Scientific Software Developer Productivity Challenges from the Molecular Sciences

Theresa L. Windus,[1,2,3] Jessica A. Nash,[2] and Ryan M. Richard[3]
1. Iowa State University, Department of Chemistry, 1605 Gilman Hall, Ames, IA 50011, USA
2. Molecular Sciences Software Institute (MolSSI), 1880 Pratt Drive, Suite 1100, Blacksburg, VA 24060, USA
3. Ames Laboratory, 2416 Pammel Drive, Spedding Hall, Ames, IA 50011, USA

Enabling developer productivity in a broad community, such as the molecular sciences (MS), is a complex, multi-dimensional challenge. If it was easy to solve, we would have done it by now! In this whitepaper, we will address just a few of the issues that we see in our MS community and a few of the solutions that we are trying to implement.

First, understanding what one means by developer productivity can be useful. There is usually some thought that the developer can quickly develop software that accomplishes some goal. But does this thought include quality, reusability, and sustainability? Maybe the developer doesn't need or want to have their work be reusable! This is especially true when the developer is just learning to code or doesn't want the responsibility and cost of maintaining their software for someone else. They may just want a one-time capability. They certainly want to get the right answer for the problem they are considering, but may not worry too much about the edge cases. Unfortunately, often these types of codes do get used by the developer or by someone else further down the line and then the software needs to be either fixed to be more friendly (i.e., providing documentation, testing, etc.) or developed again. These codes might be prototypes that eventually become "the" code or they get passed along from one student to the next to help save some time. A significant part of the legacy software that is developed in the MS community has had this type of background. Of course, there has also been significant effort in the MS to develop more community based codes. Even with this software, though, there is often not enough time/money to develop software that uses strong software engineering principles to ensure continued sustainability of the software. For the sake of this whitepaper, let's assume that developer productivity includes quality, reusability, and sustainability as parameters that must be included to have effective developer productivity.

Once we have an idea of what developer productivity means, we then have to determine how we are going to measure that productivity. This is another issue that can be very difficult to answer. For example, one very simplistic metric might be the number of lines of code per hour that a developer can produce. This metric, however, by itself does not include ideas of reuse (perhaps the best code is a few lines from the new developer, but uses functionality from other libraries that required time to understand) or quality (in the extreme case, anyone can pad a simple code with a lot of lines that don't do anything useful and may actually produce nonsensical results). Another metric might be the number of tests that are associated with the new software. This metric at least includes the idea of quality, but certainly doesn't ensure the quality of the code. Often we use fuzzy measures that incorporate the idea of time, correctness of results, good software engineering practices, usability/reusability, and lack of bugs. While

this whitepaper does not seek to solve this problem, it is certainly an issue that requires discussion.

In the MS community, there has been a growing revolution in the way that software is being developed. However, there are still multiple challenges to be addressed. In the realm of developer productivity, there is still a large tendency to design and implement new code, even when existing codes are available. A significant part of this is that the rewards (at least in academia and national labs) for developing new software are much higher than contributing or using existing software. There is still a perception that "owning" a code means that the owner is the main and perhaps only developer - being a contributor can require a large effort on the contributor's part (as well as recognition from the owner) to acknowledge the value of the contributing developer/software. Papers are still recognized as one of the leading metrics for productivity and if you are not the first or last author, your contribution can be lost (the author order is different in other fields, but this is the primary order in the MS field). While software as a deliverable is starting to be recognized, there is still the challenge of delineating one's contributions. Statistics tools that are available online, like GitHub statistics, can help with this - although, as noted above, lines of code don't necessarily translate into quality of code. Another significant reason for not using someone else's code is that it can take a significant amount of effort to understand the software before making a contribution. There is a perception that it will take longer to learn the existing software and make any necessary changes than it will take to create the software from scratch (this perception holds even for complex software where it is probably not true). Certainly poor documentation, lack of developer information for the community, and lack of forethought on the reuse of software can contribute to this perception. Combined with the existing reward system, these sociological issues are difficult to overcome.

The Molecular Sciences Software Institute (MolSSI) is trying to change these attitudes by acknowledging that software development and reusability is a key to enabling new theoretical and computational methodologies. The attitude of thoughtful reuse of software can change the landscape for developers and help them to be more productive. For example, the development of excellent basic linear algebra libraries (BLAS) and other mathematical solver libraries in the mathematics and computer science communities have enabled most MS software developers to rely on these libraries instead of creating their own version (although, even here there are reasonable exceptions). Having building block components within the MS community has been a goal for many years and has yet to be accomplished. In addition to the reasons given above, there are also the challenges of having a dependence in your software stack on software that might not be well supported. So, even if we choose to use someone else's component or library, we will have a native implementation in the code in case that component doesn't work as advertised or if the maintainers decide not to support the component anymore.

MolSSI is also providing educational resources and is reaching out to a broad range of people in the MS community to help them provide better software. The hope is that better software will be more likely to be reused in the community - thus enabling productive developers to use other people's libraries instead of developing their own (http://education.molssi.org/resources.html).

MolSSI resources and workshops provide information about careful design to enable cross use of modules, information about existing and evolving standards and their development, ideas of interoperability, use of version control, continuous integration testing, unit testing (and other testing models), software review, software dissemination, community building, and other best practices. Many of these workshops are attended by the younger generation of developers (undergrads, graduate students, and postdocs) who are more open to the ideas of software reuse.

Another challenge of developer productivity is the compiling of software and the dependencies associated with different libraries. Many developer hours are spent in setting up the appropriate build system and making it run with many different hardware and software architectures. In addition, this is one of those “thankless” tasks where the main reward is associated with the ease of others compiling the code. One does not often get thanked for making the code compile, but one certainly hears about it if the code does not compile or does not compile easily! C++ lacks a standardized build system and package manager, which in turn makes it difficult to create reusable C++ libraries and packages. CMake has become the de facto standard for writing build systems for C++ programs; however, CMake's generality and steep learning curve can make it difficult to write robust and reliable build systems. Often a developer will take a CMake system from other projects (whether they are fully appropriate for their software or not) and spend many hours adjusting and tweaking the CMake files to make the compile work on their specific system. Then they do the same thing when an issue comes up when trying to compile on a different computational platform.

As part of the NWChemEx project in the Department of Energy Exascale Computing Project, we have created a suite of CMake modules called CMakePP (https://github.com/CMakePP/), which focuses on automation and boilerplate reduction. Using CMakePP: dependencies can be downloaded, built, and installed with as little as their URLs; libraries and executables can be added simply by providing the path to the directory containing the source files; and packaging files are automatically generated for installed targets. CMakePP wraps this functionality in user-friendly functions that require minimal input. CMakePP and other compiling/portability tools such as Spack and Docker help with the compiling and deployment of software.