Bioengineered 6:4, 193–203; July/August 2015; Published with license by Taylor & Francis Group, LLC

Engineering bioinformatics: building reliability, performance and productivity into bioinformatics software

Brendan Lawlor[1,*] and Paul Walsh[2]

[1]Cork Institute of Technology; Cork, Ireland; [2]NSilico Life Science; Cork, Ireland

There is a lack of software engineering skills in bioinformatic contexts. We discuss the consequences of this lack, examine existing explanations and remedies to the problem, point out their shortcomings, and propose alternatives. Previous analyses of the problem have tended to treat the use of software in scientific contexts as categorically different from the general application of software engineering in commercial settings. In contrast, we describe bioinformatic software engineering as a specialization of general software engineering, and examine how it should be practiced. Specifically, we highlight the difference between programming and software engineering, list elements of the latter and present the results of a survey of bioinformatic practitioners which quantifies the extent to which those elements are employed in bioinformatics. We propose that the ideal way to bring engineering values into research projects is to bring engineers themselves. We identify the role of Bioinformatic Engineer and describe how such a role would work within bioinformatic research teams. We conclude by recommending an educational emphasis on cross-training software engineers into life sciences, and propose research on Domain Specific Languages to facilitate collaboration between engineers and bioinformaticians.

Keywords: bioinformatics, microbial biotechnology, process, survey, software, software engineering

*Correspondence to: Brendan Lawlor; Email: brendan.lawlor@gmail.com

Submitted: 02/04/2015

Revised: 05/05/2015

Accepted: 05/06/2015

http://dx.doi.org/10.1080/21655979.2015.1050162

Problem Description

This paper identifies a significant lack of software engineering practices in bioinformatics when compared to commercial software development, which prevents the bioinformatic community from benefiting from decades of engineering efficiencies, rigour and quality. The problem is present in computational science in general, but for the purposes of this discussion, we will concentrate on bioinformatics. Software engineering skills are lacking, as is evident in the way in which software is developed in bioinformatic contexts. Although biologists and especially bioinformaticians possess programming skills, and use those skills as part of their day to day work, they do so in a way that is unstructured and not in line with modern standards of software engineering.[1,2] The problem has serious consequences for the field of bioinformatics and demands that we find effective solutions.

We will examine these consequences under a number of headings, but in all cases they boil down to 2 overarching problems: the bioinformatic community arrives at findings more slowly than it otherwise might, and those findings, when arrived at, are less reliable than they might otherwise be. By focusing on solutions to the lack of software engineering skills in bioinformatics, we can address both of these effects. A first step is to better understand their nature by identifying more precisely how and where they arise.

Inability to reproduce findings

A lack of software engineering infrastructure and techniques means that many publications which process data informatically cannot make that software or data available in a reproducible way for peer review. As a consequence, a significant percentage of findings is likely to be reversed or withdrawn from publication. The use of infrastructure such as source

code control systems and command-line build tools would improve the situation, by giving researchers the ability to easily publish and share the software that was used as part of their work. But these tools are either unknown or simply considered unnecessary for small teams of bioinformatic researchers.

Unreliability of findings

All surveys on scientific software development that we have reviewed cite a lack of software testing as being a constant theme of scientific development. Segal points out that the "lack of any disciplined testing procedure" is a characteristic of any development practice where the end user is also the developer.[3] According to a review by Morris "unit tests[A] often do not exist."[4] Because of the fundamentally important role of such tests in separating problems in the code from problems in the hypotheses, findings based on insufficiently tested software must be considered in turn insufficiently tested themselves. Compare this to the use of defective or uncalibrated lab equipment in order to fully appreciate the nature of the problem.

Limitations in data sample size

Many scientists run their software on multi-core desktops but do so in a single-threaded way which creates performance bottlenecks.[5] This is most likely due to a lack of familiarity with the kind of parallel computing techniques available to software engineers. The constraints that this practice inevitably imposes on sample size or sophistication of data analysis are clear: In order to execute programs to completion on desktops, even if in a time frame of hours and days, researchers will naturally reduce the number of sample points used, or eliminate steps which might increase statistical power but which have exponential or factorial performance profiles.[5] A parameter study, for example, can benefit from pairwise comparisons of its features; but the number of such pairs is $\binom{n}{2}$, where n is the number of features, so even modest values of n require concurrent programming and resource management to run to completion on desktop computers. Where multi-threaded implementations *are* used in scientific programming, they typically involve using OpenMP[B] (for multi-core) or MPI[C] (for multi-server). These solutions use low-level primitives and as such are painstaking to develop and can result in error-prone code which is difficult to change, especially in large systems.[6] Software engineering research has more recently concentrated on using higher abstractions which result in more intuitive ways to achieve concurrency, for example through the use of the Actor architecture.[7] There are examples of the successful use of such engineering to the bioinformatics community.[8]

Slowing the discovery cycle

Bioinformatic research is an iterative process in which the computational element takes up a significant percentage. If the researcher has to wait days to see computational results which will decide the next direction that the research is to take, momentum is lost and the entire process of research itself is slowed down. Software engineers can bring skills like performance optimisation and concurrent programming to bear on this problem, significantly reducing waiting times.

Reinventing the wheel

According to Prabhu et al. "a considerable portion of [scientists'] time is spent in many tedious [software development] activities" such as converting data formats or retro-fitting inherited software to work for new conditions.[5] This is a direct consequence of insufficient software engineering infrastructure and practices around the research team. Researchers are obliged to repeatedly cobble together solutions for every new direction they take. Naturally the nature of these improvised solutions does not facilitate their reuse - they typically don't exhibit high levels of maintainability or build-reproducibility - and so the problem perpetuates itself.

In all of the above cases, we can discern a parallel to the argument made by Ioannidis with respect to inexpert use of statistics in studies.[9] The danger to progress in bioinformatics is that much research may later be found to be invalid due to inexpert or non-transparent development of software. As Verma et al. point out, "the end goal of creating accurate and reliable scientific software is no less critical [than with commercial software] since incorrect results would greatly compromise the validity of the discovery."[1] This is an unsettling prospect indeed.

As *in silico* experiments become an increasingly important form of research and development, problems of reproducibility and reliability will become more obvious and more urgent. Moreover, software engineering techniques will be key not just in addressing those problems, but in the initial conception and design of such experiments.

Solutions from the Literature

These are some of the things that can go wrong in bioinformatic research when we fail to address the problem of its software engineering deficit. But why does this deficit arise in the first place? And what can be done to improve matters?

A number of the authors we have reviewed offer explanations and remedies for the problems described above. Hannay et al. identify a general lack of formal education and training and a reliance instead on informal learning from peers.[10] Segal & Morris among others emphasize the differences between scientific and commercial software development.[11] Similarly, Verma et al. cite a lack of requirements engineering in bioinformatic

[A]Unit testing is a software development practice in which individual units of source code - for example a class in object oriented programming, a procedure in imperative programming or a function in functional programming - are tested in isolation to determine whether they behave correctly.

