Improving software quality in bioinformatics through teamwork

This manuscript was automatically generated on April 25, 2024.

Authors

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

- **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

- **Marieke Kuijjer**
  0000-0001-6280-3130
  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; Centre for Bioinformatics, Faculty of Mathematics and Natural Sciences, University of Oslo, Ooslo, 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

Since high-throughput techniques became a staple in science laboratories, computational algorithms and scientific software boomed. However, scientific software, specifically bioinformatics software, usually lacks software development quality standards. It results in software code that is hard to test (or independently verify), reuse, and maintain. We believe the root of inefficiency in implementing the best software development practices in academic settings is the individualistic approach, which has traditionally been the norm for recognizing scientific achievements and, by extension, for developing specialized software. Software development is a collective effort in most software-heavy endeavors. Indeed, the literature suggests teamwork directly impacts code quality through knowledge sharing, collective software development, and established coding standards. In our computational biology research groups, we explored ways to sustainably involve all group members in learning, sharing, and discussing software development 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 detailed knowledge about the work of their peers, triggering new collaborative projects. Each member learned about advanced concepts through within-group knowledge transfer without investing significant time. We can now quickly identify and access each other's expertise and have established standards to which new members must comply. We strongly advocate for improving software development culture within bioinformatics through local collective effort in computational biology groups or institutes with three or more bioinformaticians. This manuscript provides the necessary overview of the best coding practice literature. It describes how we improved our software development culture through three pillars: (1) software quality seminars, (2) code reviews, and (3) resource sharing. We hope that our shared experience and suggestions will boost the implementation of similar initiatives for improved academic training and a more sustainable environment for academic software practice.

Introduction

Bioinformatics and computational biology are indispensable and integral components of research in biology. About 90% of researchers rely on results produced by scientific software [1]. In turn, scientists heavily rely on computer science and software engineering innovations, such as programming languages, programming paradigms, or container solutions. However, adopting practices from other fields is complex, and scientific software development in the academic setting tends to lag. One implication of using outdated or poor software engineering practices is that incorrect software may result in invalid scientific findings [1,2]. Even when the software performs as intended, researchers can spend significant time on software building using suboptimal practices. Such practice results in the accumulation of technical debt, which translates to increased future time investments in extra efforts, refactoring, and rework [3,4,5].

Good software development practices (e.g., pair programming and code reviews) are established in other software-heavy fields to mitigate the risk of incorrect software solutions and save development time. However, scientists, such as bioinformaticians, who work with scientific software often lack formal education in computer science and software development [2,6,7]. This lack of or limited training hinders the adoption of good coding practices. Moreover, the current funding and academic evaluation frameworks often restrict research projects to being primarily driven by a single trainee. As a consequence, the limitations of an individual's skills restrict academic software development, which often fails to follow software development guidelines and remains poorly maintained beyond the end of the project [7,8,9,10]. One way to expand the knowledge and application of good software quality practices is to rely on other experts and leverage complementary knowledge. We suggest that

practices from the field of software development, such as code reviews, can be repurposed as learning opportunities.

In research-oriented environments, what defines a “team” is perceived differently than in software development. Group members generally discuss and help each other with scientific suggestions. Still, a single person often drives the code base’s design and implementation process to address specific scientific questions. Since adhering to or disregarding software development guidelines typically depends on personal judgment, proper software engineering practice is often considered secondary. However, when multiple group members develop software, researchers tend to appreciate software engineering concepts [6]. Notably, larger development teams often enhance communication and software documentation, which are crucial to the success of high-profile code bases. [9]. The systematic adoption of team coding practices homogenizes the software engineering competence of individuals across the research group and contributes to the dynamism of the research environment. To summarize, organizing individual bioinformatics in a team structure results in increased validity and reproducibility of scientific findings and improved maintenance of the computational resources for the community [11,12].

