Improving software quality in bioinformatics through teamwork

This manuscript (permalink) was automatically generated from ferenckata/SQSeminarPaper@61caa50 on December 22, 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

Ever since the high-throughput techniques became a staple in science laboratories, the generation of computational algorithms and scientific software boomed. However, more recently it has been noted that scientific software, more specifically bioinformatics software, lacks the quality standards of software development. The consequence of this is code hard to test (or independently verify), reuse, and maintain. We believe the root of inefficiency in implementing the best software development practices in the academic settings is the individualistic approach. Software development is a collective effort in most software-heavy endeavours. The literature suggests that team work directly impacts code quality through knowledge sharing, redundancy, and standards. In our computational biology research groups, we explored ways to sustainably involve all group members in learning, sharing, and discussing software, while maintaining the personal ownership of research projects and related software products. We found that through weekly meetings, within a year, regular members improved their coding skills, became more efficient bioinformaticians, and obtained a detailed knowledge about the work of their peers. Through within-group knowledge transfer, without investing significant amount of time, each member obtained knowledge about advanced concepts, can now quickly identify and access the expertise of each other, and established standards to which new members are also required to comply. We advocate for improvement of software development culture within bioinformatics through local collective effort, while hoping that top-down approaches such as publication requirements for code become standard in the near future.

Introduction

Bioinformatics and computational biology are continuously gaining importance in biological research. About 90% of researchers rely on results produced by scientific software [1]. In turn, scientists are heavily relying on inventions of computer science and software engineering, such as programming languages, programming paradigms, or container solutions. However, adopting practices from other fields is not without difficulties and scientific software development tend to lag behind. One implication of using outdated or poor software engineering practices is that incorrect software results in invalid scientific findings [1,2]. Beyond that, even when the software performs as intended, researchers spend significant amount of time on software building using suboptimal practices which can further increase the necessary time investment in the future [3,4,5].

Good software development practices have been established in other software-heavy endeavours to mitigate the risk of incorrect software solutions. However, bioinformaticians or more generally scientists working with scientific software often lack formal education in computer science or software development [2,6,7]. The lack of theoretical and practical foundations hinders the adoption of good coding practices (e.g. unit tests, continuous integration, code reviews). To prioritize good practices, it is beneficial to know about the target qualities a software may have. Although the software engineering community has defined software quality attributes [8], these are largely unknown to practising bioinformaticians. Among these attributes are functional suitability and performance, which are implicitly prioritized within bioinformatics. On the other hand, the list also contains usability, reliability, maintainability, and portability, which are implicitly neglected in most bioinformatics endeavour. The consequences of this implicit prioritization is that the default behaviour (i.e. producing prototype software) does not change even when the scope of the software product changes.

Beyond the limits of individual education, many of the good software practices rely on redundancy of knowledge within team members, supported by practices such as pair programming, regular stand-up meetings, and code reviews. In contrast, historically, academic research projects are mostly driven by a single person (a PhD student or a post-doc) and these projects are often part of the academic

degree evaluation. Unfortunately for software development purposes, the academic world still often tends to disregard the team effort, priding itself on enabling individual achievements and career progression instead. Taking this into account, with one (or very few) person developing the software for a project, scientific software remains poorly maintained even if it is used by a significant number of researchers worldwide [7,9,10,11].

The concept of a team is therefore different in a research-oriented project compared to a software development project. While the group members help each other with scientific suggestions, most often there is a single person responsible for the design and implementation of the code base. As official guidelines on coding practices are rarely definitive, but rather suggestive, the actual craft of software engineering is treated as secondary task and is often up to individual judgment. These guidelines may naturally emerge in larger groups, if software is used or even developed by multiple group members. It was reported previously that researchers tended to rank software engineering concepts higher if they worked in a team [6]. High-profile code bases often feature larger development teams and their activities are marked with longer commit messaged indicating better communication and documentation of the software [10]. However, it is not always obvious that to follow the guidelines requires a form of team organization not intrinsic to academic groups. We hypothesize that a form of team structure organized around individual software products could improve the quality of our scientific code. Specifically, the development and maintenance of scientific software can be improved by taking into account good software practices. This can be implemented in research groups by creating a teamwork atmosphere where trainees and staff are directly or indirectly working on the same software.