[B]OpenMP - Open Multi-Processing - is an API specification for shared memory parallel programming. See http://openmp.org/wp/.

[C]MPI - Message Passing Interface - is a standardized and portable message-passing system designed by a group of researchers from academia and industry to function on a wide variety of parallel computers. See https://www.open-mpi.org/.

projects, as well as other factors that create the "unique situation for the field of software engineering" represented by bioinformatics.[1] Umarji et al. focus exclusively on the gaps in the education of bioinformatic software developers in software engineering principles.[12]

It is important to correctly identify all of the significant causes of the problem. If we start with a false or incomplete diagnosis the treatment is unlikely to be effective. We will look in detail at the root causes proposed by previous studies, but from the previous paragraph we can see that there are some elements in common in the way previous authors have understood the problem, and so in the solutions that they have proposed. Here we will categorize them as *education*, *methodology* and *special pleading*, and they can be described as follows:

Education

Some authors have found that bioinformaticians lack the necessary training in Software Engineering skills. Umarji et al. have surveyed bioinformatics curricula in the United States and found that "out of a total of 79 program offerings, there were only 2 instances where a software engineering related course was a required part of the curriculum" and that "there was no mention of the role and importance of software engineering in the curricula."[12]

Methodology

The wrong processes - or no processes at all - are being applied to the practice of bioinformatic research. Verma et al. report that "little emphasis is paid on the organization and requirement gathering process in the early stages of the software."[1]

Special Pleading

According to some authors, the field of scientific software development is so far removed from the commercial settings in which modern Software Engineering has emerged, that the rules from the latter simply do not apply. Authors have suggested that the 2 contexts are "fundamentally different" for reasons of subject domain complexity, requirements volatility and budgetary constraints. These differences make it problematic to "impose software engineering techniques on scientists."[11] So much space has been given to the differences between scientific and commercial development, that it is useful to break it down further as follows.

- **Subject Domain Complexity.** Segal & Morris assert that in the case of scientific software development the subject matter is simply too complex for the "average developer." In a similar vein, Hannay suggests that "developers are much less likely to need to be domain experts" in "regular" software development compared to scientific.[11]
- **Requirements Volatility.** According to Segal & Morris, "full up-front requirement specifications are impossible" where scientists are concerned, and that requirements rather "emerge" on an ongoing basis. The suggestion is that this is a distinctive feature of scientific programming, which makes the application of software engineering techniques more difficult.
- **Budget and Resources.** Verma et al. and Umarji et al. cite tighter budget and timetable constraints as a differentiating factor of bioinformatic software development, and therefore as one possible cause of a lack of software engineering best practices in that field.

End User (Scientist) as Developer

A number of authors point out cultural differences between scientists and software engineers as an important issue. Segal & Morris suggest that due to the subject domain complexity already mentioned, developers are likely to be the end-user scientists. But as Verma et al. point out, biologist stakeholders - who are the primary stakeholders in these settings - "may be more inclined to sacrifice program structure to get something that works."

Naturally enough, the solutions proposed by these studies flow from the diagnoses of the problem. Those who conclude that the problem lies in education propose improvements to curricula. Those that implicate incorrect methodologies suggest alternatives that are more suitable to bioinformatics. Papers which emphasize the disconnect (real or perceived) between scientific and software engineering worlds don't offer suggestions about how to bring software engineering values into the scientific community, which again is natural, given their premise.

Software Engineering vs Computer Programming

Before we examine the existing explanations and remedies for the software engineering deficit in bioinformatics, we make a brief but important digression: We outline the differences between computer programming and software engineering in order to prepare for later arguments that lean on these differences.

The skills required to program are not the same as those required to engineer a software solution. Programming is a subset of the discipline of software engineering in much the same way that draftsmanship is a subset of the skills required for architecture. This uncontroversial fact is under-appreciated in scientific settings, for reasons about which we might only speculate. It takes a great deal longer to make a software engineer than it does simply to make a programmer. This should come as no surprise, given the fact that Software Engineering is a distinct academic course of studies and a distinct professional discipline. Practicing software engineers draw from a large body of academic knowledge and a long and vital component of workplace experience. There is a long-standing recognition, going back to thought-leaders like EW Dijkstra, that software engineering is as much a craft as a science.[13] As such, its skills are acquired as much through a kind of apprenticeship as through the academic studies that precede it. This has been sufficiently appreciated by educators that some have sought to incorporate elements of that apprenticeship model into academic coursework.[14] The elements of software engineering practice that are often absent from bioinformatic teams

correspond to those elements which are typically learned by the software engineering apprentice (source control, build systems, unit testing etc). This is hardly surprising: Scientists learn more software development skills informally from other scientists, and through self-study, than through formal education.[10] In one study 84% of scientists who were surveyed indicated that they had relied mostly on self-learning for their software skills.[1] In either case, neither mode of learning can be compared to the prolonged exposure to best practices that software engineers typically enjoy.

Elements of Software Engineering Practice

In this section, we give an overview of some of the primary tools, techniques and skills of software engineering, and present the results of a survey which seeks to quantify the prevalence of these software engineering elements in bioinformatic settings. Our choice of which tools and techniques to emphasize are based on experience as practitioners, and we find ourselves in full agreement with other authors such as Wilson et al. with respect to those choices.[15]

Figure 1 shows the essential elements of software engineering practice, and illustrates the dependencies between them. We categorize Software Engineering elements into the separate layers of *infrastructure*, *processes* and *practices*, each layer building on the one below.

The basis of good practice lies in the correct choice of the *Tools and Infrastructure* indicated in **Figure 1**. Of course a software engineer chooses the tools based on the practices that she wishes to encourage, but their presence in a development environment is like a genetic marker that accompanies good engineering standards. The layers representing automated *Processes* and experience-based *Practices* contain their own 'markers' which depend on those in the layers below: Even the most skilled and experienced engineer will be thwarted by an inadequate development environment. With this in mind, we designed a survey to measure the prevalence of these layered 'markers' in bioinformatic research teams.

