are also associated with longer commit messages as a sign of better communication and documentation of changes. Relying on these findings, we asked ourselves whether a form of team structure organized around individual software products could improve the quality of our scientific code.

In this work we first review the findings of software engineer researchers on computational scientists, and their suggestions for improving the overall quality of scientific software. We do not limit our scope to bioinformaticians, as bioiformatics software quality and development practices share many characteristics with scientific software in general. Next, we review the literature on guidelines posed by bioinformaticians to the bioinformatics community. These works highlight the priorities within the community. Noticing the distance between software development practices and academic reality, we also turned to theory on team management. Finally, we discuss how our research group, inspired by the literature, incorporated weekly discussions on scientific software quality. Good practice requires investment in time and effort that may not pay off on an individual basis. Sharing the onboarding effort of new tools, and normalizing discussions on implementation details resulted in an overall better perceived process quality and code quality of the members with a reduced effort on an individual level. We aim to provide motiviation and framework on how to get started with improving the process quality of bioinformatics groups, even without members who have formal training in computer science or software engineering.

Status of scientific and bioinformatics software from the SE perspective

A landmark paper in 2007 by Diane F. Kelly [11] discussed the separation ("chasm") between software engineering and scientific-computing community. She points out the need to bridge the one-size-fits all software engineering solutions and the particularities of the scientific software development which relies heavily on domain knowledge. Without such bridge, scientific computations keep on being performed using error-prone development practices and reaching suboptimal solutions and poor software quality. Her predictions seem to hold true even after almost 20 years.

In the past two decades, software engineering researchers have been surveying bioinformatics software, and more broadly scientific software, and its development practices from the software engineering perspective[3,4,9,12]. There are multiple guidelines and suggestions to improve the quality of scientific code [1,13], many of which would target students of scientific disciplines [1,12]. Recently, an extensive literature review has been published which collects known issues and suggested solutions [2]. Yet, these guidelines seem to not have reached the majority of the bioinformatics community, which is found to be "still in its infancy compared with the majority of other scientific disciplines" [2]. Indeed, the guidelines suggested include following agile practices, the DRY (don't repeat yourself) principle, requirements gathering and unit testing, none of which concept is intuitive, well known in the bioinformatics community or even trivial to adapt to bioinformatics software [15]. Without a shift in coding culture within bioinformatics, these concepts might remain unavailable to help bioinformatics professionals.

lack of time on SE concepts? In case of concepts, like object-oriented programming and extreme programming, which were found to be known and used in the bioinformatics community [12], the authors note that these methods are not used to their full power. For example, they found that a bioinformatics programmer may use classes and inheritance, but rarely encapsulation and polymorphism [12].

Undoubtedly, scientific software development has its own challenges. However, it cannot be an excuse for skipping good practices: as Carole Globe [1] puts it "in Hannay's survey [4], only 47 percent of scientists had a good understanding of testing, and just 34 percent thought any formal training was important. This is strange because presumably they wouldn't use and trust the results of a microscope or telescope that hadn't been built by qualified engineers or tested. Yet software is the most prevalent of all the instruments used in modern science". Software engineering emerged and has been developing to address issues naturally arising from poorly planned development, such as project failures, delays, incorrect functionality or defects [16], none of which is unknown to the scientific community. In fact, the crisis of scientific software is fairly well known and suggestions are being made from both inside and outside the community [17,18].

One key challenge is the limited funding and the lack of recognition for development and maintenance of scientific software. Currently, as Alexander Szalay puts it “The funding stops when they (researchers) actually develop the software prototype” [19]. This is a problem, because future researchers would want to build on each other’s findings, use previous software as tools, and spend less time adopting or maintaining software [3,5,20]. In line with this notion, [12] found that bioinformatics software developers view their code as “means to an end” and care less about the future of the software they are writing. The authors point out that bioinformaticians are not well aware of the relationship between complexity, size, age, and the change-proneness of a code, which heavily affect maintainability.

Another obstacle is the non-trivial nature of testing of scientific software [14,15]. In a recent review paper [15] two key aspects of scientific software testing has been highlighted: the oracle problem and the cultural differences between scientists and software engineers. Software behaviour can be tested against an expected output, but often in science we use software to find new knowledge. This results in an oracle problem, when scientists actually do not know *a priori* how the software should behave, thus straight forward verification is impossible. According to the authors, scientists also view their scientific model and the implementation as a single entity. Therefore, scientists tend to test the validity of the model but not verify the code which produces it. Uncovered faults can and do lead to incorrect scientific insights as shown in multiple examples [21].