In this work we start with a literature review on the concepts we build on. First, we present guidelines suggested by both software engineers evaluating the computational scientists (emphasizing, but not limiting to bioinformatics), and bioinformaticians. Thus, we obtain an external and internal view on the envisioned standards and priorities for the bioinformatics community. Next, we explore the main aspect of team management established in non-academic software development settings. Finally, we describe and discuss how our research groups, motivated by personal experience of working with scientific software and inspired by the literature, created weekly sessions to discuss and learn different aspects of software quality relevant for computational biology.

We find that good practices require investment in time and effort that may not be feasible to fulfil as an individual (e.g. having limited time to perform and to deliver doctoral thesis). However, we suggest that our practices results in shared standards and an overall better code quality of the members with a reduced effort on an individual level. These practices include sharing the existing software knowledge base in the group, learning new tool development and implementation together, and normalizing showing and discussing code. We thus aim to provide a motivation and framework on how to get started with collective software development by directly or indirectly involving all bioinformatician group members, with or without formal training in software engineering.

Guidelines and practices of bioinformatics as seen by software engineers and bioinformaticians

The internet is full of learning and support material for developing or working with software products. Since bioinformaticians are often self-taught programmers and only a small fraction have formal training in computer science and software engineering, these recourses become vital. They include blog posts from peers, freely available lecture materials from universities, forums or articles that propose guidelines on how to code or analyse data in a better way. The encouraged practices are plenty and vary a lot, therefore the code produced by bioinformaticians lack a certain standard. The scientific software status has been analysed by the software engineer community, where certain

limitations and caveats were identified and discussed. We provide an overview of the available literature on these discussion points.

First, let us summarize the main themes in papers with guidelines for bioinformaticians. These articles would be the entry point for bioinformatician who aim to improve their programming skills. We used several phrases to search for papers: “guidelines for bioinformatics software”, and “rules for biologists learning bioinformatics”. Selected papers focus on specific suggestions, often referred to as rules or “tips & tricks”, or they more broadly direct the readers towards good practices of coding, which are put together into guidelines. Specialized topics include particular data analysis in a single disease [12], while broad themes span from next-generation sequencing (NGS) data analysis to outlining tips on how to start on computational analysis of the experimental data [13,14,15]. 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. The guideline papers tend to target early career researchers with minimal coding experience (e.g. first time terminal users), while also encourage the usage of state-of-the-art software solutions (e.g. containers). Therefore, the guidelines might be a mix of basic and advanced concepts, especially from the perspective of a standard computer science and software engineering curriculum. For example, documentation and version control are most commonly highlighted [16,17]. However, instructions and tips on how to make your documentation and code up to software quality standards are usually limited, possibly due to the short nature of these guidelines and the lack of sufficient background of the assumed readers.

Next, we performed a second round of literature review to obtain an external view on the status 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.

It is apparent from the literature that scientific software is not up to software engineering standards. Already almost two decades ago, Diane F. Kelly wrote that scientific computations keep on being performed using error-prone development practices and reaching suboptimal solutions and poor software quality due to lack of appropriate software engineering practices [18]. Therefore, the software engineering community also writes guidelines on how these practices should be followed after surveying the current state of software in scientific community and specifically the bioinformatics community [1,3,6,11,19]. In addition, an extensive literature review has been published recently in which known issues and suggested solutions are collected [2]. We collected these recommendations into Table 1.