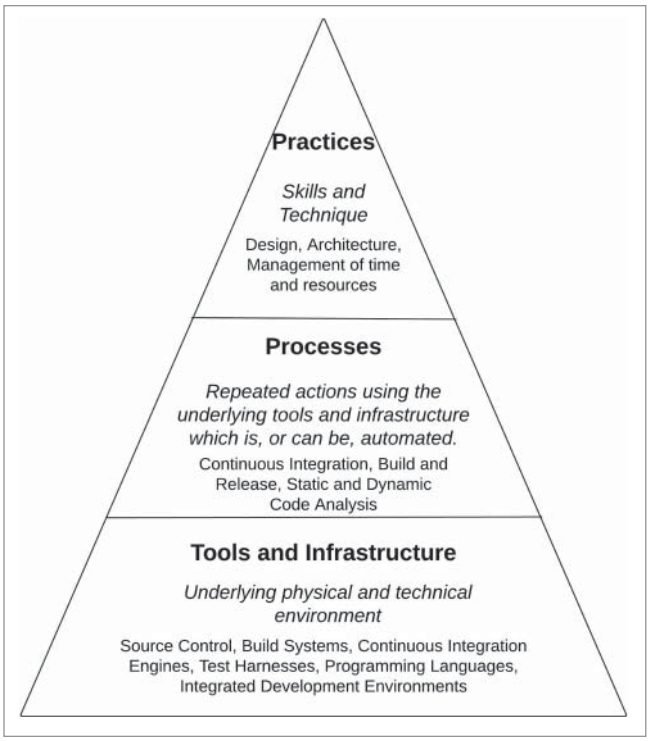

Figure 1. Key Components of Software Engineering.

A Survey of Bioinformatic Software Engineering Practice

We conducted 2 parallel surveys, one distributed to life scientists, and the other to developers of business software. In both cases we asked questions to identify attitudes toward certain key 'markers' of software engineering as described in the previous section. We reached 81 life scientists, 45 of whom developed their own software, and 36 business software developers. We used the Likert system of questionnaire design in which respondents rate their attitudes to statements from *strongly disagree* to *strongly agree* with a total of 5 degrees to choose from. We present the results below in a form that compares the differences between the 2 groups. The purpose of the business software data is to act as a control for attitudes toward the software engineering 'markers'. From the first set of results (**Fig. 2**) it's clear that business software developers and life scientists have distinctly different attitudes toward the standard elements of software engineering infrastructure.

Commercial developers almost unanimously strongly agree with the statements that build systems, source control, IDEs and Continuous Integration engines are used in their place of work. Life scientists show no such consensus. The closest they come to each other is in their attitude to the statement on source control where on average they agree with it, but where a significant minority have no opinion or disagree. Source control systems are of central importance in software engineering practice, on a par with disinfectant in an operating theater. Complete adherence to their use should be considered the norm, as is borne out by the business software respondents. The other 3 elements should be considered similarly vital to good software engineering practice.

When it comes to processes (automated or automatable) applied using the elements of infrastructure, the distinction between life scientists and the control group of business software developers is still clear even if less pronounced (**Fig. 3**). This difference is mostly a function of a reduced consensus among software engineers rather than a positive change in attitudes from

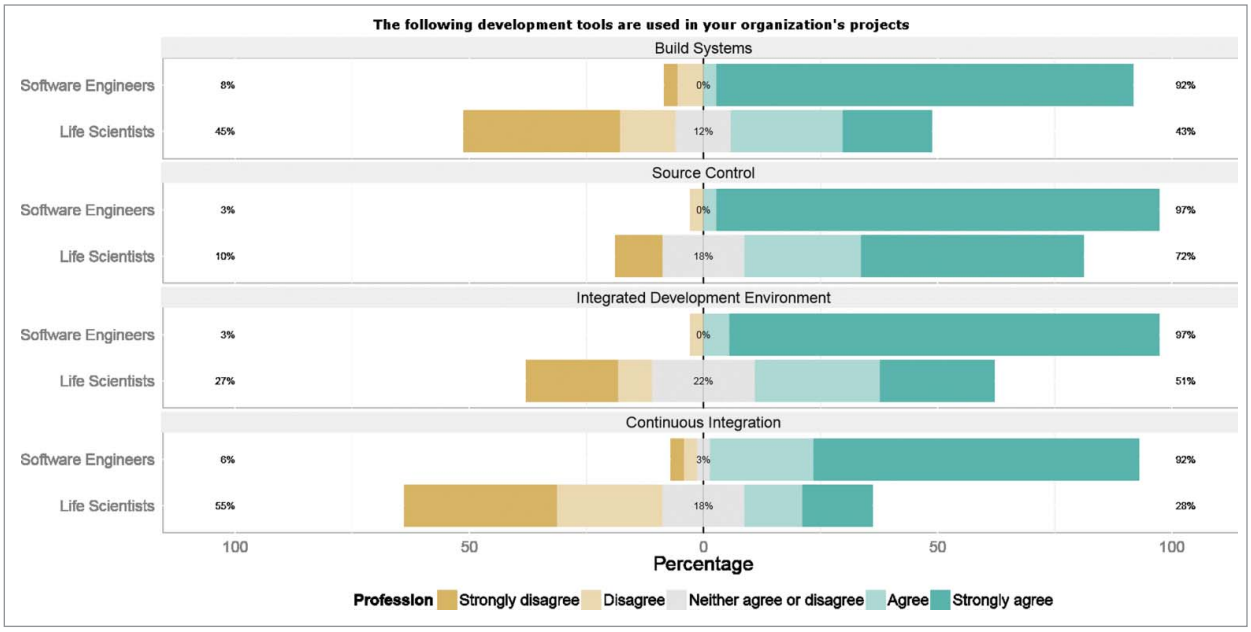

Figure 2. Responses to questions on infrastructure.

the life scientists. A particular point to notice is that although there is a relatively good showing for the use of source control in the previous set of results, life scientists generally neither agree nor disagree with the use of branching, despite the fact branching is one of the main advantages of using source control.