This paper first reviews relevant literature on the individual and team coding practices suggested within and outside scientific research groups. To overcome the obstacles limiting researchers from adopting good practices, we present our groups’ approaches, where we learn, teach, and apply concepts to improve the quality of our software products in a team setting. We created weekly meetings where group members discuss aspects of software quality relevant to computational biology. Code review sessions complement these meetings to discuss and review the code of our peers. We suggest that our team-based activities result in shared standards and an overall better code quality with a reduced effort on an individual level. Furthermore, we offer a framework for initiating collective software development involving members of a bioinformatics group, with and without formal training in software engineering.

We emphasize that collaborative efforts provide a suitable environment to improve software quality in an academic setting. More specifically, the discussions and reflections we developed in those settings accelerated the adoption of software engineering skills in our teams. We draw a visual metaphor where improving software quality is similar to a rock climbing exercise, with the top of the rock representing our goal of good quality software (**Figure 1**). To reach this goal, one needs to become proficient in the various concepts depicted by the holds. These concepts are selected from the literature and our professional experience but are not exhaustive, and each group can tailor them to their specific needs. The higher they are on the wall, the more advanced we consider the concepts to be. As the progress is cumulative, we show the holds representing related concepts that build upon each other in the same color. This way, we mimic traditional computer science education. The order of visiting the topics can vary between groups, but reiterating certain core concepts (e.g., modularization and testing) is valuable. Nevertheless, the most crucial point is that rock climbing requires a partner to belay you, just as we believe that other people’s input helps us become better programmers.