Nevertheless, software quality standards (Figure 1) are universally applicable. Depending on the application of the scientific software, whether it is a tool or a data analysis pipeline, the authors may prioritize different quality attributes [3]. For example, in the world of big data, performance and efficiency gain importance. Shown in a previous study reviewing mappers, individual tools have varying level of compatibility, usability, and portability [7]; quality attributes which directly impact user experience. Frameworks, such as Snakemake [22] or Nextflow [23] support usability, reliability, and maintainability. Anaconda [24] and container solutions [25,26] help achieve portability. These are also compatible with Snakemake and Nextflow, making these frameworks staple for reproducible data analysis.

In Table 1, we collected recommendations to improve the software quality of scientific, and whenever possible bioinformatics, software. The literature search was performed in multiple iterations using Google (to include grey literature) and Google Scholar based on phrases “scientific software development”, “software engineering bioinformatics” and “bioinformatics software recommendations” throughout 2023. Additionally, relevant articles were selected based on the snowball effect from the references of the initial publications. [3] notes that a scientific software developer needs to prioritize the software quality attributes to make choices among the good practices. They also provide a matrix to help with the selection and priorization. Beyond prioritization, the list below is somewhat daunting, especially while the main priority is to produce scientifically relevant results, and may actually discourage scientists to change their habits. One solution could be joining efforts and create a community to collectively increase software quality, while reducing the barrier to explore new techniques.

Table 1: Collection of recommendations for improving scientific software quality. Some guidelines are more vague than others, they also have varied scope, and they target different stakeholders. Therefore, it may be hard to find individual responsibility and actionable points from the literature.

Recommendation	Source
version control	[2,3,5,13]
user (and developer) documentation	[2,5,12,13]
standardised tests	[2,3,5,12,13]
independent review of source code	[12,13,20]

Recommendation	Source
recognition and assignment of adequate time for quality-assured development	[12,13]
recognition of software development as academic achievement	[13,20]
standardized working environment and automation	[2,13]
financial support for software development and maintenance	[13,20]
support for developer community for long term maintenance (when applicable)	[13,20]
licensing	[5,13]
requirements gathering	[2,12]
containerization for portability	[2,5]
reuse existing (reliable) software	[3,5]
agile software development methodology	[2,3]
educated choice of software development methodology	[12]
adoption of international best practice standards of software quality	[13]
establish validation and acceptance procedures	[13]
cooperation between developers and users	[13]
description of the software version used, its configurations and parameters in publications	[13]
preferentially selecting freely available open-source software	[13]
encourage user participation in the software development process	[13]
tagging of software version for reproducibility	[5]
sanity check on input parameters	[5]
do not hard-code changeable parameters and paths	[5]
rely on package managers	[5]
do not require superuser privileges	[5]
provide a small test set	[5]
ensure reproducibility of results	[5]
refactoring	[3]
usage of design patterns	[3]
quality monitoring (e.g. SonarQube)	[3]
continuous integration	[3]
contribute to open-source development	[20]

(TODO: join this table with a similar from the next section.) (The distinction may actually be a bit arbitrary between this section and the next.)

Review of existing suggestions for improving software quality in biomedical sciences

We reviewed the existing literature that focuses on providing guidelines for programming practices for scientists without extensive training in computational sciences. We used several key phrases to search for papers: “guidelines for bioinformatics software”, “rules for biologists learning bioinformatics”. Such papers are quite abundant and have a number of things in common. For example, they can focus on specific suggestions, often referred to as rules or “tips & tricks”, or they can more broadly direct towards good practices of coding, which are put together into “guidelines”.

Papers often choose a main focus or a theme and then distinguish a set of rules or guidelines. That focus can be very specialized, such as particular data analysis in a single disease [27]. Or the theme can be more general, for example setting the rules for next-generation sequencing (NGS) data analysis or outlining tips on how to start on computational analysis of the experimental data [28,29,30]. Both types of papers emphasize the need to learn how to analyse data properly and provide good suggestions to do that, based on the chosen topic. However, one needs to be aware when reading specialized focus guidelines that some of them can be much less applicable if the context of the analysis is different. Finally, the reference point matters. There is a difference, if the guidelines are read by a person with less experience in computational data analysis, where even a small set of rules can mean a steep learning curve, because some skills that are dependencies for the rules might be common knowledge for those with more experience. For example, a guideline to use version control systems, such as git, is a very important one, but for that a reader should most likely know the basics of the unix command line, which is not necessarily a rule on itself. One of the first things that should be done while suggesting your set of guidelines is clearly defining your target audience, even if they are meant to be broader suggestions.