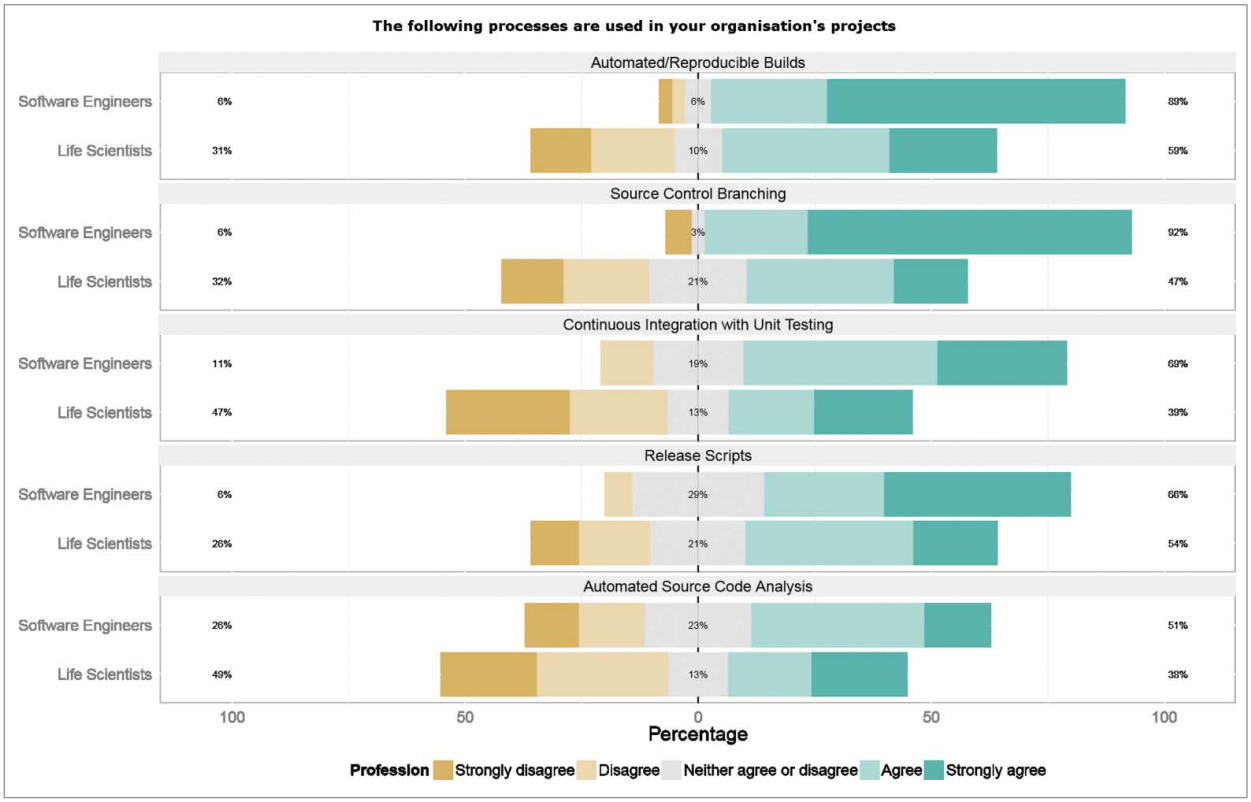

Figure 3. Responses to questions on processes.

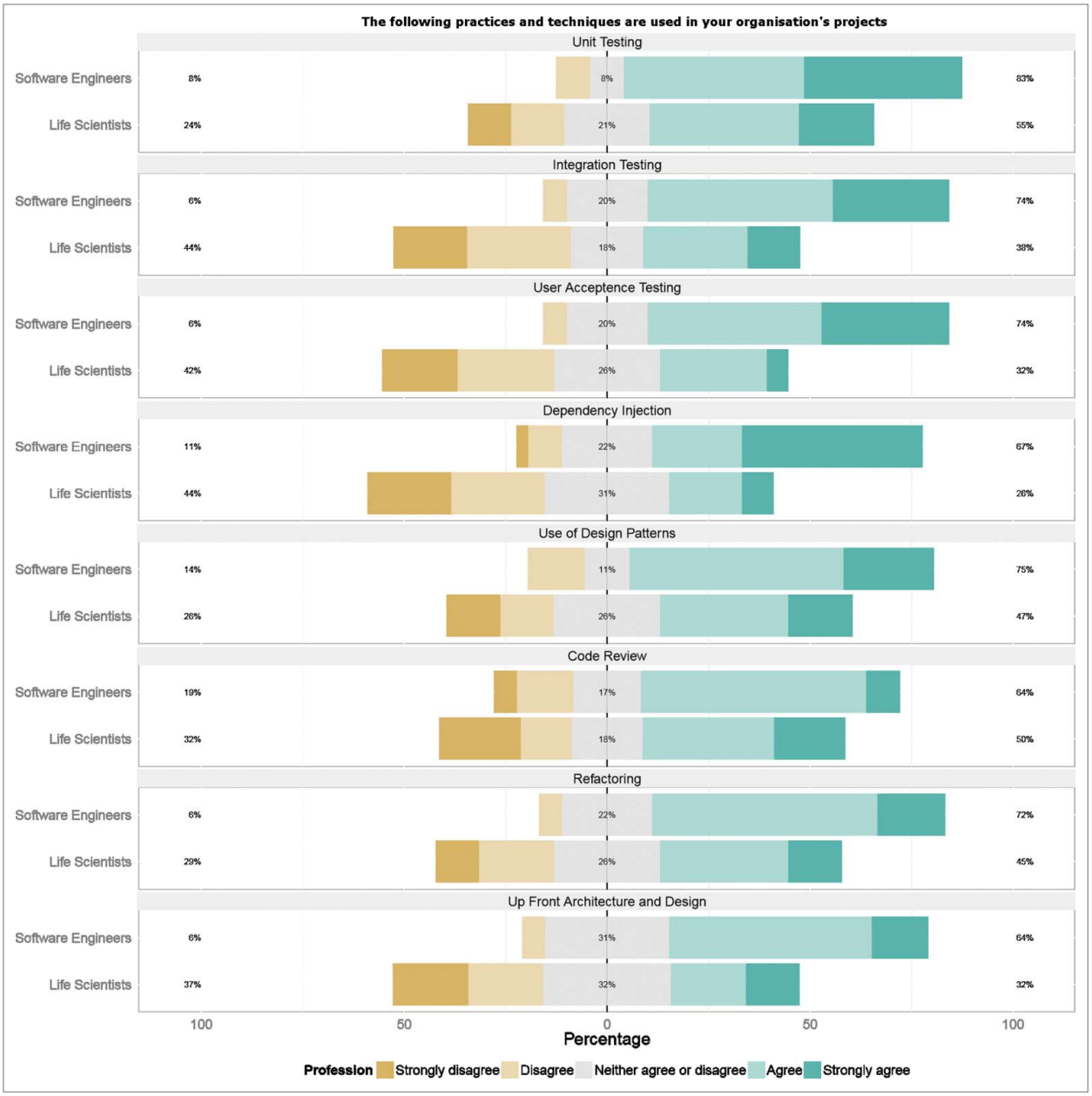

Figure 4. Responses to questions on practices.

As we look at the results for practices and skills (**Fig. 4**), a pattern begins to emerge. The further up the pyramid we go, the 'softer' the consensus among software engineers, while the attitudes of the life scientists remain more or less static. The overall picture of a clean, albeit smaller, separation remains.

The results dealing with goals and ambitions (**Fig. 5**) present a break with the previous pattern. Rather than the software engineers falling back to the neutral position of the life scientists, the latter group shows a stronger and clearer consensus in favor of the statements presented to them. In fact there is no discernible difference in attitudes between the 2 camps. It is interesting that in this section we have posed our questions in a slightly different way. Rather than asking about actual use, we have asked about importance. The goals and aspirations of the life scientists with regard to software architecture are no different to those of commercial software engineers. What they lack however, as indicated by the previous results, are the instruments and techniques necessary to achieve those goals.