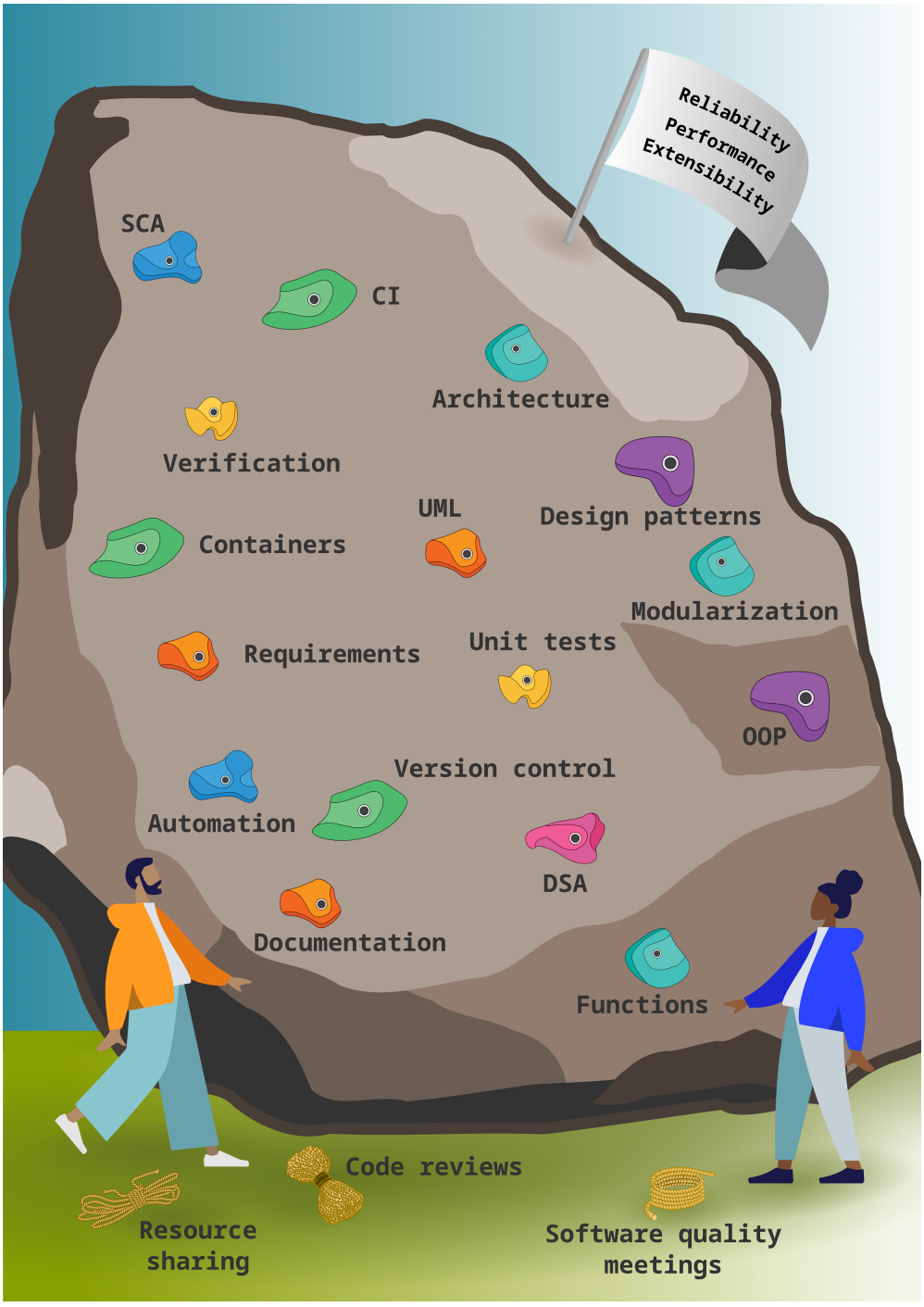

Figure 1: **An illustration comparing the improvement process in software writing to rock climbing.** DSA: data structures and algorithms, OOP: object-oriented programming, UML: Unified Modelling Language, CI: continuous integration, SCA: static code analysis

Overview of suggested coding practices

Bioinformaticians often use error-prone development practices to create software or pipelines. This practice leads to poor quality and solutions that fall short of ideal due to the insufficient application of proper software engineering methods [13]. In the past two decades, software engineering researchers have surveyed and discussed the limitations and caveats of scientific software development practices and products (e.g., see [1,3,6,10,14]). Moreover, online learning and support resources are vital for bioinformaticians who are self-taught programmers. These include peer blog posts, open-source lecture materials from universities, forums, and published guidelines for improved coding and data analysis. The encouraged practices are plenty. However, the current coding practices relevant to bioinformatics software development vary and depend on factors such as prior training in computer

science and specific scientific research fields. Finally, the suggested practices do not necessarily include a consistent view in line with mainstream software standards.

Thus, we selected articles as entry points for bioinformaticians who aim to improve their programming skills. We provide an overview of the suggestions presented in these papers in **Table 1** (**Supplementary Methods**). While this type of article usually targets early career researchers with minimal coding experience (e.g., first-time terminal users), they often encourage using state-of-the-art software solutions (e.g., containers), which seem contradictory. Therefore, these guidelines represent a mix of primary and advanced concepts usually provided as lists of rules or "tips & tricks." The large spectrum of guidelines to follow highlights the unique challenges emerging in bioinformatics, even for routine pipelines and software development for data analysis.

Table 1 might initially intimidate due to the extensive list of recommendations. This complexity reflects how the bioinformatics community struggles to implement these guidelines [2]. Recognizing that the widespread adoption of these standards will prove complicated through individual efforts, we considered a more collaborative strategy. Indeed, we aimed to revise our techniques and methods to ensure the successful integration of these guidelines into our practices. More specifically, our experience indicated a greater likelihood of adopting these critical practices and working collectively towards better software engineering proficiency.

Updating software development practices or gaining a good understanding of new concepts is not trivial. For instance, Arvanitou *et al.* noted that a scientific software developer must choose between various good practices depending on developing a software tool or a data analysis pipeline [3]. The authors argue that the selection of practices should follow the software quality attributes to prioritize [15]. Importantly, it implies that priorities will vary between software products based on the trade-offs between these attributes (e.g., performance vs security). As bioinformaticians are rarely familiar with the meaning and importance of these attributes [17], we present an overview and short descriptions in **Supplementary Table 1**. Within bioinformatics, attributes such as functional suitability and performance are implicitly prioritized, while other attributes, such as maintainability, portability, and reliability, are often neglected. Through implicit prioritization, most software is developed as a prototype, even when the goal is to create a long-term product [18]. This report focuses on three target quality attributes as our learning goals: reliability, performance, and extensibility (see also **Figure 1**).

