J Comput Aided Mol Des (2017) 31:301–304
DOI 10.1007/s10822-016-9997-x

The need for scientific software engineering in the pharmaceutical industry

Brock Luty[1] · **Peter W. Rose**[2]

Received: 23 July 2016 / Accepted: 8 December 2016 / Published online: 19 December 2016

Abstract Scientific software engineering is a distinct discipline from both computational chemistry project support and research informatics. A scientific software engineer not only has a deep understanding of the science of drug discovery but also the desire, skills and time to apply good software engineering practices. A good team of scientific software engineers can create a software foundation that is maintainable, validated and robust. If done correctly, this foundation enable the organization to investigate new and novel computational ideas with a very high level of efficiency.

Keywords Scientific software engineer · Research software engineer

Introduction

Computer-aided drug design (CADD) is a broad field including the traditional computational chemists who directly support discovery projects. These computational chemists are essential to drug discovery and make up the vast majority of what is recognized as CADD. However, there is a sub-type of CADD scientists who also have informatics and software engineering skills. This class of scientist could rightfully be differentiated as scientific software engineers (SSEs).

✉ Brock Luty
bluty@dartneuroscience.com

[1] Department of Scientific Computing, Dart NeuroScience, LLC, San Diego, CA 92131, USA

[2] RCSB Protein Data Bank, San Diego Supercomputer Center, UC San Diego, La Jolla, CA 92093, USA

We first started applying rigorous scientific software engineering practices in the early 90s. Agouron pharmaceuticals had just lost a number of CADD scientists that had been writing code. Their software was written using different styles and in most cases, they were the only ones able to repair and extend their code bases. Often the code consisted of commercial and academic codes glued together by scripts to solve a specific problem without reuse in mind. Thus a lot of effort was completely lost.

Following this catastrophe, a significant decision was made to follow good software engineering practices by a sub-group of CADD. This team made a concerted effort to write a common software library as the foundation for all in-house software applications. Everyone on the team was on the same roadmap, with coding standards, code reviews, and employing true software engineering practices (see below). We then developed applications using these libraries in a way that the applications could easily be combined. This allowed us not only to quickly try out new ideas that appeared in the literature, but also explore our own proprietary ideas in a robust and straight-forward fashion.

In the longer term, informatics skills and other enterprise software engineering skills were also introduced and the "ADDS group" was deeply involved in producing some of the most innovative and widely used tools at Pfizer. The Pfizer crystal structure database [1] allowed crystallographers to submit asymmetric units which were automatically prepared for SBDD (oriented in a project reference frame, optimized hydrogen networks, etc). The CCT services framework allowed computational chemists and other scientists to publish protocols (shell scripts, Pipeline Pilot Protocols, etc) and have the methods immediately exposed in the desktop applications. MoViT [2] is very powerful and user-friendly interactive 3D ligand and structure-based drug design application, which was directly

linked to the Pfizer crystal structure database and the CCT Services. These tools along with many other algorithms (e.g., AGDock [3] for docking, HTS [4] for scoring) are heavily used directly or indirectly via CCT Services by almost all discovery scientists. Having a small group produce this type of enterprise software was in a large part due to a dedication to following good software engineering and database design practices to create robust and maintainable software systems.

What is a scientific software engineer?

A scientific software engineer (SSE) is typically a PhD Scientist with a background in computational chemistry, computational biology, physics, or related fields. They, like many of their counter-parts in CADD, have a deep understanding of the physics of receptor-ligand interactions and the physical and biological behaviors of molecules. Occasionally SSEs are experts in other fields like statistical mechanics, machine learning, and enterprise informatics.

But the difference from the SSE and a traditional CADD scientist is in the way they develop software. A good SSE takes as much pride in their software engineering skills as they do in their scientific abilities. They make sure that the code that they write is exactly the kind of code that they would like to inherit. They have the ability to write code in a way that models the true science which makes it easier for another SSE to understand and maintain. They trade short term gains for long term benefits to the entire organization.

SSEs work as a team. All source code is shared among the team, with no duplication of effort and no direct "ownership" of any particular part of the source code. Ideally every piece of source code that is committed is reviewed by at least one other member of the team so the entire code base is understood by more than one engineer.

Scientific software engineers use best-practice software engineering techniques. This includes knowing how to gather requirements using their scientific abilities. SSEs also know how to design software that fits with the current architecture and how to leverage existing well-tested libraries to minimize the code they have to write. Designs are reviewed by the team to reach a consensus and/or approval from the principal architect all before any coding is done. SSEs use proper tools to support the construction of software including (1) issue tracking with prioritization, assignments and milestones (2) source code control which is ideally integrated into their development environment with each source code commit connected to the issue it resolves, (3) thorough unit and integration testing, ideally integrated with an automated build system that runs all tests upon code commits and (4) a staging process that involves migrating software from development to testing to staging and finally to production without any kind of recompilation or other changes. Databases for transactional data are designed in third normal form with proper constraints, to ensure that referential integrity is enforced. If performance is not acceptable, then an ETL system (see below) can be used to create a mart that has been de-normalized for performance. Logic is kept all in the source layer and no logic is put in the database (e.g. pl/sql). Putting logic in the database layer not only makes it hard to follow the logic in the middle-tier, it also greatly complicates the debugging process. Also if logic is in the database it greatly complicates the deployment process.