As most guidelines tend to be written for researchers or students with minimal coding experience, suggestions often overlap. Most commonly highlighted are documentation and version control [31,32]. These are basic aspects of the software quality, which are still often overlooked in the publications making it hard to reproduce the research or use a published software [2]. Even the people with the least experience in computational analyses should always document their code by writing extensive README documentation. Documentation practice is very strong in the wet lab, where having a lab journal is unquestionable. This should not be different when working with computational tools. Version control of the code improves research reproducibility and the usage of the software. The most popular way to do that is by using git and remote repositories, for example, GitHub or Bitbucket [27,32,33]. Less commonly emphasized are testing of the developed code or software and optimization of pipelines [32]. These are very important rules to follow to end up with a proper collection of code or a stable software [2]. However, it might be considered to be too advanced for “beginners” guidelines. Unfortunately, not being aware of such requirements can lead to a code that either takes a long time to run or is buggy, which accumulates the technical debt and, in a way, encourages the bad practices. To avoid that it is important to at least be aware of the caveats of your code and document them for fellow researchers that might use your code in the future.

As code testing and organization can be overwhelming for a less experienced computational scientist, there are multiple available tools to help organize the coding tasks and the code itself. For example, using task management through Jira or GitHub issues, which are commonly used in team projects, where multiple people work on the same code. However, these resources are not emphasized in the guideline’s literature as this literature is most commonly focused on the individual persons and their personal practices. Often “other people” are only mentioned when guidelines suggest where to seek help when encountering a problem with the code. This includes consulting with colleagues, finding a mentor or participating in online communities (for example, Stack Overflow or Biostars) [32]. More recently, new tools are introduced that integrate artificial intelligence (AI) as a helper in coding. Tools such as GitHub copilot or editor extensions with access to ChatGPT allow to ask the models to write a piece of code to address a problem. To our knowledge bioinformatics literature almost never presents suggestions how to code in a team setting and utilize multiple people’s expertise on software development. In contrary to software engineering-oriented literature, where there is a lot of focus on coding in a team practices [34,35]. Hagan et al. described Code Clubs - the practice in their research lab, where group members are collectively engaged in software development through code reviews and pair coding and software engineering education through workshops or seminars [36]. The authors give tips on how to organize such meetings and what should be the ground rules. Sharing your coding experience with others helps minimize the isolation, allows individuals to learn from their peers, and finally helps to write a better quality software.

Review of team management literature in the context of software engineering

Programing as a collective practice is a key notion in software engineering, which prompted a corresponding rich literature examining how to effectively organise it [**doi:10.1287/mnsc.l080.0921?**]. A central theme in this literature is maximising team cohesion while minimising code coupling. Authors (citation needed) argue that the viability of a software project along it succesive development phases is largely determined by the adoption of sound software design enforcing modularity and extensibility coupled with team management practices centered around communication and collective governance.

Before delving into more specific recomendations, let us first define the scope of what constitutes management. We understand management as the set of tasks ensuring the viability of a software project. These tasks revolve around planning, monitoring resources, and tracking progression [37]. Typically the overseeing of these functions would be taken up by a single individual referred to as the “manager” of a project.

In the particular context of computational projects in academia, a strict division of labour is rarely found with regards to the management of software projects. Furthermore, some tasks, such as risk, budget and time management and maintenance are decoupled from the actual software development phase. The remaining management tasks would often be deliberated by the developer(s), eventually reaching a consensus on the desired way forward and acted upon.

This sort of self-management, at times collective, echoes some of the prescriptions of the SCRUM method [38]. SCRUM is a framework to perform these management tasks through team self-management. This framework was introduced to respond to the aspiration for more autonomy and responsivity from software developers, best illustrated by the agile manifesto [39]. This proposition was a reaction to the typical blueprint-like management for engineering projects which proved ineffective in addressing the emerging challengens of large software projects[38] This similarity is probably the reason why agile practices are part of guidelines for scientific software developers (Table 1).