The hardship of systematic, automated testing of scientific software has been discussed in detail [1,16, 17]. Scientific software development requires dedicated testing methods to maintain scientific integrity since unexpected discoveries can confound expected outcomes and error detection. Scientists tend to assess the model's validity but do not thoroughly test the code that implements it, which betrays the common confusion of a model and its implementation as a single entity [17]. Uncovered faults can, and sometimes do, lead to incorrect scientific insights, as shown in multiple examples [19]. These observations highlight the importance of considering unit testing and verification for scientific software (**Figure 1**, **Supplementary Table 2**).

A fundamental aspect of bioinformatics software development involves the integration of functionalities from various software packages – a concept referring to collections of code that perform specific tasks. This approach of combining packages from several resources has several implications [1,3,7,8]. Among these, we focus on two critical issues. First, over time, the software may become increasingly challenging to maintain. Indeed, the complexity, size, age, and code's change-proneness heavily affect its maintainability [14]. One can address this increased complexity through a shared understanding and implementation of functions and modularization (**Figure 1**, **Supplementary Table 2**). Second, package management (including versioning) is crucial to ensure

maintenance and ease of development, reproducibility, and reusability. Frameworks [20,21] and package management solutions [22,23,24] are required to achieve these qualities (**Figure 1**, **Supplementary Table 2**).

We found that the literature mainly targeted towards bioinformaticians rarely covers the topic of coding as a team or using multiple people's expertise in software development. Usually, the advice encourages beginners to ask for help when facing problems with their code, not to work together from the start. Such advice includes consulting with colleagues, finding a mentor, or participating in online communities (e.g., Stack Overflow or Biostars) [25]. However, the described framework primarily focuses on individual practices, calls upon a specific (often scientific) issue, and insufficiently recognizes unknown unknowns. This scheme contradicts software engineering-oriented literature, where the main focus is on practices for coding in teams [26,27]. The only counterexample we found is the Code Clubs described by Hagan et al. [28]. In their research group, members collectively engage in software development through code reviews, coding in pairs, and software engineering education through workshops or seminars [28]. The authors of Code Clubs present tips on organizing meetings to receive the best support and learn new software-related concepts. Sharing coding experiences with others helps minimize isolation, allows individuals to learn from their peers, and maintains standards for writing better-quality software. Therefore, we also established a learning club called software quality seminars, regular code reviews, and a resource-sharing platform to foster team effort (**Figure 1**). In the following sections, we illustrate the merit of developing a learning community that enables the adoption of good software engineering practices by reflecting on our own code production.

Table 1: **Collection of recommendations for improving scientific software quality.** Some guidelines are vague, have a varied scope, and target different stakeholders. Therefore, finding individual responsibility and actionable points from the literature may be challenging.

Category	Recommendation	References
Software development 101	Sanity check on input parameters	[7]
	Do not hard-code changeable parameters and paths	[7]
	Do not require superuser privileges for installation and usage	[7]
Advanced software development	Usage of design patterns	[3]
	Adoption of international best practice standards of software quality	[29]
	Regular refactoring	[3]
Software development process	Continuous integration	[3]
	Agile software development methodology	[2,3]
	Educated choice of software development methodology	[14]
	Independent review of source code	[1,14,29]
	Code quality monitoring	[3]
	Inclusion of appropriate license	[7,29]
	Cooperation between developers and users	[29]
Testing and validation	Establish validation and acceptance procedures	[29]
	Provide a small test set	[7]
	Standardized tests	[2,3,7,14,29]
	Ensure reproducibility of results	[7]