Scientific software engineers follow an agreed upon software development process like the Agile process. The Agile process loosely consists of (1) a daily "stand-up" and (2) "sprint transitions". At the daily stand-up, the entire team, led by a "scrum master", reviews their progress to date and describes any "blockers" that are keeping them from making progress. At the sprint transition, progress is presented to a larger group including demos to the scientist who will ultimately use the software. At these meetings, which are typically every other week or once a month, the customers of the software give feedback on the progress and help prioritize the features and bug fixes for the next "sprint". For additional discussion of different Software Processes please see the "Appendix".

Scientific software engineers use agreed upon rules for writing software. Common object-oriented design patterns are used where possible. Usually some kind of coding style is introduced based on established standards. Templates and guidelines for starting a new project in a given technology are documented and followed. Occasionally "Manifestos" to describe how certain technologies should be used are written. An example would be the rest services technology [5] which is a critical technology, but has a specification that leaves a number of choices to be made during implementation. A Rest Manifesto can complete the specification so code is easier to maintain and it fits within the desired architecture.

Commercial options

Many may think that scientific software engineering can be replaced by commercial vendor's applications. This is true to an extent, but with the advent of the pure "button-pushing" applications, there can be a lot of confusion about what a particular button exactly does. Even worse, a feature may be purposely obscured with no documentation, so there is no way to know exactly what the button is doing. These tools can be quite flexible and even allow integration points, but at some point it becomes clear that certain tasks can't be effectively done within the "box" that is defined

by the commercial application. Finally, a commercial tool is trying to fit all needs and these circumstances typically lead to two solutions (1) the "least common denominator" solution where only features that can be agreed upon by large numbers of users are included which leaves a lot to be done manually or (2) the "cockpit" where everything is included and figuring out how to use the application can be very challenging.

On the other hand, there are very good commercial (e.g., OpenEye [6], ChemAxon [7]) and open source libraries (e.g., RDKit [8], BioJava [9], Apache [10]) that can be used to save a SSE from having to write low level code like SDF, PDB parser, chemical perception code, simple property calculation code, etc. Admittedly, libraries can be just as opaque as the other commercial applications, but they are at a much lower level and can be much more readily understood and validated. The library APIs are typically documented and may even have example code. This allows the SSE to write at a higher logical level where the libraries are used for the more mundane tasks or very specialized functions. Working at this level, the SSE has a great deal of control and a better understanding about what is happening relative to push-button applications.

Importantly, a SSE in the pharmaceutical industry typically has access to biological data that has been generated in a very uniform manner and in large quantities. Undoubtedly commercial vendors get access to some data from collaborations with pharmaceutical companies. They could also use public data, but that comes with a very big caveat about the uniformity of the data measurements that are used. In our personal experience, developing methods using in-house data is far superior to using the public data. Additionally, in the right organization, an SSE can even get experiments run to help validate algorithms and models.

Finally, writing software in-house can incur a large initial cost. However, if the in-house architecture is done using solid software engineering techniques, the code base can easily be maintained and extended. This saves the significant costs incurred with annual licenses and, as we have seen in the past can save money even in the 2–3 year timeframe.

Aside: getting the new methods to the scientists

Once a new algorithm is built and validated (e.g., predicting ADME or other properties, pharmacophore scoring models, structure prediction for a ligand–protein complexes) the next hurdle is to get these algorithms into the hands of discovery scientists. In the past, this was largely accomplished by a Scientist sending a specific request or question directly to the CADD member that developed the tool. This process has a number of drawbacks, the CADD scientist might be out-of-the-office or simply too busy to respond. Also they may accidently forget the request if made verbally or even accidently delete an e-mail. Finally, if they haven't used a source code control system and done full regressions of every release, there could be errors accidently introduced into the algorithm producing inconsistent and unreliable results. Mistakes, of course, can be made even if proper software engineering techniques are used, but they are much less frequent.

Computational Service Frameworks have evolved at a number of Pharmaceutical Companies. Minimally they serve as a repository for relevant and, hopefully, validated algorithms. Typically computational services frameworks also have access to a compute farm to process the incoming requests. At the higher end, a computational services framework may also perform automated format conversion (e.g., Corp ID to SMILES) and split up a perfectly parallelizable algorithm into chunks of inputs, run the chunks on independent compute nodes and aggregate the results. This produces the same results for the Scientists but with a significantly reduced wait time.

To make this kind of system work the client applications (e.g., SAR tool, HTS tool, library design tool, SBDD tool) that the scientists use daily, must be made aware of the services that are available. If done correctly, new services can appear in the client applications without requiring a rebuilding and releasing of the application. As another big benefit, every application gives consistent results by using the available services (e.g., cLogP for a given compound).