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
standardized tests	[2,3,7,19,20]
version control	[2,3,7,20]
user (and developer) documentation	[2,7,19,20]
independent review of source code	[1,19,20]
standardized working environment and automation	[2,20]
licensing	[7,20]
requirements gathering	[2,19]

Recommendation	Source
containerization for portability	[2,7]
reuse existing (reliable) software	[3,7]
agile software development methodology	[2,3]
educated choice of software development methodology	[19]
adoption of international best practice standards of software quality	[20]
establish validation and acceptance procedures	[20]
cooperation between developers and users	[20]
description of the software version used, its configurations and parameters in publications	[20]
preferentially selecting freely available open-source software	[20]
encourage user participation in the software development process	[20]
tagging of software version for reproducibility	[7]
sanity check on input parameters	[7]
do not hard-code changeable parameters and paths	[7]
rely on package managers	[7]
do not require superuser privileges	[7]
provide a small test set	[7]
ensure reproducibility of results	[7]
refactoring	[3]
usage of design patterns	[3]
quality monitoring (e.g. SonarQube)	[3]
continuous integration	[3]
contribute to open-source development	[1]
recognition and assignment of adequate time for quality-assured development	[19,20]
recognition of software development as academic achievement	[1,20]
financial support for software development and maintenance	[1,20]
support for developer community for long term maintenance (when applicable)	[1,20]

The first impression Table 1 might give is being intimidated by the sheer amount of recommendations. It is unrealistic to expect that a bioinformatician on temporary contract, working towards publishing, without formal training in computer science, or institutional support would be able to gain a good understanding and practice in all of them. Beyond understanding, Arvanitou et al. note that a scientific software developer, deepending on the application of the software (e.g. whether it is a tool or a data analysis pipeline), needs to prioritize the software quality attributes to make choices among the good practices [3]. An issue might be that bioinformaticians are rarely familiar with the meaning and importance of software quality attributes. furthermore, among the recommended practices are agile software development, the DRY (don't repeat yourself) principle, requirements gathering and unit testing. We noted that these concepts are not intuitive, well known in

the bioinformatics community, or even trivial to adapt to the bioinformatics software development environment [22]. Therefore, it is not surprising, that the guidelines from software engineers are struggling to penetrate the bioinformatics community [2]. However, it is self-evident that the challenges in adopting good software development practices cannot be an excuse for skipping them. In the coming paragraphs we highlight some insights about the recommended concepts and practices from the literature review.

The hardship of scientific software testing has been discussed in detail [1,21,22]. Globe emphasized the importance of software testing with an analogy, comparing it to the importance of testing the functionality of a microscope, which is self-evident to all researchers [1]. In a recent review paper [22] two key aspects of scientific software testing have been highlighted: the oracle problem and the cultural differences between scientists and software engineers. First, 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. Second, 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 [23].

Another insight is about the complexity of bioinformatics software. In bioinformatics analysis it is common to combine the functionalities that are coming from various packages. This has several implications [1,3,7,9], here we highlight a few of them. First, over time the software becomes increasingly hard to maintain. The complexity, size, age, and the change-proneness of a code heavily affect maintainability [19]. However, as bioinformatics software developers view their code as “means to an end”, they care less about the future of their software. Second, package management (including versioning) is a crucial aspect to ensure not only maintenance, but also ease of development, reproducibility, and reusability. Frameworks [24,25] and package management solutions [26,27,28] are required to achieve these qualities. Third, it is practically impossible to test all functionalities of all modules, and the combinations of various functionalities. It is therefore instrumental that the developers of the modules are trustworthy and responsible in their development.

A recurring question is whether a script needs refactoring or can remain a prototype. Taschuk and Wilson [7] suggest a cut-off where a script is being reused, shared with others or used to produce findings in a publication. This definition would potentially include the majority of code written by bioinformaticians, but the time spent on improving the scripts should be weighed against the time required to deal with suboptimal code. Overall, as good practices become routine, the required time investment will be reduced and the benefits will become more apparent.