Category	Recommendation	References
Reproducibility	Standardized working environment and automation	[2,29]
	Version control	[2,3,7,29]
	Rely on package managers	[7]
	Containerization for portability	[2,7]
	Tagging of software version for reproducibility	[7]
Documentation	User (and developer) documentation	[2,7,14,29]
	Requirements gathering	[2,14]
	Description of the software version used, its configurations, and parameters in publications	[29]
Community effort	Contribute to open-source development	[1]
	Reuse existing (reliable) software	[3,7]
	Preferentially selecting freely available open-source software	[29]
	Encourage user participation in the software development process	[29]
	Recognition and assignment of adequate time for quality-assured development	[14,29]
	Recognition of software development as academic achievement	[1,29]
	Support for developer community for long-term maintenance (when applicable)	[1,29]
	Financial support for software development and maintenance	[1,29]

Coding in teams

Beyond the brief mention of getting support in the guidelines for bioinformaticians, specialized literature examines how to organize team coding activities effectively. Indeed, programming as a collective practice is a critical notion in software engineering. Maximizing team cohesion while minimizing code coupling represents a central theme [30]. From the code's perspective, the adoption of sound software design enforcing modularity and extensibility ensures the viability of a software project [31]. Nevertheless, from a software development perspective, one should prefer team management practices centered around communication and collective governance [31].

We generally understand management as performing specific tasks: planning, monitoring resources, and tracking progression [32]. Typically, a single individual, referred to as the "manager" of the project, takes on these functions. In this context, manager is a role rather than a particular person's title. In the specific context of academic computational projects, one rarely finds a strict division of labor when considering software project management. Some tasks, such as risk assessment, budget, and time management, are discussed at the project's conception (e.g., during grant application) and thus decoupled from the actual software development phase. The remaining management tasks would often fall on the developer(s). Implicit decision-making is one of the critical challenges current bioinformatics projects face.

As agile is the primary recommendation for team management in these guidelines (**Table 1**), its relevance to academic settings is worth discussing. Agile methodologies, which emphasize iterative development, regular feedback, and flexibility, could significantly improve the way teams handle the

evolving demands and rapid changes typical in academic computational projects. Through more team communication, one outstanding aim of agile is the aspiration for more autonomy in organizing the work of software developers. Practices and methods aligned with agile prescriptions include planning a minimum viable product, documenting requirements, organizing stand-up meetings, defining and assigning tasks, pair programming, and code reviews. Many of these practices do not require the manager's presence but assume a collegial work culture and standardized procedures. The additional overhead in terms of time and resources needed when developing is offset by the benefits mentioned above regarding software resilience and improved team capabilities.

Incentivizing collective ownership and governance promotes the adoption of software engineering best practices among developers contributing to a software project [33]. 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 project's different parts [30]. Furthermore, by exposing every developer to various tasks throughout the project development, one strengthens the knowledge and skill base of the team as a whole, as well as creates an improved mutual awareness of team member expertise. This mutual awareness, the transactive memory system, is linked to increased team performance [34]. These merits further improve the team's capacity to overcome technical challenges that will arise throughout the development process.

We do not believe that all the software engineering guidelines employed in the industry are necessarily relevant to producing scientific software in academia. Indeed, the circumstances differ significantly, mainly due to the outcomes of research projects (papers, tools, protocols, etc.), and we need to credit the particular individual researchers for their career progression. However, this should not prevent interactions between developers of distinct projects, as these interactions will ensure compliance with the rules and increase quality. Our proposition of building a learning community centered on coding is our attempt to bridge the gap between the merits of coding in teams and the reality of individually credited projects in the academic setting.

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

Our experience in improving development processes as a team

Our team (composed of members from several academic research groups) adopted improved software development practices by implementing three pillars: 1 - software quality seminars, 2 - code review sessions, and 3 - resource sharing. We describe our experience and recommendations for implementing these pillars.