One outstanding aim of agile is the aspiration for more autonomy for organising the work of software developers. In the particular context of large computational project, agile opened the opportunity for collective governance and a move away from a project structure producing a division of labour coupling one developer to a particular task or aspect indefinitely. Incentivising a collective ownership and governance of the codebase as a whole, promotes the adoption of software engineering best practices among developers contributing to a software project[40]. Indeed by aspiring to make any developer within the team interchangeable across the various ongoing tasks, we create the need for robust testing, comprehensive documentation and coherence across the differents parts of the project [**doi:10.1287/mnsc.l080.0921?**].

Furthermore by exposing every developer to a variety of tasks over the course of the project development, we strengthen the knowledge and skill base of the team as a whole, as well as a better mutual awareness of team member expertise (transactive memory system [41]). Taken together these merits further improve the team's capacity to overcome technical challenges that will arise over the course of the development process.

The previous paragraph outlined some of the desirable outcomes of agile-like practices. Such benefits require the implementation and effective adoption of this mode of project management. This in turn relies on the execution of a variety of methods whose success in realising the merits of agile depends heavily on setting the adequate circumstances for the team to need to incorporate elements of agile in their regular work practice. Practices and methods aligned with agile prescriptions include stand-up meetings, task allocations, pair-programming, or code reviews. Note, that many of these practices do not require the presence of the manager, but assumes a work culture and standardised procedures. At their core, these practices incentivise continuous communication and collective decision meaking among developers. This constitutes an additional overhead in terms of time and ressources needed when developing, but this is offset by the aforementioned benefits in terms of coding practice, software resilience and improved team capabilities.

Optional section that might be put in discussion instead.

- Comparison to current academic coding
  - agile would be more stringent
  - agile would challenge the one project : one developer configuration
  - agile would highlight the impossibility to capitalise dividends of long-term building of knowlegebase and team chemistry
  - agile would point at the training gap
- Approaches and circumstances promoting the benefits of agile

Let us not forget that academia comes from a different place than where agile was developed. The Agile Manifesto was written against "bureaucracy, infantilization, and sense of futility" [38]. The academic software development agile practices would in fact be a more stringent approach to software development than current practices. For example it would include writing down requirements in form of user stories, plan a minimal viable product, plan an initial architecture, and dividing the project into tasks. This is important, because most literature pictures agile as management style free from traditional management. We should not forget that ad-hoc coding does not comply with agile, and we cannot use it as an excuse to continue our current practices. The agile system assumes that software engineering professionals seek to find the best approaches, and are well equiped to make good decisions on their own - when faced with shifting requirements and complex code base [38]. Additionally, the lack of top-to-bottom management, architects and system analysts in academic software development put even more responsibility on the individual developer. However, as noted in previous sections, current scientific software developer education does not necessarily cover these elements [12].

- Cost-Benefit of adoption

Working in teams is not an option, but a must for large projects. Although it is not necessary in smaller projects, the benefits are significant. - over time a small project might be taken over by another person, thus accidentally becoming a sort of team project with (by definition) insufficient communication - lack of standards and good practices undermine quality and maintainability - ease of technical knowledge transfer (which requires time and social factors) [**doi:10.1287/mnsc.l080.0921?**] - knowledge on who knows what (transactive memory system) speeds up problem solving [41]

However, it is not trivial to assemble a well functioning team. Authors found that the overall performance increases when team members are familiar with each other and build problem-solving routines together through cumulative experience [**doi:10.1287/mnsc.l080.0921?**]. Miriam Posner also points out that team management practices do not protect from a toxic environment [38].

Our experiences for development processes involving teams

The preceding sections mention a lot of possible approaches to improving software quality. Given the abundance of opinions on this topic, and the variety of challenges bioinformaticians face, we believe that everyone should find out what works best for them. Here, we describe the practices that we have settled on.

The software development practices that we have adopted can be broadly separated into three categories: code reviews, what we have called software quality meetings, and resource sharing.

Code reviews

Code reviews are not a new invention and many people have discussed their benefits [38]. Here we would just like to briefly summarize how being made to present your code and receiving feedback leads to improvement in the process of creating software.