It is also important to build the services framework to be very scalable and reliable. When done well, it doesn't matter if there is a new service that has an error, the framework will protect itself (e.g., by running the new service in a separate process) and can even kill off services that are misbehaving (e.g., exceeding their estimated time of completion). Also, if designed correctly, the services framework can even employ compute clouds like AWS providing almost unlimited scalability.

By adding an additional layer on top of the services infrastructure it is possible to create an extract transform load (ETL) system. This requires introduction of some triggering event along with extensions to understand data sources and data sinks. The ETL system extracts the data from a data source, transforms the data using the service and stores the data into a data sink (mart). It is important to note that the data sources can be public data and incremental update systems can be implemented.

With a strong computational service infrastructure a "service publisher" can be built that allows not only the new algorithms to be exposed but can also run commercial applications wrapped in shell or python scripts. At the high end, it is even possible to publish KNIME/pipeline pilot protocols as new services that immediately appear in the

desktop applications. This allows all of CADD and other technically savvy scientists to publish new methods and make them accessible to project team scientists.

Conclusions

Research informatics (RI) is essential to any mature pharmaceutical company. Typically RI has a number of software engineers who can follow good software engineering practices. However these software engineers come from a broad range of experience and do not necessarily have a deep background in Science. When there is a low-level of scientific knowledge needed, then this can work well. However when the science gets more involved, it can be quite difficult for a RI software engineer, to understand the business needs as explained by a scientist who may have little technical background. This usually leads to very costly products that never quite work the way the scientist envisioned.

CADD computational chemists typically have a very deep knowledge and intuition learned over the years by working directly on discovery project teams. Some computational chemists do code, but this is typically on a smaller algorithm or script. It is very difficult for a computational chemist covering drug discovery projects to have time to do proper software engineering.

The scientific software engineer bridges these two disciplines with deep scientific knowledge and the time, desire and ability to follow good engineering practices. If there is going to be large scale innovation in CADD taking advantage of the growing knowledge base and being carried along with the ever increasing abilities of computational hardware and commercial and open-source software, scientific software engineers are essential.

Appendix: software processes

Although we didn't specifically advocate Agile as a software process, one reviewer requested more background on alternative processes that we have had experience with.

The standard cliché is that having some process is better than having no process at all. There are many variables and constraints that lead to the choice of a process: is it a well-defined problem or will scientific research be required, what is the experience level and characteristics of the members of the development team, will team members be working in the same location or even the same time zone, what is the flexibility given by management to try new processes.

We have been involved in very simple processes to very complex processes. One simple process was very effective with highly experienced software engineers. Basically a white-board discussion would take place and then the developer would write a "one-pager" describing what they understood the problem to be, and how they were intended to solve it. One complex process that we investigated was called "clean room" where each developer would have to mathematically prove how a piece of code would behave for all possible input parameters. This is a process we would not recommend because the resulting quality was not worth the amount of time and effort required.

We have experimented with hybrid processes where we mixed-and-matched techniques based on how to most effectively use the development resources. We mentioned Agile above as a process, but even in that case it has been adapted to fit the development teams needs. For example, we do not do pair-programming but instead every check-in is carefully reviewed by another developer. The daily scrums seem to work well, but we no longer make them pure stand-ups limited to 10 min. They are still short (less than 30 min) but they do include longer discussions about over-all design and are not strictly about what is underway or blocking. The sprint transitions are highly effective, they keep the developers motivated to reach the milestones and they give the users a chance to see the product as it grows and make changes and re-prioritize if needed. The drawback to the way we do agile is that it becomes difficult to predict overall timelines which are highly dependent on what changes the users would like. We are still trying to understand this and be able to better predict overall timelines.

Again, we did not advocate the Agile process will work well for all. The process that is most effective for a software team depends on the software teams and the environment.

References

1. Gehlhaar D, Rose P, Luty B, Cheung P, Litman A. The pfizer crystal structure database: an essential tool for structure-based design at Pfizer (submitted for publication)
2. Howe JW (2008), An integrated desktop computing environment for medicinal and computational chemists, ACS National Meeting, 17–21 August, #236—technical sessions http://www.acscinf.org/content/236-technical-sessions
3. Verkhivker GM, Bouzida D, Gehlhaar DK, Rejto PA, Arthurs S, Colson AB, Freer ST, Larson V, Luty BA, Marrone T, Rose PW (2000) Deciphering common failures in molecular docking of ligand–protein complexes. J Comput Aided Mol Des 14:731–751
4. Marrone TJ, Luty BA, Rose PW (2000) discovering high-affinity ligands from the computationally predicted structures and affinities of small molecules bound to a target: a virtual screening approach. Perspect Drug Discovery Des 20:209–230
5. Fielding RT (2000) Chapter 5: representational state transfer (REST). Architectural styles and the design of network-based software architectures (Dissertation). University of California, Irvine
6. OpenEye Scientific Software. http://www.eyesopen.com/
7. ChemAxon. https://www.chemaxon.com
8. RDKit. http://www.rdkit.org/
9. BioJava. http://biojava.org/
10. Apache. http://www.apache.org