Software quality seminars

Within the context of our software quality seminars, we have implemented a system for transferring knowledge among participants. Such seminars substitute for a more formal computer science education, which most bioinformaticians lack [2]. Each seminar is structured to build a shared vocabulary among members to facilitate discussions on implementation details and code structures. These seminars include presentations and demonstrations covering basic concepts. Beyond this, we also use this platform to introduce new techniques and showcase tools not limited to specific projects. It helped in three ways: broadening our collective knowledge, serving as an opportunity to look at those more theoretical concepts in practice, and building a community by encouraging all members to

present their topics of interest. While preparing the lectures can represent a significant time investment, this effort is rewarded by the substantial knowledge gained from attending others' presentations. Finally, the investment pays off in the long run since acquiring knowledge leads to greater efficiency and expertise in future projects.

Code reviews

The benefits of code reviews have been reviewed [37]. Some benefits, such as implementing consistent coding standards and detecting bugs/errors, are obvious. In contrast, others represent less expected outcomes, such as diverse learning, fostering a positive environment, or enhancing efficiency. Before a scheduled code review, the developer writes their code in a way that others will understand, which enforces improved code readability and structuring. This expectation is largely self-inflicted as each person feels pressure to expose their weaknesses - even within a friendly environment. During a code review session, the developer must clearly explain some aspects of their code (e.g., structure, algorithm implementation, or performance-related decisions). We recommend that the developer decide the aspect of the code they want to focus on during these sessions. While code review sessions generally focus on implementation details, they can trigger discussion on any aspect of the code, including user interface design, documentation, and architectural considerations.

The other participants may not be deeply familiar with the project addressed explicitly by the code, but they bring their complementary knowledge and viewpoint. The feedback obtained can help fix existing or potential future issues, improve the implementation, and produce cleaner and more concise code. Our experience indicates a broader adoption of theoretical aspects and good software engineering practices. We highlighted these during code review sessions. We found that the implicit soft peer pressure derived from these code review sessions successfully addressed most goals: standardization of practices, improved code quality, and enhanced software usability.

As a beneficial side effect, we observed an enhanced understanding of the members' projects that naturally resulted from scrutinizing the code. It gave a deeper understanding of the underlying scientific questions and led to more insightful comments during subsequent group meetings. Additionally, a hands-on analysis of everyone's code revealed the repetitiveness of certain coding elements across the projects. To address this redundancy, we recommend implementing a system to share resources.

Resource sharing

Resource sharing fundamentally involves ensuring that valuable resources are readily accessible to all participants. We efficiently implemented resource sharing through two perspectives: external open-access resources (forums, repositories, packages, and libraries) and internal resources (including within-group tools). The internal aspect is crucial as it fosters team contributions that enhance individual project development. For instance, consider a shared repository containing various computational tools developed by group members. These tools are universal and aligned with the group's research questions. For example, a module that performs a gene set enrichment analysis and summarizes the results can be incorporated into different pipelines without rewriting the code that applies particular packages. A set of modules with standardized documentation and API becomes a toolkit available for reuse by all group members. Furthermore, the group collectively develops and reviews the underlying codebase, enhancing utility and quality.

We believe these three pillars are the minimum requirement for achieving lasting improvement in software development within research teams. Still, bioinformaticians of other groups should tailor the content and the frequency of these meetings to their specific needs. As a concrete example, many

regular attendees of our sessions have extensively implemented various software development methods discussed (such as object-oriented programming, user stories for documenting requirements and assumptions, using Jira for feature additions and bug reporting, and continuous integration with Git) while collaboratively working on the same codebase for the latest JASPAR database release [11]. Additionally, it is worth noting that this article was successfully written using Manubot, a tool based on continuous integration [38].

In the following sections, we discuss how software quality seminars and code reviews helped with the examples of three specific software engineering notions: modularization, testing, and dependency management.

Examples of improved development processes

From our experience, we observed that it might be challenging to adopt the listed practices. Therefore, instead of providing rules, we aim to encourage a mindset of mastering software engineering through an iterative process. We illustrate it by discussing our example concepts from different angles, levels, and formats in the software seminars and in practice at code reviews. We share specific examples from our projects to highlight how these concepts changed how we produce software (**Supplementary Figures**).