Prior to a scheduled code review, the author is forced to write their code in a way that it will be explainable and understood by others, which is always desirable. In a large distributed project this may be trivial, but because the bioinformatic projects are often handled by a single person, it is very possible to make the code needlessly complex and obfuscated. We also observed that during data analysis parts of the code are re-run in an ad-hoc manner (e.g. by commenting out parts), making it increasingly difficult to reproduce the same analysis.

During the code review, the author has to explain some aspect of their code clearly (e.g. structure, algorithm implementation, performance related decisions), which depends on them understanding it. Trying to explain your code to someone is shown to help with understanding (rubber duck method [43]). The feedback obtained can help fix existing or potential future issues, improve the implementation, and produce cleaner, more concise code. The other participants may not be deeply familiar with the particular project, but they have their unique knowledge and point of view. We agree with the ten simple rules described by Hagan et al. [36], and note that many of those naturally emerged as a code of conduct after a few rounds of trial and error.

After the review, the received suggestions should be implemented swiftly to improve the code before advancing the project. The success of code review is highly dependent on its frequency (long time between reviews - a lot of new code, hard to cover all changes in a single session, potentially a lot of rewrite post review), and hence they should be as regular and frequent as reasonably possible.

Our experience indicate a broader adoption of notions and practices of good software engineering standards highlighted during these code review sessions. Here we will focus on couple examples to illustrate how code reviews incentivised coding practices and team self-managements aligned with agile prescriptions. Code review involves some elements of problem solving, often revisiting fundamental notions of design patterns, algorithms or data structures. Recurringly we would examine best strategies to modularise the presented code and discuss what would consitute effective and self-contained computational task and elaborate collectively possible design patterns. This strengthens the team’s overall competency as well as promoting some form of standardization regarding the mental models to use for common tasks and objects solicited in many computational projects. An important part of the code review process focuses on the compliance with good code practices, and constitutes an explicit attempt at standardization. This is particularly well illustrated with the review of documentation which goes beyond simple linting. Effectively this process promotes the adoption of a shared and systematic manner to describe and document the behavior of the considered tool, which facilitates its intelligibility for a wider audience.

As a positive additional outcome, we noticed an increasing understanding in each other's projects that naturally emerged through talking about the analysis code. This enabled us to give more involved comments during subsequent group meetings too. We noted however, that the focus can easily shift from the code to the biological question at hand. This we believe is more of a feature than a bug, as each code review session is led by the person bringing the code and the rest of us are there to support to the best of our abilities. Especially after the general level of coding style and quality increased to a good baseline. E.g. after about half a year, it was trivial for everyone involved that code organized into functions is preferred over spaghetti. The shared knowledge base and standards also allow us to make new group members adopt good coding practices more quickly.

Software quality meetings

Within the framework of software quality meetings, we have established larger-scale knowledge transfer between the participants. Presentations and demonstrations of new techniques and tools that are not necessarily tied to a specific project help broaden our knowledge base and awareness. In this sense, they form almost a substitute for a more formal computer science education, which most bioinformaticians lack [2]. Topics can arise from code reviews, own projects, or effectively be a reproduction of a useful talk or seminar given elsewhere.

The presenters benefit as well by having to research the topic further and present it coherently. It is not necessary to have these meetings be as regular as code reviews. The time investment is higher, given that a preparation is needed unlike just writing code as for code reviews.

During the software quality meetings, we have also explored the possibility of collaborative projects and pair programming, but have not managed to implement it successfully yet outside the scope of preparation for the JASPAR 2024 release (reference). The main reason for this is that we experimented with collaboration on a software tool not directly used by any of the members. As researchers, we could not afford to invest time in a hobby project.

The outcome of these sessions are manifold. A few examples: 1) a shared vocabulary that enables quick discussion about implementation details and code structures (e.g. design patterns, software architecture, data structures and algorithms), 2) a kind of toolkit and set of recordings we can sample from and build on in our own research projects (e.g. planning with UML diagrams, git features to ease and quicken software development), 3) awareness of previously unknown algorithms and packages, improving software performance and quality (e.g. dynamic programming, heap, Python packages such as bioframe).

Resource sharing

Resource sharing is a basic thing, but it boils down to making sure that useful online resources are brought to the attention of all participants easily.