Finally, throughout our literature review we found only one instance of suggestions on how to code in a team setting and utilize multiple people’s expertise on software development. Often guidelines for starting bioinformaticians encourage reaching out to others, but mostly to seek help when encountering a problem with their code. This could include consulting with colleagues, finding a mentor or participating in online communities (for example, Stack Overflow or Biostars) [17]. However, it is still mainly focused on individual practices, does not involve peer-pressure, and insufficient to recognize unknown unknowns. The one counter example is the Code Clubs described by Hagan et al. [29]. In their research group members are collectively engaged in software development through code reviews and pair coding and software engineering education through workshops or seminars [29]. It is in contrary to software engineering-oriented literature, where the main focus is on practices when coding in a team [30,31]. Sharing your coding experience with others helps minimize the isolation, allows individuals to learn from their peers, helps to establish and maintain standards, and helps to write a better quality software.

Coding in team

Beyond the brief mention of getting support or coding in a team in guidelines for bioinformaticians, specialized literature exists that examines how to effectively organize coding activities in a team. Programming as a collective practice is a key notion in software engineering. A central theme in this literature is maximizing team cohesion while minimizing code coupling [32]. Authors argue that the viability of a software project along it successive development phases is largely determined by the adoption of sound software design enforcing modularity and extensibility coupled with team management practices centred around communication and collective governance [33].

In general, 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 [34]. Typically, the oversight of these functions would be taken up by a single individual referred to as the "manager" of a project, where manager is a role rather than a title of a particular person. In the particular context of computational projects in academia, a strict division of labour is rarely found in regard to the management of software projects. This means that the management and execution of a software project falls into the hands of the same person. Furthermore, some tasks, such as risk, budget and time management, and maintenance are discussed at the conception of the project (e.g. during grant application) and thus decoupled from the actual software development phase. The remaining management tasks would often be deliberated by the developer(s), eventually, and often implicitly, reaching a consensus on the desired way forward and acted upon.

This sort of self-management, at times collective, echoes some prescriptions of the SCRUM method [35]. 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 [36]. This proposition was a reaction to the typical blueprint-like management for engineering projects which proved ineffective in addressing the emerging challenges of large software projects [35]. The similarity is probably the reason why agile practices are part of guidelines for scientific software developers (Table 1). As agile is the only recommendation about team management present in these guidelines, we discuss it here in detail.

Within the context of the software industry, one outstanding aim of agile is the aspiration for more autonomy for organizing 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. Incentivizing 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 [37]. 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 difference parts of the project [32]. 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 create a better mutual awareness of team member expertise. This mutual awareness is known as transactive memory system, and has been linked to increased team performance [38]. Taken together these merits further improve the team's capacity to overcome technical challenges that will arise over the course of the development process.

To benefit from agile-like practices, the implementation and effective adoption of this mode of project management is required. In turn, it relies on the execution of a variety of methods whose success in realizing 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 collegial work culture and standardized procedures. At their core, these practices incentivize continuous communication and collective decision-making among developers. This constitutes an additional overhead in terms of time and resources needed when developing, but this is offset by the aforementioned benefits in terms of coding practice, software resilience and improved team capabilities.

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” [35]. In academic software development the agile practices would in fact be a more stringent approach than current practices. For example, it would include writing down requirements in form of user stories, plan a minimal viable product, and divide the project into tasks. This is important, because most literature pictures agile as management style free from traditional management. The agile system assumes that software engineering professionals seek to find the best approaches, and are well-equipped to make good decisions on their own - when faced with shifting requirements and complex code base [35]. Additionally, the lack of top-to-bottom management, architects and system analysts in academic software development put even more responsibility on the individual developer. As noted in previous sections, current scientific software developer education does not necessarily cover these elements [19].

We do not believe that all the software engineering guidelines employed in the industry are necessarily relevant to the production of scientific software. The circumstances differ significantly, mainly due to how the outcomes of research projects (papers, tools, protocoles, etc) need to be credited to paricular individual researchers for their career progression. Regardless of the optimality of this situation, personal projects remain the norm, and it would be futile to expect another group member to achieve an equal level of familiarity with one’s project. However, this should not prevent interactions between the people in the group, as it is through these interactions that rules are enforced and quality increased.

In our research groups, we have practically implemented the environment in which we, as a group, learn about and implement software quality practices that have been discussed in literature. We want to share this experience and propose how simple additions, such as weekly code review sessions or seminars, can lead to improved quality collective or personal software.

