The Many Faces of a Software Engineer in a Research Community

Cristina Marinovici, Harold Kirkham

Pacific Northwest National Laboratory
Electricity Infrastructure
Richland, WA

cristina.marinovici@pnnl.gov, harold.kirkham@pnnl.gov

Abstract

The ability to gather, analyze and make decisions based on real world data is changing nearly every field of human endeavor. These changes are particularly challenging for software engineers working in a scientific community, designing and developing large, complex systems. To avoid the creation of a communications gap (almost a language barrier), the software engineers should possess an 'adaptive' skill.

In the science and engineering research community, the software engineers must be responsible for more than creating mechanisms for storing and analyzing data. They must also develop a fundamental scientific and engineering understanding of the data. This paper looks at the many faces that a software engineer should have: developer, domain expert, business analyst, security expert, project manager, tester, user experience professional, etc. In authors' opinion, the conclusions drawn while developing a power-systems scientific software could be extended to describe the process of any software development project, and thus guide all current and future developers.

Biography

Cristina Marinovici *joined Pacific Northwest National Laboratory as a software engineer in August 2010. Before joining PNNL, she worked as software engineer at IBM, OR. She is responsible for the software testing and validation of projects modeling wind generation integration, power system operation and smart grid. She has a Ph.D. in Applied Mathematics from Louisiana State University, Baton Rouge, LA. She is a member of the IEEE, IEEE WIE, SWE and SIAM.*

Harold Kirkham *worked at American Electric Power, NASA's Jet Propulsion Laboratory in Pasadena, CA, and in 2009 he joined the Pacific Northwest National Laboratory, where he is now engaged in research on power systems. His research interests include both power and measurements. He is a past chair of the PES Instrumentation and Measurements Committee, and a Fellow of IEEE. He received the PhD degree from Drexel University, Philadelphia, PA.*

1. Introduction

In the scientific and engineering research community, there are two groups of people: scientists and/or engineers who model the real word problem, and software developers who write software. Each group has its own way of operating and developing products, and finding the common ground represents a difficult task. Usually, while working on common tasks, the two culture clash. Each tries to impose its culture on the other: software engineers try to impose a “traditional” software engineering culture on scientists (for example by using change management procedures), while the scientist try to impose their culture on the software engineers (by deliberately avoiding such methods). In order to have finalized and successful products, a common ground and common strategies should be found and applied.

For a software engineer working in a scientific community the job required skills extend beyond the regular job description posted by human resources (HR). The posted job description covers the standard skills required from the employee (the standard coding and software engineering skills), but cannot cover all of the skills that define the ‘adaptability’ of the software engineer. In scientific development communities, the developer should possess communication skills, flexibility to work with diverse software and hardware platforms in interdisciplinary teams, know-how to deal with ambiguities, continuously changing requirements and, last but not least, some domain knowledge.

It takes a fair amount of time and practice to grasp the concepts and understand the technical terminology of any scientific field. One of the factors that make a scientific computing development different from the commercial software developments is the complexity of the domain. There are intricacies in problem formulation, and maybe dependencies on the environmental constraints. Even more, the commercial software is based on common sense and has roots in day-to-day life. In the scientific research environment it may be fairly straightforward to implement a weather gathering interface, but will not be obvious what is necessary for the implementation of an economic dispatch model.

2. Skill set

As in any other profession, in software engineering it is necessary to poses diverse skill sets and to continuously expand them. The standard skill set is attainable through traditional curriculum training, or through some more practical matters as research or projects work. The skills worth mentioning from this set include:

- Any software engineer should be able to write code in an engineered way, to account for the future needs of the provided solution and to assure flexibility of the final product.
- It is desired from software engineers to be able use quality assurance (QA) and software testing skills and techniques. The shifts toward customer/user friendliness are changing the role of the software engineer, forcing him to be also the tester and the user of its product.
- The use of scripting languages represents one of the important skills. As John Ousterhout mentioned in his article, “scripting languages are designed for ‘gluing’ applications; they use typeless approaches to achieve a higher level of programming and more rapid application development than system programming languages. Increases in computer speed and changes in the application mix are making scripting languages more and more important for applications of the future.” (Ousterhout, 1998).
- The ability to process and clean data is another important skill. The operation of cleaning real-life data it is not an easy process, and it is the most important step since incorrect or inconsistent data can lead to false conclusions. Data cleansing is an iterative, expensive and time-consuming process.