Resource sharing could be discussed from two perspectives: external open-access resources (forums, repositories, packages and libraries) and internal (within-group resources with tools). The latter is very important as it allows for team contribution that can benefit the individual project development. A simple example of this could be a shared repository of various computational tools that were developed by members of the group. Such tools are universal enough and fit the group's research questions, so all people in the group can re-use them. In addition, each tool can be potentially developed and reviewed by multiple group members.

During software meetings, we aimed to set aside time to improve these tools from perspectives identified by the members. We observed that many of these tools do not have a clear scope and are rather a small script for a sub-task from a previous project. Based on this observation, we noted that there is a difference between a script and a standalone tool that can be inserted into various projects. The latter requires exploration of use cases related to the tool, handling of unexpected inputs, and extensive documentation, to name a few tasks. This understanding was actually quite relevant in a code review discussion when the expected usage modes of a new tool was the main focus.

(Explanation of figure) (Figure 2)

Figure 2: wall_climbing

Figure 2: wall_climbing

Discussion

We implemented a form of code review that fits our specific needs and context. We used our experience from these sessions, with special focus on team-based software development practices ensuring good code quality during in the update of the curation analysis software of the 2024 release of the JASPAR database (citation). In this project, as well as in some of our own individual research projects, we introduced user stories when documenting the assumptions, current features and new ideas, and we relied on development tools such as Jira and git. We note that the usage of these tools is not necessarily aligned with industry practices, due to the experimental nature of scientific software. Nevertheless, as bioinformatics becomes a more and more software-heavy field, we believe a good direction is to collectively lower the barrier to adapting to new technologies.

How to decide when it is time to invest in a script? [5] suggests a cut-off where a script is being reused, shared with others or used to produce findings in a publication.

There is a method called ad-hoc testing, which could be beneficial for scientists without test plans or documentation. It is flexible, allows creativity and discovery of new requirements [44]. However, it is not advised to be used on its own, as it is not robust and requires extensive know-how.

References

1. **Better Software, Better Research**
Carole Goble
IEEE Internet Computing (2014-09) https://doi.org/vjz
DOI: 10.1109/mic.2014.88

2. **Improving bioinformatics software quality through incorporation of software engineering practices**
Adeeb Noor
PeerJ Computer Science (2022-01-05) https://doi.org/gsm3hg
DOI: 10.7717/peerj-cs.839 · PMID: 35111923 · PMCID: PMC8771759

3. **Software engineering practices for scientific software development: A systematic mapping study**
Elvira-Maria Arvanitou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, Jeffrey C Carver
Journal of Systems and Software (2021-02) https://doi.org/gq3jtm
DOI: 10.1016/j.jss.2020.110848

4. **How do scientists develop and use scientific software?**
Jo Erskine Hannay, Carolyn MacLeod, Janice Singer, Hans Petter Langtangen, Dietmar Pfahl, Greg Wilson
2009 ICSE Workshop on Software Engineering for Computational Science and Engineering (2009-05) https://doi.org/bw966x
DOI: 10.1109/secse.2009.5069155

5. **Ten simple rules for making research software more robust**
Morgan Taschuk, Greg Wilson
PLOS Computational Biology (2017-04-13) https://doi.org/gfvpqw
DOI: 10.1371/journal.pcbi.1005412 · PMID: 28407023 · PMCID: PMC5390961

6. **ISO 25010** https://iso25000.com/index.php/en/iso-25000-standards/iso-25010?limit=3%20

7. **Empirical study on software and process quality in bioinformatics tools**
Katalin Ferenc, Konrad Otto, Francisco Gomes de Oliveira Neto, Marcela Dávila López, Jennifer Horkoff, Alexander Schliep
Cold Spring Harbor Laboratory (2022-03-13) https://doi.org/grx4jr
DOI: 10.1101/2022.03.10.483804

8. **A large-scale analysis of bioinformatics code on GitHub**
Pamela H Russell, Rachel L Johnson, Shreyas Ananthan, Benjamin Harnke, Nichole E Carlson
PLOS ONE (2018-10-31) https://doi.org/gskr8b
DOI: 10.1371/journal.pone.0205898 · PMID: 30379882 · PMCID: PMC6209220

9. **A survey of scientific software development**
Luke Nguyen-Hoan, Shayne Flint, Ramesh Sankaranarayana
Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (2010-09-16) https://doi.org/brfqvq
DOI: 10.1145/1852786.1852802