Our experiences for development processes involving teams

In our professional careers, we have experienced hardships with scientific software - both from the user and from the developer’s perspective. We have seen a variety of suggestions in the literature aiming to improve the status of bioinformatics software. We recognized that for a single person achieving a good understanding of them all, and subsequently prioritizing, and adopting them would require a substantial amount of time. Paradoxically, researchers in academia, especially trainees, often work under time pressure, since projects, such as doctoral thesis, have pressing deadlines. Even basic software quality standards (e.g. standardized environment, independent review of source code) might seem out of scope and impossible to implement for a single researcher. On the other hand, we also have seen that the industry standard approach heavily relies on a team structure and team management. Therefore, within our groups, we aimed to create a system where the individual scientific software projects are supported through collective learning, understanding, and discussions. In this section we describe the practices that we have settled on.

The software development practices that we have adopted can be broadly separated into three categories: 1 - what we have called software quality meetings, 2 - code reviews, and 3 - resource sharing. We intend these as an exemplary approach for other computational biology groups or institutes of 3 or more bioinformaticians. However, given the abundance of opinions on this topic, and

the variety of challenges bioinformaticians face, we believe that each group should find out what works best for them.

Software quality meetings

Within the framework of software quality meetings, we have established a large-scale knowledge transfer system between the participants. Presentations and demonstrations of basic concepts, 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 literature recommendations, previous education, own projects, code reviews, 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. We recommend keeping these meetings regular, e.g. at least once a month, given the amount of knowledge that can be learnt together (see Table 1).

The outcomes of these sessions are manifold. A few examples:

1. a shared vocabulary that enables quick discussion about implementation details and code structures (e.g. object-oriented programming, design patterns, data structures and algorithms);
2. awareness of previously unknown packages or technical solutions, improving software performance and quality (e.g. bioframe, S4 object system, R Markdown);
3. a kind of toolkit and set of recordings we can sample from and build on in our own research projects (e.g. containerization, git features to ease and quicken software development, planning with UML diagrams).

During the software quality meetings, we have also explored the possibility of collaborative projects and pair programming. We experimented with collaboration on different software tools that are available for all members of the research group and developed by multiple people in the group (see section on Resource sharing). However, this activity showed to require an additional amount of time we can rarely spare. A project, where we extensively applied software quality features (object-oriented programming style, user stories when documenting the requirements and assumptions, Jira to add features and report bugs, continuous integration with Git) when working on the same collection code as a team was the latest release of JASPAR database [39]. From this project experience we concluded that continuous integration is yet to be conquered on bigger software project, but code reviews turned out to be a driver of efficient and quality software development. However, we want to note that this article in fact was successfully written using a continuous integration based tool Manubot [40].

Code reviews

The benefits of code reviews have been reviewed in the past [29,35,41]. In this text we will briefly summarize how presenting your code and receiving feedback leads to improvement in the process of creating software. We found that during these meetings implicit peer-pressure helps us achieve most goals: standardization of practices, improved code quality, and enhanced usability of the software. We would like to note, that the efficiency of these meetings are improved with a shared understanding of the concepts covered during the software quality meetings. Therefore, we advise starting with learning before discussing the code.

Prior to a scheduled code review, the author is expected to write their code in a way that it will be explainable and understood by others. This expectation is largely self-inflicted as each person feel the pressure of exposing their weaknesses - even within a friendly environment. In a large distributed project clean coding style may be trivial, but because the bioinformatic projects are often handled by

a single person, it is very possible to make the code complex and obfuscated. We observed that during data analysis parts of the code are re-run in an ad-hoc manner (e.g. by commenting out or re-writing parts), making it increasingly difficult to explain the code or 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, as with the rubber duck method [42]. 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. [29], and note that many of those naturally emerged as a code of conduct after a few rounds of trial and error. In our settings, it is entirely up to the author to choose which aspect of the code, or software product to discuss. Although it is implied that participants of code reviews are intended to discuss implementation details, we accept and enjoy discussions about any other aspect of the code, such as user interface design, documentation, or architecture considerations.