- The software engineer is required to have the ability to apply numerical linear algebra, computational and statistical methods to model data in order to analyze and interpret the real life collected data.
- In recent years, the software engineer should scientifically understand the results of the analyzed data and employ these results in the decision making process.

To thrive in the scientific software development world, any software developer needs to learn continuously. This learning may come from formal degree programs with supplementary courses, or in seeking certifications in in-demand technologies. Lately, evolving your skill in response to the market changes will require blending of business and leadership courses with IT training. As developers are becoming more important to the business, they should be able to interact more with the customers and influence the way in which business value is delivered.

The science and engineering research community does not abide by the same criteria as the software engineering community. The rawness of software engineering and the complexity of scientific problems make collaboration between software engineers and scientists difficult. Typically, scientists' formal education in software development is limited; their knowledge of software development has been gathered informally from books, websites, from their colleagues, and often from the working code-sources they have encountered (Segal, 2011). Frustration due to miscommunication and misunderstanding lead scientists to perform the software design and development in isolation, writing clever code for specific architecture and forsaking the slower, more methodical approach to development favored by the software engineers. For scientists the emphasis may be on creating papers. Before publication, papers are peer reviewed and thoroughly scrutinized, but the review does not extend to the software used. The software can escape a lot of testing, and does not need to have flexibility to apply to other problems.

The goal is to unify the two cultures. The hope is to move scientific software development towards methodologies and techniques that have proved valuable for the software engineering community. The way to go is to impose that culture on the scientists, rather than the other way round.

Scientists are not generally early adopters of the software engineering techniques, They want proof that adoption of new methodologies will do no harm to their scientific product. It is desired to conduct the scientific development process in an iterative way, to work in small increments, and to define work-load for shorter iteration. Straight application of technological methodologies (i.e., Agile, lean, XP) may not be suitable for the scientific software developments, but modifying them to tailor every project needs may improve productivity and quality of the delivered products. Application of Agile methodologies, pair programming and creation of an open communication environment help one of the power-system projects in which the author worked on to achieve a usable and deliverable phase (Cristina Marinovici, HICSS 2013).

In scientific research communities, the work is partitioned per projects and pools of money. All the team members (scientists, engineers, software developers, analysts, testers or management) are paid from the same pool of money. The work will be carried on until the task is finished or the financial support runs out, whichever comes first. In this milieu, another good skill to have for a software engineer is the ability to estimate accurately. People are afraid to make estimations, since they are perceived as a prediction of the amount of effort and time required to deliver something specific. It is true that every estimate has risk, but providing accurate estimates helps management. They need good estimates for financial planning.

3. Domain Knowledge

Software development for scientific communities is challenging. Domain knowledge is not optional; it is essential. Software engineers who possess domain knowledge are valued. It is not easy to gain knowledge in scientific fields and to be able to "speak" the same language as the scientists and engineers. In order for software engineers to gain and grow their domain knowledge, a lot of practice and experience on dedicated projects is required, and continuous question asking is involved. Developers need to learn how to see through an end- user's eyes and how to employ the gathered information in the correct way. As it can be seen, the gain of domain knowledge is based on communication between the interdisciplinary team members. Many projects fail due to lack of proper communication between the scientists and developers.

This sort of difficulty has ancient roots. James Clerk Maxwell wrote in an 1877 report to Cambridge University (Maxwell, 1877): “It has been felt that experimental investigations were carried on at a disadvantage in Cambridge because the apparatus had to be constructed in London. The experimenter had only occasional opportunities of seeing the instrument maker, and was perhaps not fully acquainted with the resources of the workshop, so that his instructions were imperfectly understood by the workman. On the other hand the workman had no opportunity of seeing the apparatus at work, so that any improvements in construction which his practical skill might suggest were either lost or misdirected. During the present term a skilled workman has been employed in the Laboratory, and has already greatly improved the efficiency of several pieces of apparatus.” Maxwell described a two-way communication gap similar in essence to those we still have. These days, our “instrument makers” are software developers, but the effect of the communication gap is similar.