Alternative Solutions

The results of our survey confirm the deficit in bioinformatic software engineering skills, while at the same time

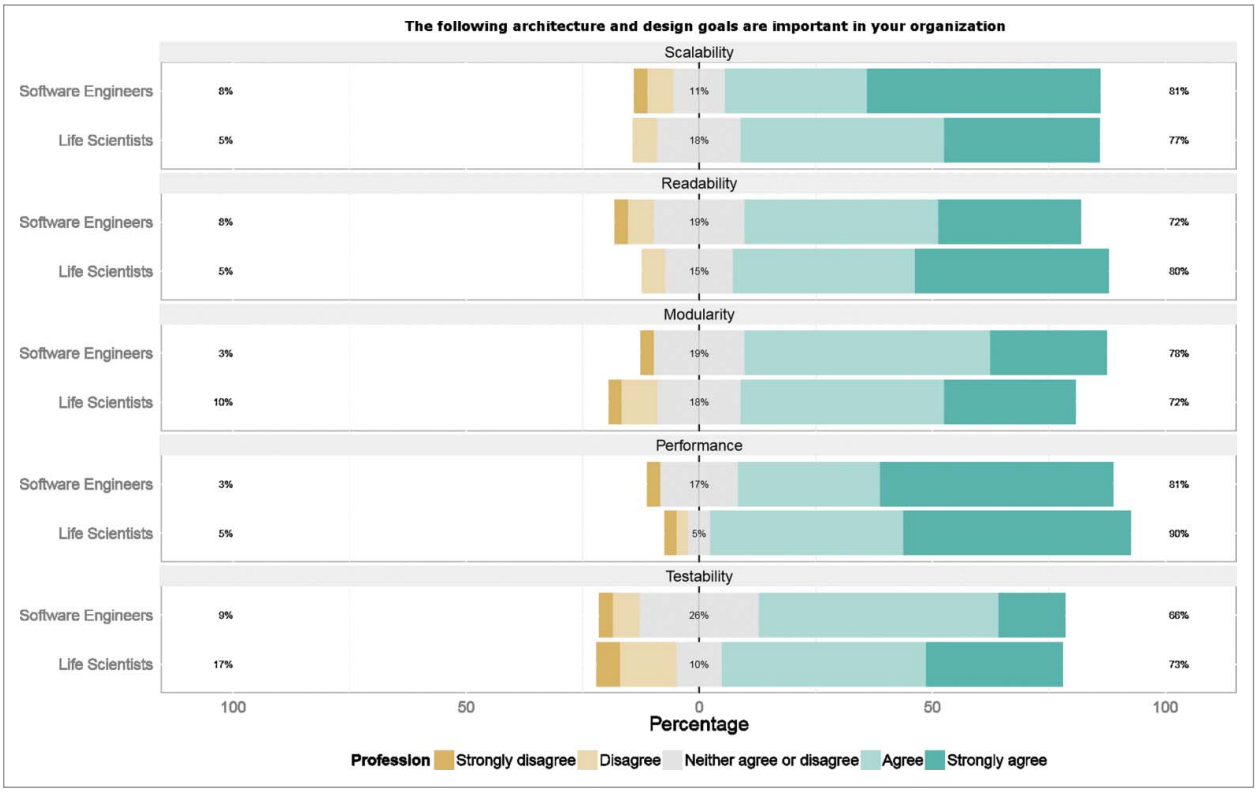

Figure 5. Responses to questions on goals.

indicating an ambition among bioinformaticians to bridge the gap. We now look at what the causes and remedies of this deficit might be and revisit the reviewed literature. We believe that in order to address this deficit effectively, we must take into account the difference between computer programming and software engineering as discussed above. We assert that it is impractical, if not impossible, to introduce the missing software engineering expertise into bioinformatics by treating that expertise as a sub-component of bioinformatics. Software Engineering encompasses too large a body of knowledge, which is acquired by too different a form of education to simply be bolted on to existing bioinformatic curricula. Put another way, we believe that the most effective way of introducing software engineering values into bioinformatic research is to introduce software engineers themselves, by recognizing the separate role of the Bioinformatic Engineer in bioinformatic research projects, and identifying the interface between the engineer and the scientist. Before we discuss how this might be done, we look again at the solutions from the existing literature, in the light of our assertions and findings above.

Education

While improvements in bioinformatic curricula, as suggested by Umarji et al. would be a positive step that could lead to improved communication between bioinformaticians and software engineers, such improvements would not be sufficient to bridge the current gap.[12] We believe that in addition, educators should target software *engineering* curricula and create specialized Masters and PhD programs in Bioinformatic Engineering, creating specialized software engineers who can dialog with biologists and bioinformaticians *as customers* based on a shared understanding of the research environment and the biology domain. Early introduction of software engineering graduates into bioinformatic research programmes would have a positive influence on the software developed as part of such research.

Methodology

Selecting appropriate methodologies is another necessary but insufficient step. Investigations into software engineering methodologies that suit bioinformatics projects are worthy, but who would steer the use of such techniques in the absence of a skilled and experienced software engineer? As Kane et al. have found, "[the] agile development approach . . . provides a model for collaboration between software engineers and researchers."[16] In other words, a good methodology works best in the context of existing software engineering skills, rather than as a replacement for them. Given such a context, it is worth pointing out the advantages of applying agile methodologies to scientific software development in general, and bioinformatics in particular. One benefit of agile processes is a rigour in defining requirements while at the same time embracing change in a way that permits discovery through prototyping. Agility can be seen as an example of modern software engineering serving the needs of bioinformatics.

Special Pleading

What about those arguments touched on above which suggest that scientific software development is too complex, too fluid in its requirements, and too badly funded to use software engineering techniques? Such arguments are based on special pleading and are problematic in a number of ways. Firstly, they don't point toward solutions. And secondly, such claims of being a special case can be arrived at too easily by specialist groups such as biologists, and fit too well with assumptions and professional biases - asserted and accepted without ever being truly examined. We examine those assumptions now in the order outlined above: *complexity of domain*, *volatility of requirements* and *limited resources*.

Complexity of Domain

There is something inherently contradictory in the claim that systems biology is too complex for software engineering or software engineers, thus making the biologist-as-developer a necessary feature of the bioinformatic landscape. If biological systems are complex then it follows that the software systems which model them will be complex. The complexity of the software however is twofold: Firstly there is the Problem Domain complexity inherited directly from the biology. Secondly there is the Solution Domain complexity that is inherent in any software abstraction. This latter, software-specific complexity is equal to the complexity of the modeled biology, but may add extra complexity of its own, depending on how sensibly the software is designed and realized. It takes a skilled software engineer, using modern software engineering techniques, to minimize this software complexity factor. It is clear that the "average developer" will not acquire biological expertise to same extent as the biologist, and will understand the biology only to that extent required to capture the necessary abstractions for the problem in hand, in collaboration with the biologist. It should be equally clear that complex systems modeled in software exclusively by biologists with limited software engineering experience will suffer from the limitations outlined at the beginning of this article. In the increasingly parallel, distributed and data-saturated context of modern bioinformatics, the exclusive role of scientist-as-developer advanced by Segal & Morris should be considered a bug rather than a feature.