After the review, the received suggestions, if crucial, should be implemented swiftly to improve the code before advancing the project. Some other suggestions (e.g. coding style) do not require instant refactoring, these may be viewed as suggestions for future projects. At the start of implementation of regular meetings, the recurring comments were about modularization, documentation, and variable declarations, until these became standard among the members. For example, after about half a year, it was trivial for everyone involved that code organized into functions is preferred over the so-called "spaghetti code". It is important to note that the success of code review is highly dependent on its frequency. A long time between reviews means a lot of new code, difficulty to cover all changes in a single session, and potentially a lot of rewrite post review. Our group of 5-10 people settled for weekly code review sessions.

Our experience indicates a broader adoption of notions and good software engineering practices highlighted during these code review sessions. A couple examples will illustrate how code reviews incentivized 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. Recurrently we would examine the best strategies to modularize the presented code and discuss what would constitute 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 behaviour of the considered tool, which facilitates its intelligibility for a wider audience. The shared knowledge base and standards also allow us to make new group members adopt good coding practices more quickly.

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, where we would naturally discuss each other's scientific projects. Additionally, seeing and analysing everyone's code on a more hands-on level showed us how repetitive some pieces of code can be in different projects. This redundancy can be removed by implementing a system to share resources.

Resource sharing

Resource sharing boils down to making sure that useful online resources are brought to the attention of all participants easily. It can 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 input, 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. We aim to include this knowledge in the upcoming events when a new tool is added to the shared repository.

In sum, software quality meetings, code reviews and shared resources in the research group can be implemented as separate activities choosing all or any of them. We observed that even a single activity is benefiting members' coding experience and the resulting code quality.

Conclusions and future perspectives

Software engineering emerged and has been developing to address issues naturally arising from poorly planned software development, such as project failures, delays, incorrect functionality or defects [43], none of which is unknown to the scientific community. Indeed, the crisis of scientific software in general is widely discussed [44,45]. It is only natural that the bioinformatics community learns from those more experienced, solving problems that have been identified. In this case, it is both the software engineering research community and the industry experts on software and team management.

In our computational biology groups, we introduced regular seminars to learn about software solutions, and code reviews that fit our specific needs and context. Through these meetings, we learnt about and adopted various concepts that achieve a better quality software, with a special focus on reliability, performance and extensibility. Furthermore, we have established coding standards within our groups, which ease within-group support and collaborative projects. 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.

When discussing our approach, it is implied that team dynamic is important, especially for such bottom-up approaches. The overall performance increases when team members are familiar with each other and build problem-solving routines together through cumulative experience [32]. In a group the knowledge on who knows what speeds up the problem-solving [38]; time spent together and social factors ease technical knowledge transfer [32]. We therefore motivate group leaders of groups with even a small computational component to build an environment for their trainees to communicate and discuss software quality aspects.

Working in teams is not an option, but a must for large projects which support thousands of reserachers world-wide, and contribute to novel findings. Although it is not necessary in smaller projects, the benefits are significant. All software projects start as small prototype-like software. Then

they may be abandoned by the original developer. They could also technically survive the original developer, deposited on platforms like GitHub, but they might be overly cryptic and poorly documented so that no other scientist can take over [46]. 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 maturation, addition of new features, and general maintainability. Thus, they may prevent a smart solution to be used and reused over time.

In order to put all we have discussed into an illustration, think about the exercise of rock-climbing (Figure 1). At the top of the rock is our goal of good quality software. Specifically, we identified reliable, performant, and extensible software as our aim. In order to reach it, we need to become proficient in the various concepts depicted by the holds. The higher they are on the wall, the more advanced we consider the concepts to be. As the progress is gradual, we have chosen to show the holds in the same colour if they represent related concepts that build upon each other. This way, we mimic traditional CS education, compared to the guidelines of a mixture of concepts. The most important point, however, is the fact that rock climbing requires a partner to belay you, just as we believe the input of other people helps us become better programmers.