10. **CZI – Essential Open Source Software for Science**
Chan Zuckerberg Initiative
https://chanzuckerberg.com/eoss/

11. **A Software Chasm: Software Engineering and Scientific Computing**
Diane F Kelly
IEEE Software (2007-11) https://doi.org/cbrmv5
DOI: 10.1109/ms.2007.155

12. **Software Engineering Education for Bioinformatics**
Medha Umarji, Carolyn Seaman, AGunes Koru, Hongfang Liu
2009 22nd Conference on Software Engineering Education and Training (2009)
https://doi.org/b57ccg
DOI: 10.1109/cseet.2009.44

13. **Handreichung Zum Umgang Mit Forschungssoftware**
Matthias Katerbow, Georg Feulner
Zenodo (2018-02-27) https://doi.org/ghk5fk
DOI: 10.5281/zenodo.1172970

14. **Developing Scientific Software**
Judith Segal, Chris Morris
IEEE Software (2008-07) https://doi.org/bnm3xp
DOI: 10.1109/ms.2008.85

15. **Testing Scientific Software: A Systematic Literature Review**
Upulee Kanewala, James M Bieman
arXiv (2018) https://doi.org/gsrxg5
DOI: 10.48550/arxiv.1804.01954

16. **Bridging the Chasm**
Tim Storer
ACM Computing Surveys (2017-08-25) https://doi.org/gftvrn
DOI: 10.1145/3084225

17. **Hunting for the best bioscience software tool? Check this database**
Matthew Hutson
Nature (2023-01-12) https://doi.org/gsnnww
DOI: 10.1038/d41586-023-00053-w

18. **Why science needs more research software engineers**
Chris Woolston
Nature (2022-05-31) https://doi.org/gsnnwt
DOI: 10.1038/d41586-022-01516-2

19. **Ex-Google chief's venture aims to save neglected science software**
David Matthews
Nature (2022-07-13) https://doi.org/gsnnwv
DOI: 10.1038/d41586-022-01901-x

20. **High-Performance Mobile Internet**
Anirban Mahanti, Subhabrata Sen
IEEE Internet Computing (2014-01) https://doi.org/gsszgq
DOI: 10.1109/mic.2014.8

21. **A Scientist's Nightmare: Software Problem Leads to Five Retractions**
Greg Miller
Science (2006-12-22) https://doi.org/fbvb8b
DOI: 10.1126/science.314.5807.1856

22. **Sustainable data analysis with Snakemake**
Felix Mölder, Kim Philipp Jablonski, Brice Letcher, Michael B Hall, Christopher H Tomkins-Tinch, Vanessa Sochat, Jan Forster, Soohyun Lee, Sven O Twardziok, Alexander Kanitz, … Johannes Köster
F1000Research (2021-01-18) https://doi.org/gjjkwv
DOI: 10.12688/f1000research.29032.1 · PMID: 34035898 · PMCID: PMC8114187

23. **Nextflow enables reproducible computational workflows**
Paolo Di Tommaso, Maria Chatzou, Evan W Floden, Pablo Prieto Barja, Emilio Palumbo, Cedric Notredame
Nature Biotechnology (2017-04) https://doi.org/gfj52z
DOI: 10.1038/nbt.3820

24. **Anaconda | The World's Most Popular Data Science Platform**
Anaconda
https://www.anaconda.com/

25. **Home**
Docker Documentation
(2023-08-22) https://docs.docker.com/

26. **Home** https://apptainer.org/

27. **Guidelines for bioinformatics of single-cell sequencing data analysis in Alzheimer's disease: review, recommendation, implementation and application**
Minghui Wang, Won-min Song, Chen Ming, Qian Wang, Xianxiao Zhou, Peng Xu, Azra Krek, Yonejung Yoon, Lap Ho, Miranda E Orr, … Bin Zhang
Molecular Neurodegeneration (2022-03-02) https://doi.org/gptgqt
DOI: 10.1186/s13024-022-00517-z · PMID: 35236372 · PMCID: PMC8889402

28. **A Clinician's Guide to Bioinformatics for Next-Generation Sequencing**
Nicholas Bradley Larson, Ann L Oberg, Alex A Adjei, Liguo Wang
Journal of Thoracic Oncology (2023-02) https://doi.org/gsm3mb
DOI: 10.1016/j.jtho.2022.11.006 · PMID: 36379355 · PMCID: PMC9870988