Volatility of Requirements

The observation that scientific requirements are simply too fluid will bring a rum smile to the face of any experienced software engineer. The day-to-day reality of commercial projects is very different to the clean lines described in methodology literature. Perceived business needs always come first, often to the detriment of best practice. Part of the engineer's job is to incorporate unexpected and even capricious requirements into the project while minimizing the damage done.

In one sense, the life sciences enjoy an important advantage over business: The problem domain is much more stable over time and across projects. Certainly it grows to incorporate discoveries and occasional upheavals. But amino acids and cell division don't go in and out of fashion like financial instruments or business processes. Biologists uncover and even invent, but the underlying biology itself limits novelty. This allows engineers to build up *and usefully retain* expertise in the problem domain. (This cannot be said about commercial domains, where the only underlying biology that limits change is the neocortex of the customer.) One feature of modern software development which can take advantage of this relatively stable domain and facilitate communication between engineer and scientist is the Domain Specific Language (DSL).[17] As an alternative to a general purpose programming language, a DSL can provide a fluent interface between the problem domain of the biologist and the solution domain of the engineer. As such a DSL "offers substantial gains in productivity and even enables end-user programming."[18] As pointed out by Swertz et al, "[t]he working systems biologist wants to apply software tools to increase the understanding of biological function without having to 'tinker under the hood'."[19] DSLs bring some potential disadvantages as well, for example the risk of creating 'islands' of code so specialized as to become impenetrable to the non-expert user. Notwithstanding such risks, and indeed by way of addressing them, we consider the application of DSLs to bioinformatic software development as a worthy subject of further research.

Limited Resources

Budgets on commercial software projects are tight, as are the deadlines, and any experienced developer knows that there is a continuous cost/benefit calculation involved when making any significant technical decision. In this sense, commercial projects are no different to scientific research programs. What does differ is the budgeting process. Bioinformatic researchers should allocate adequate resources for software development at the outset.

Bioinformatics Engineering

We are arguing here for the recognition of the separate role of Bioinformatic Engineer in research teams, but this raises many questions of a practical nature and perhaps some philosophical ones too. How should bioinformatic engineers and bioinformaticians best communicate? Where would their competencies overlap? What should small teams with limited funding do? And in any case, does this separation of roles fly in the face of the cross-disciplinary nature of bioinformatics itself?

The intersection of the 2 sets in **Figure 6** shows the role that education can play in preparing bioinformaticians and bioinformatic engineers to work together. Engineers need to know enough about the biology domain to communicate effectively with bioinformaticians. Complexity of the problem domain does not prevent this from happening in similarly complex commercial settings, and despite much special pleading in the literature there is insufficient reason to think that bioinformatics would be different. Commercial software engineers typically specialize in 'verticals' and market themselves as much on the basis of their domain experience as

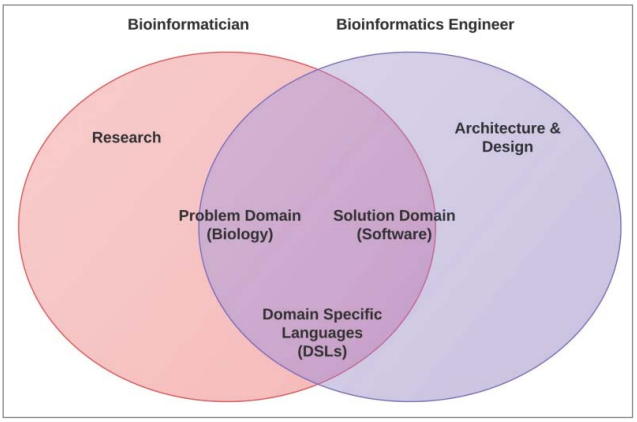

Figure 6. Suggested project Roles of bioinformaticians and bioinformatic engineers.

on their technical skills. Bioinformatics can be seen as a particularly stable and well defined problem domain, itself subdivided into various verticals. Bioinformaticians already understand programming enough to communicate their ideas and requirements through code (even if, as we have indicated earlier, there is enormous potential for DSLs to close the communication gap even further).

The non-intersecting parts of the 2 sets demonstrate the need for the bioinformatic engineer in the first place. The entire field of software engineering is too large to incorporate into the skillset of bioinformatics, and much of it is of no interest to the bioinformatician in the first place. Nobody expects him to build, or even understand the inner workings of the centrifuges and mass spectrometers that are so essential to research. Why then should we expect him to master the art of building large-scale, performant and production-ready software systems?

The point we are making in distinguishing the role of Bioinformatics Engineer can be summarized as follows: Software Engineering is vital to the discipline of bioinformatics *without being a core skill of that discipline*. This question of specialization is a logistic or even economic one which finds echoes in Ricardo's Law of Comparative Advantage: Even if it were possible for bioinformaticians to subsume the entire discipline of software engineering into their body of knowledge, it would not be desirable.[20] It would simply represent bad value. A bioinformatician investing the necessary time in engineering skills would pay a heavy price in terms of Opportunity Cost - the time *not* spent on study and research in core biological questions. Much better to lean on an engineering specialist in those key moments of research and development when engineering skills come to the fore.

What, then, are those key moments? **Figure** 7 categorizes the kinds of software development that would typically take place in a research team into 4 quadrants, based on 2 variables: Whether the work is core or peripheral to the team's output (focus), and whether the resulting software should be considered temporary or permanent (durability). We can use these variables to pinpoint the phases of research where bioinformaticians could increase their productivity by handing over to bioinformatic engineers, or at the very least, "change hat" and temporarily adopt an engineering approach.

To explain what we mean by these categories and variables, we refer to Morris' observation[4] that "[o]ne concern is that scientific prototype code, if successful, segues into applications that are distributed for wider research use. Later it may be adopted for production purposes, sometimes even for safety critical use." In other words, it is important to allow bioinformaticians to create code that is exploratory in nature but fragile from an engineering point of view. But it is equally important to ensure that such code does not form the basis of published findings or shared products and tools. The consequence of such fragility on published finding includes, but is not limited to, a difficulty in reproducing results, or a difficulty in analyzing the correctness of the code (due for example to poor readability, unreproducible builds, or even access to the correct version of the code). The consequence of fragile engineering on shared products and tools should be self-evident. A necessary balance between the need to explore and the need to consolidate must be struck, and we model this balance with the *durability* variable that distinguishes between temporary and permanent software.

Once we know which category a particular piece of software belongs to, we can create procedures for moving it to a different category should the need arise. For example, according to Sanders and Kelly some teams took a "do it twice" approach - that is, a rewrite of software according to more exacting engineering requirements.[21] This corresponds to moving software from the upper half to the lower half of **Figure** 7. So this is already practiced in some research teams. The point is to explicitly recognize these categories and put processes in place to avoid the kind of error that Morris describes.

The other variable, *focus*, distinguishes between software that is used as part of the scientific discovery process in a specific line of research, and code that could be considered 'utility code' to be reused in many different settings. The former should in principle be published along with the findings it helped to produce. The latter might find its way into a commercial or opensource product to be shared with the wider bioinformatic community. In both cases, the need for a transformation from temporary to permanent is the same, but the engineering skills and processes used to achieve it would differ - hence the distinction between core and peripheral software.