Modularization

Modular design is one of the most common approaches for team programming, ensuring the maintainability and extensibility of a software product. We understood from the guidelines and experiences of the team that moving from unstructured scripts to organized code with functions brings several benefits at a low cost. Consequently, this topic was covered several times during our software quality seminars and code review sessions.

Specifically, we dedicated software quality seminars to the following topics to improve modularization (**Supplementary Table 2**): object-oriented programming, class diagrams and unified modeling language in general, design patterns, software architecture, Snakemake [20], S4 objects, R package development, a case report from the organization of the JASPAR database project [11], and a review of the book titled The Pragmatic Programmer [39]. We understood that modularization can take form on many levels. On a small scale, it means naming and organizing parts of the code into functions. Once a code grows, one can start refactoring into classes and focus on the coherence and coupling of the parts (**Supplementary Figure 1-2**). When building a pipeline of scripts, one can identify coherent modules that would translate to rules in Snakemake [20] (**Supplementary Figure 3-4**.) Modularization means continuously monitoring the code, recognizing a code that grew too much, and re-structuring it into smaller parts. It involves an understanding that the code is not a static entity but an ever-growing, ever-changing organism.

A recurring question is whether a script needs refactoring or can remain a prototype. Taschuk and Wilson [7] suggest a cut-off at which one reuses a script, shares it with others, or uses it to produce findings in a publication. This definition would potentially include most code written by bioinformaticians. Still, we should weigh the time spent on improving the scripts against the time required to deal with suboptimal code on a case-by-case basis. With practice and exposure to a lot of code, modularization becomes the norm, which reduces the distance between a prototype and a refactored code.

Testing

As highlighted in the literature [17], testing is complex for scientific software. However, it is a central concept in team programming, as test coverage increases trust and allows the safe addition of new features by any member. We revisited testing multiple times in our session through discussion around debugging tools, how to write unit tests in Python (`pytest` and `unittest`) and R (`testthat`), the type of functions to test, and the automation of tests via continuous integration (e.g., GitHub Actions) (**Supplementary Table 2**). The main challenge usually lies in viewing software testing as more than just validating the scientific features of the software on a small test dataset. Similarly to modularization, a recurring question was when to start adding tests. We view testing not as a form of quality control that acts as a sort of checkpoint but as part of an iterative development process. Therefore, we advise testing early, departing from the mindset: "I will start tomorrow after implementing this new idea." Peer reviews challenge the implementation, while code reviews provide an optimal platform to discuss tests. One can view this aspect as a scientific endeavor when we envision the edge cases and discuss the properties of the biological question.

Dependency management

Given the large number of dependencies and the entire ecosystem of tools involved in scientific software development, managing these dependencies is crucial for ensuring the reproducibility of findings. In a team setting, it is natural to create identical environments for all developers to ensure they can run the code identically. In the software quality seminars related to dependency management, we covered the following topics: container solutions [23,24], R package development, and Anaconda [22] (**Supplementary Table 2**). We established a DockerHub account for our group [40] to share our custom containers (**Supplementary Figures**). This resource enables easy installation of our Snakemake pipelines across different servers.

In sum, we view the implementation of software quality seminars, code reviews, and shared resources in the academic setting as critical tools to improve coding and to better trust the resulting scientific discoveries. Nevertheless, we recognize that scientists can choose to implement all or any of them as independent activities. We observed that even a single activity benefits the members' coding experience and the resulting code quality. As good practices become routine, the required time investment will decrease, and the benefits will become more apparent. Finally, the shared knowledge base and standards allow us to make new group members adopt good coding practices more quickly.

Conclusions and Future Perspectives

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. Working in a team and following standards is not an option for large software projects that support many researchers and contribute to novel findings. We argue that following good software quality practices and mimicking team structure also benefit small projects. The overall performance increases when team members are familiar with each other and together build problem-solving routines through cumulative experience [30]. In a group, the knowledge of who knows what speeds up problem-solving [34]; time spent together, and social factors ease technical knowledge transfer [30]. Therefore, we motivate all group leaders with a computational component, be it small or not, to build an environment for their trainees to communicate and discuss software quality aspects.