Figure 1: An illustration comparing the process of improvement in software writing to rock climbing. DSA: data structures and algorithms, OOP: object-oriented programming, UML: Unified Modelling Language, CI: continuous integration, SCA: static code analysis

Figure 1: An illustration comparing the process of improvement in software writing to rock climbing. DSA: data structures and algorithms, OOP: object-oriented programming, UML: Unified Modelling Language, CI: continuous integration, SCA: static code analysis

We envision a future where scientific software for core applications is appreciated, reliable, and actively maintained. All scientists would benefit from a strong backbone of software solutions, that would support quick and efficient prototyping, as well as maturation of working solutions. The lack of funding for the maintenance of software, prevents achieving a level of software quality that would inspire confidence in the results [1]. Funding is typically provided for the development of novel software, and it can be hard to justify spending time on maintenance which provides no output in terms of articles. Currently, as Alexander Szalay puts it “the funding stops when they [researchers] actually develop the software prototype” [46]. Researchers want to build on each other’s findings, use published novel software as tools, but they might need to spend quite some time adopting or maintaining that software [1,3,7]. The infrastructure would benefit from funding earmarked for maintenance, and from dedicating time to it in project proposals. Fortunately in recent years, the lack of funding is being recognized and addressed by a few agencies, such as the Chan Zuckerberg Initiative Essential Open Source Software for Science fund [47]. Scientific community and funding agencies should welcome the efforts of maintaining original software and encourage its updates instead of the development of a replacement software that risks remaining unmaintained.

To summarise, today it is important for the scientific community to recognize the limitations of the software we are producing. This includes acknowledging the flaws in the process of coding. As a potential great improvement we propose organizing activities, such as software quality and code review meeting, that would involve the whole research group in each other’s projects, therefore allowing the sharing of knowledge and feedback on the code practically. We also advocate for sustainable funding for the maintenance of existing and newly developed scientific software.

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. https://c2.com/doc/oopsla92.html

5. **Managing technical debt**
   Eric Allman
   Communications of the ACM (2012-05) https://doi.org/grx4cv
   DOI: 10.1145/2160718.2160733

6. **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

7. **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

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

9. **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

10. **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

11. **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

12. **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

13. **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

14. **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

15. **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

16. **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

17. **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

18. **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

19. **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

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

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

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

23. **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

24. **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

25. **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

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

27. **Home**
Docker Documentation
(2023-12-04) https://docs.docker.com/

28. **Apptainer - Portable, Reproducible Containers** https://apptainer.org/

29. **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

30. **Cooperative Software Development** https://faculty.washington.edu/ajko/books/cooperative-software-development

31. **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

32. **Team Familiarity, Role Experience, and Performance: Evidence from Indian Software Services**
Robert S Huckman, Bradley R Staats, David M Upton

Management Science (2009) https://www.jstor.org/stable/40539129

33. **The psychology of computer programming**
Gerald M Weinberg
Dorset House Pub (1998)
ISBN: 9780932633422

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

35. **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/

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

37. **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

38. **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

39. **JASPAR 2024: 20th anniversary of the open-access database of transcription factor binding profiles**
Ieva Rauluseviciute, Rafael Riudavets-Puig, Romain Blanc-Mathieu, Jaime A Castro-Mondragon, Katalin Ferenc, Vipin Kumar, Roza Berhanu Lemma, Jérémy Lucas, Jeanne Chèneby, Damir Baranasic, … Anthony Mathelier
Nucleic Acids Research (2023-11-14) https://doi.org/gs4n8b
DOI: 10.1093/nar/gkad1059

40. **Open collaborative writing with Manubot**
Daniel S Himmelstein, Vincent Rubinetti, David R Slochower, Dongbo Hu, Venkat S Malladi, Casey S Greene, Anthony Gitter
PLOS Computational Biology (2019-06-24) https://doi.org/c7np
DOI: 10.1371/journal.pcbi.1007128 · PMID: 31233491 · PMCID: PMC6611653

41. **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

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

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

44. **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

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

46. **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

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