If a research team cannot fund a dedicated software engineer, it can still make use of the ideas presented here. The crossover points in competencies that we have identified above can serve as process boundaries, indicating where bioinformaticians should "change hat" and begin to approach their work with different goals in mind. But in order for this to happen, they must know that these boundaries exist; at a minimum they should be educated in an *appreciation* of software

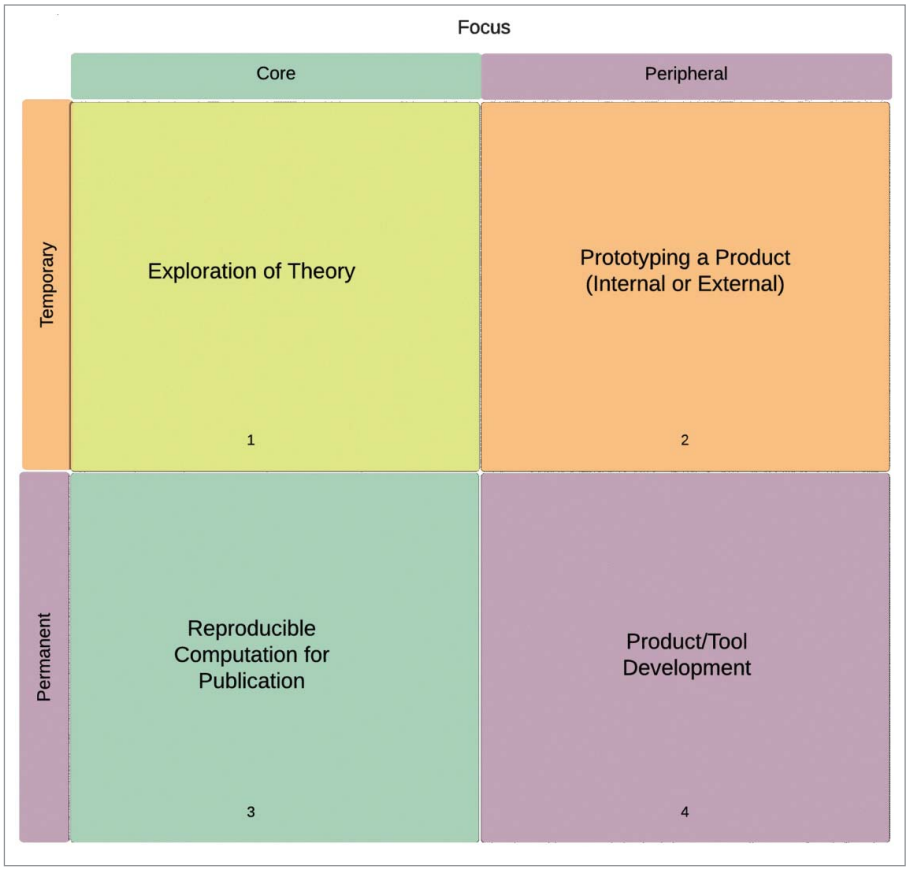

Figure 7. Handover points between bioinformaticans and bioinformatic engineers.

engineering even if their own engineering training will be of necessity - a peripheral part of their curriculum. As teams ramp-up in size and funding, they will permit themselves to take on specialists, and we contend that bioinformatic engineers should be one category of such specialists.

Projects that weave software engineering best practices into bioinformatics research and *in silico* experimentation reap concrete rewards. By employing software engineering techniques such as a layered architecture, explicit development models and a rigorous requirements-gathering approach, Walsh et al.[22] produced an accelerated research workflow tool, which is amenable to extension and is highly scalable. A blended team of biologists, computational scientists and engineers which has modeled integrated physiological processes of Caenorhabditis elegans (C. elegans) *in silico* has asserted that "[i]n order to be able to effectively manage the complexity that comes with integrating and maintaining coarse-grained architectures, tools, digital information artifacts and codebases, it is important for computational biology to fully embrace software engineering methodologies and best practices and follow the lead of the simulation based research in the physical sciences."[23,24]

Conclusion

Bioinformatics is still in the cradle, compared to many of its sibling sciences. In common with many other fields that combine computation, mathematics and statistics with the sciences, a lot of thought and energy is going into the creation of truly cross-disciplinary practitioners. The goal is to combine in one brain a rich knowledge of both biology and computation, because answering the questions that arise in one has become heavily dependent on mastering the skills developed in the other.

While there is no doubt about the soundness of this ambition, we feel that a distinction must be made between computational skills and software engineering skills. More to the point, we feel that these skill sets are so diverse and mastered by such different methods, that it is unrealistic to expect a single practitioner to combine biology, computational methods and software engineering. Moreover, it is unnecessary and uneconomical to try.

The alternative is already available to us. Software engineering is a discipline in which we apply computational skills to problems of other disciplines in such a way as to result in robust, reliable and maintainable solutions. While some fields of application are more exacting than others there is *no qualitative difference* between commercial software engineering and scientific software engineering. The extra degree of scientific complexity has parallels in commercial software development. The existing tools, techniques and practices of software engineers can bend to the particular needs of research. The only question that remains is how to reliably place those skills of modern software engineering at the disposal of bioinformatic researchers.

We argue for the explicit recognition of the role of the bioinformatic engineer, a software engineer who has been educated in the standard way for that discipline, and has specialized in the 'vertical' of systems biology (or a sub-field such as genomics, or metabolomics). Such an individual would embody all the skills that one would expect from an expert software engineer but would also have a deep understanding of the kinds of problems that biologists need to solve, and an appreciation for the manner in which they go about their research. In other words, we believe that the most effective way of introducing software engineering values into bioinformatic research is to introduce software engineers themselves. As a reasonable compromise, where this ideal is not immediately achievable, bioinformaticians could perform the *role* of bioinformatics engineer during the delineated phases of project work that we have identified.

One difficulty to be addressed as part of the proposed approach is hinted at by Prabhu et al. when they quote one scientist as saying that even "funding agencies think software development is free," and regard development of robust scientific code as "second class" compared to other scientific achievements.[5] The way in which research projects are funded does not currently take into account the costs associated with developing software. While not every project will be able to budget for a full-time bioinformatic engineer, research groups should be able to share such resources, or make use of specialized external software companies which would grow in number to meet demand.

The bioinformatic engineer does not in any sense remove the need for the cross-disciplinary figure of the bioinformatician. On the contrary - it is essential to an effective collaboration between bioinformatician and engineer that one have the skills and vocabulary to communicate needs to the other. The bioinformatician will very often communicate with the engineer using source code. As suggested by Wilson et al. it would be best if the bioinformatician also had a working knowledge of the basic tools of software engineering such as source control and unit tests. But the responsibility of identifying problems in design and code, fixing them, and shaping exploratory code into well-engineered solutions would lie with the bioinformatic engineer. We predict that this would substitute hours of drudgery for the scientist with hours of true productivity, and at the same time ensure performant, testable, maintainable and shareable code and reproducible results for the bioinformatic field at large.