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 and the maturation of working solutions. Unfortunately, we recognize that the lack of funding for software maintenance prevents achieving a level of software quality that would inspire confidence in the results [1]. Funding agencies, often through peer reviewers, emphasize “novelty” by enforcing new software development. This focus can make it difficult to justify dedicating time to the maintenance of software or computational resources that offer limited scientific output, especially when considering journal publications as the only token of success. As Alexander Szalay puts it, “the funding stops when researchers develop the software prototype” [18]. Researchers want to build on each other’s findings and use published novel software as tools, but they might need to spend a significant amount of time adopting or maintaining that software [1,3,7]. The scientific ecosystem would benefit from funding earmarked for maintenance and dedicating time to it in project proposals. On a positive note, a few agencies, such as the Chan Zuckerberg Initiative Essential Open Source Software for Science fund [41] and the Schmidt Futures through their Virtual Institute of Scientific Software [18], have recognized this missed opportunity. They are starting to address the lack of funding in this aspect. The scientific community and funding agencies should welcome the efforts of maintaining original software and encourage its updates instead of the development of replacement software with the risk of remaining unmaintained.

To summarize, it is of utmost importance for the scientific community to recognize the limitations of the software it produces and to acknowledge the flaws in the coding process. To harness the potential for substantial improvement, we recommend that scientific groups and institutions organize three pillar activities: software quality seminars, code review sessions, and the implementation of resource sharing. These initiatives will engage members of research groups in one another’s projects, facilitating the practical exchange of knowledge and feedback on code and beyond. Finally, we strongly advocate for sustainable funding to maintain existing scientific software and recognize software that adheres to best practices.

Acknowledgements

The authors acknowledge the contributions made by all the participants of our code reviews and software quality seminars at the Centre for Molecular Medicine Norway (NCMM), University of Oslo, and the helpful feedback on an early version of the manuscript provided by Ine Bonthuis, Nolan Newman, and Romana Pop. The authors were supported by funding from The Norwegian Research Council [187615], Helse Sør-Øst, and the University of Oslo through the Centre for Molecular Medicine Norway (NCMM) (to Mathelier and Kuijjer groups); the Norwegian Cancer Society [197884, 245890] (to Mathelier group), the Research Council of Norway [288404] (to Mathelier group), the Norwegian Research Council [313932] (to M.L.K.), the Norwegian Cancer Society [214871, 273592] (to M.L.K.), the European Union’s Horizon 2020 research and innovation program under the Marie Sklodowska-Curie grant agreement [801133] (to L.H.)

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. **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

9. **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

10. **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

11. **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

12. **The Network Zoo: a multilingual package for the inference and analysis of gene regulatory networks**
Marouen Ben Guebila, Tian Wang, Camila M Lopes-Ramos, Viola Fanfani, Des Weighill, Rebekka Burkholz, Daniel Schlauch, Joseph N Paulson, Michael Altenbuchinger, Katherine H Shutta, … John Quackenbush
Genome Biology (2023-03-09) https://doi.org/gtg476
DOI: 10.1186/s13059-023-02877-1 · PMID: 36894939 · PMCID: PMC9999668

13. **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

14. **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

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

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

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

18. **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 · PMID: 35831588

19. **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 · PMID: 17185570

20. **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

21. **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 · PMID: 28398311

22. **Unleash AI Innovation and Value**
Anaconda
https://www.anaconda.com/

23. **Home**
Docker Documentation
(2024-03-07) https://docs.docker.com/

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

25. **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

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

27. **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

28. **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

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

30. **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

31. **The psychology of computer programming**
Gerald M Weinberg

Dorset House Pub (1998)
ISBN: 9780932633422

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

33. **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

34. **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

35. **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

36. **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/

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

38. **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

39. **The pragmatic programmer, 20th anniversary edition: journey to mastery**
David Thomas, Andrew Hunt
Addison-Wesley (2019)
ISBN: 9780135957059

40. **Docker** https://hub.docker.com/u/cbgr

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