29. **Practice guidelines for development and validation of software, with particular focus on bioinformatics pipelines for processing NGS data in clinical diagnostic laboratories**
Nicola Whiffin, Kim Brugger, Joo Wook Ahn
PeerJ (2017-05-29) https://doi.org/gsm3mc
DOI: 10.7287/peerj.preprints.2996

30. **Standards and Guidelines for Validating Next-Generation Sequencing Bioinformatics Pipelines**
Somak Roy, Christopher Coldren, Arivarasan Karunamurthy, Nefize S Kip, Eric W Klee, Stephen E Lincoln, Annette Leon, Mrudula Pullambhatla, Robyn L Temple-Smolkin, Karl V Voelkerding, … Alexis B Carter
The Journal of Molecular Diagnostics (2018-01) https://doi.org/gcsstd
DOI: 10.1016/j.jmoldx.2017.11.003

31. **Top considerations for creating bioinformatics software documentation**
Mehran Karimzadeh, Michael M Hoffman
Briefings in Bioinformatics (2017-01-14) https://doi.org/bzmp
DOI: 10.1093/bib/bbw134 · PMID: 28088754 · PMCID: PMC6054259

32. **Ten simple rules for getting started with command-line bioinformatics**
Parice A Brandies, Carolyn J Hogg

PLOS Computational Biology (2021-02-18) https://doi.org/gh32h2
DOI: 10.1371/journal.pcbi.1008645 · PMID: 33600404 · PMCID: PMC7891784

33. **Practice guidelines for development and validation of software, with particular focus on bioinformatics pipelines for processing NGS data in clinical diagnostic laboratories**
Nicola Whiffin, Kim Brugger, Joo Wook Ahn
PeerJ (2017-05-29) https://doi.org/gsm3md
DOI: 10.7287/peerj.preprints.2996v1

34. https://faculty.washington.edu/ajko/books/cooperative-software-development/

35. **Studying the impact of social interactions on software quality**
Nicolas Bettenburg, Ahmed E Hassan
Empirical Software Engineering (2012-04-28) https://doi.org/f4mhdp
DOI: 10.1007/s10664-012-9205-0

36. **Ten simple rules to increase computational skills among biologists with Code Clubs**
Ada K Hagan, Nicholas A Lesniak, Marcy J Balunas, Lucas Bishop, William L Close, Matthew D Doherty, Amanda G Elmore, Kaitlin J Flynn, Geoffrey D Hannigan, Charlie C Koumpouras, … Patrick D Schloss
PLOS Computational Biology (2020-08-27) https://doi.org/gg92xw
DOI: 10.1371/journal.pcbi.1008119 · PMID: 32853198 · PMCID: PMC7451508

37. **What Is Software Project Management?** https://www.wrike.com/project-management-guide/faq/what-is-software-project-management/

38. **Agile and the Long Crisis of Software**
https://www.facebook.com/logicisamagazine
Logic(s) Magazine https://logicmag.io/clouds/agile-and-the-long-crisis-of-software/

39. **Manifesto for Agile Software Development** https://agilemanifesto.org/

40. **A teamwork model for understanding an agile team: A case study of a Scrum project**
Nils Brede Moe, Torgeir Dingsøyr, Tore Dybå
Information and Software Technology (2010-05) https://doi.org/cvr88b
DOI: 10.1016/j.infsof.2009.11.004

41. **Communication in Transactive Memory Systems: A Review and Multidimensional Network Perspective**
Bei Yan, Andrea B Hollingshead, Kristen S Alexander, Ignacio Cruz, Sonia Jawaid Shaikh
Small Group Research (2020-12-11) https://doi.org/ghpwvf
DOI: 10.1177/1046496420967764

42. **Walking the Talk: Adopting and Adapting Sustainable Scientific Software Development processes in a Small Biology Lab**
Michael R Crusoe, CTitus Brown
Journal of Open Research Software (2016-11-29) https://doi.org/gsmb78
DOI: 10.5334/jors.35 · PMID: 27942385 · PMCID: PMC5142744

43. **The pragmatic programmer: from journeyman to master**
Andrew Hunt, David Thomas
Addison-Wesley (2000)
ISBN: 9780201616224

44. **The Complete Guide to Ad hoc Testing**
BrowserStack

https://browserstack.wpengine.com/guide/adhoc-testing/