Recommendations

- Explicit recognition of the role of Bioinformatic Engineer, along with a shared understanding of the competencies, functions and interfaces of that role.
- The creation of specialist post-graduate curricula to allow software engineering graduates to specialize in bioinformatic engineering. This should be seen as a parallel and complementary effort to the enlistment of computer science and biology graduates into bioinformatics post-graduate courses.
- Research into bioinformatic Domain Specific Languages to facilitate collaboration between bioinformaticians and bioinformatic engineers.
- Adequate funding for software engineering as part of bioinformatic research projects.
- Measures to encourage the creation of bioinformatic engineering companies to service the needs of smaller research teams which cannot afford dedicated internal bioinformatic engineering staff. Such companies could recruit and cross-train experienced commercial software engineers as well as taking up masters and PhD graduates from the specialist bioinformatic engineering curricula we have suggested above.

Disclosure of Potential Conflicts of Interest

No potential conflicts of interest were disclosed.

Funding

Paul Walsh is CTO of NSilico Life Science Ltd and is a Senior Research Fellow on FP7Marie Curie Actions IAPP Project ClouDx-i under grant agreement no. 324365.

References

1. Verma D, Gesell J, Siy H, Zand M. Lack of software engineering practices in the development of bioinformatics software. ICCGI 2013; 2013:57-62
2. Baxter SM, Day SW, Fetrow JS, Reisinger SJ. Scientific software development is not an oxymoron. PLoScomputational Biol 2006; 2:e87
3. Segal J. Some problems of professional end user developers. Visual languages and human-centric computing, 2007 vL/hCC 2007 iEEE symposium on 2007; 111-8; http://dx.doi.org/10.1109/VLHCC.2007.17
4. Morris C. Some lessons learned reviewing scientific code. Proc 30th Intl Conference Software Eng (iCSE08) 2008
5. Prabhu P, Jablin TB, Raman A, Zhang Y, Huang J, Kim H, Johnson NP, Liu F, Ghosh S, Beard S, et al. A survey of the practice of computational science. State Practice Rep 2011; 19
6. Schindewolf M, Biliari B, Gyllenhaal J, Schulz M, Wang A, Karl W. What scientific applications canbenefit from hardware transactional memory? In: High performance computing, networking, storage and analysis (sC), 2012 international conference for. IEEE; 2012; 1-11
7. Agha GA, Mason IA, Smith SF, Talcott CL. A foundation for actor computation. J Functional Programming 1997; 7:1-72; http://dx.doi.org/10.1017/S095679689700261X
8. Wiewiórka MS, Messina A, Pacholewska A, Maffioletti S, Gawrysiak P, Okoniewski MJ. SparkSeq: fast,scalable, cloud-ready tool for the interactive genomic data analysis with nucleotide precision. Bioinformatics 2014; btu343; PMID:24845651
9. Ioannidis JP. Why most published research findings are false. PLoS Med 2005; 2:e124; PMID:16060722; http://dx.doi.org/10.1371/journal.pmed.0020124
10. Hannay JE, MacLeod C, Singer J, Langtangen HP, Pfahl D, Wilson G. How do scientists develop anduse scientific software? Proceedings of the 2009 iCSE workshop on software engineering for computational science and engineering 2009; 1-8; http://dx.doi.org/10.1109/SECSE.2009.5069155
11. Segal J, Morris C. Developing scientific software. Software IEEE 2008; 25:18-20; http://dx.doi.org/10.1109/MS.2008.85
12. Umarji M, Seaman C, Koru AG, Liu H. Software engineering education for bioinformatics. Softwareengineering education and training, 2009 cSEET'09 22nd conference on 2009; 216-23; http://dx.doi.org/10.1109/CSEET.2009.44
13. Dijkstra EW. Selected Writings on Computing: a Personal Perspective. Springer-Verlag New York, Inc. 1982
14. Surendran K, Hays H, Macfarlane A. Simulating a software engineering apprenticeship. Software, IEEE 2002; 19:49-56; http://dx.doi.org/10.1109/MS.2002.1032854
15. Wilson G, Aruliah D, Brown CT, Hong NPC, Davis M, Guy RT, Haddock SH, Huff KD, Mitchell MI, Plumbley MD, et al. Best practices for scientific computing. PLoS Biol 2014; 12:e1001745.; PMID:24415924; http://dx.doi.org/10.1371/journal.pbio.1001745
16. Kane DW, Hohman MM, Cerami EG, McCormick MW, Kuhlmman KF, Byrd JA. Agile methods inbiomedical software development: a multi-site experience report. Bmc Bioinformatics 2006; 7:273.; PMID:16734914; http://dx.doi.org/10.1186/1471-2105-7-273
17. Van Deursen A, Klint P, Visser J. Domain-specific languages: An annotated bibliography. Sigplan Notices 2000; 35:26-36; http://dx.doi.org/10.1145/352029.352035
18. Kosar T, Barrientos PA, Mernik M, A preliminary study on various implementation approachesof domain-specific language. Information Software Technol 2008; 50:390-405; http://dx.doi.org/10.1016/j.infsof.2007.04.002
19. Swertz MA, Jansen RC. Beyond standardization: dynamic software infrastructures for systems biology. Nat Rev Genet 2007; 8:235-43; PMID:17297480; http://dx.doi.org/10.1038/nrg2048
20. Ruffin R. David ricardo's discovery of comparative advantage. History Political Economy 2002; 34:727-48; http://dx.doi.org/10.1215/00182702-34-4-727
21. Sanders R, Kelly D. Dealing with risk in scientific software development. IEEE software 2008; 25:21-8; http://dx.doi.org/10.1109/MS.2008.84
22. Walsh P, Carroll J, Sleator RD. Accelerating in silico research with workflows: a lesson in simplicity. Computers Biol Med 2013; 43:2028-35; http://dx.doi.org/10.1016/j.compbiomed.2013.09.011
23. Szigeti B, Gleeson P, Vella M, Khayrulin S, Palyanov A, Hokanson J, Currie M, Cantarelli M, Idili G, Larson S. OpenWorm: an open-science approach to modeling caenorhabditis elegans. Frontiers Computational Neuroscience 2014; 8; http://dx.doi.org/10.3389/fncom.2014.00137
24. Idili G, Cantarelli M, Buibas M, Busbice T, Coggan J, Grove C, Khayrulin S, Palyanov A, Larson SD. Managing complexity in multi-algorithm, multi-scale biological simulations: an integrated software engineering and neuroinformatics approach. Frontiers Neuroinformat Conference Abstract: 4th iNCF Congress Neuroinformat 2011; 112