There is, between the user and the software developer, an excellent chance of a failure to communicate. Here is an example reported at a meeting of the IEEE Power and Energy Society recently (Berrisford, 2012). A power company supplying a large consumer (in Canada) changed out the electric meter. Afterwards, the consumer complained that the electric bill had increased, but they asserted that the load had not. The puzzle was that both the new meter and the old were digital, and there was no reason to think there was anything wrong with either. Indeed, investigation showed that both meters were performing accurately. It is a reasonable question to ask how this could be. In fact, the bill had increased not because the customer was being billed for greater energy consumption, the bill increased because the installation of the newer meter resulted in a penalty for the customer based on the power factor. The two meters did different calculations that (one would think) would give the same result, since both were (ostensibly) measuring the same thing: the power factor. The result of the measurement would have been the same had the waveforms been sinusoidal, but they were not. Distortion caused the difference.

The scientific field that the authors work in is power systems. Getting familiarized with the terminology and the physical meaning of the concepts is not simple. In many cases, one understand the words, but does not understand the engineering or scientific meaning. During one of our scientific software development projects, it became clear to the project team (the term project team is intended to means the whole group, including the developers, testers and power-system engineers), that there were communication problems. Team members did not have a common vocabulary and did not have a shared perspective on the software development process. Hence, misunderstandings arose related to the software’s feature set, priorities, architecture and structural requirements. These misunderstandings caused several false starts, and unproductive project tasks. Clearly, team internal communication affected the productivity of the project and the knowledge exchange process was reduced.

In Agile methodologies, the willingness of the team to assume a more cross-functional role when it comes to testing the built product is vital (Lee Henson, 2012). In our scientific development project, adoption of appropriate methodologies improved the team dynamic and its communication: the power engineers worked together with the developers and testers to properly translate the requirements into software. This type of knowledge sharing can be viewed as a way of saving everyone’s time. Migration from the initial system to the second one also helped individual knowledge to be transferred to the whole group.
In many cases, the power-systems engineers discussed the problem to be modeled with the team, helping the selection of the optimal approach. Since the designed simulator was solving problems that could not be verified by hand, in many cases the system testing required pairing with a power system engineer. For example, for a system with G generators considered as ways to meet the load in T time steps, there are $(2^G)^T$ combinations to examine. For any realistic situation, that is a large number; even 5 generators and 24 hours (a trivial system size) generates more than 10^{36} combinations. Also, there are many constraints that must be satisfied in the same time, and their combined effect cannot be easily predicted (M.C. Marinovici, ISGT 2013).
Analyzing the output required a lot of domain knowledge (i.e., power systems, optimization, distributed system behavior modeling, etc.), knowledge that only experienced engineers could have. In many cases

little was known about the proper and desired behavior of the software product and the well defined software testing and software quality strategies weren't enough. At that point, exploratory testing was performed. It helped us to define the next steps in testing and go beyond the obvious. Simultaneous learning, test design and test execution were incorporated in our testing strategy. This approach revealed gaps in the implementation that weren't evident from visual-inspection of the results and graphs.

4. Developing 'Adaptive' Systems

The need to embrace change becomes an accepted fact of life in software development environments. Any software project is subjected to continuous changes as it supports evolving user needs. Looking at the motivations behind the software changes during our software simulator life cycle, the following can be identified:

- Dynamic marketplace - changes required because of company changes, such as management changes or restructuring;
- Variety of environments - changes required to meet some external environment or outside-the-organization needs;
- Technological evolution - required to meet new technical demands and learning changes (due to gained knowledge as a result of individuals or group learning) (Cristina Marinovici, HICSS 2013).

The above changes also determine different perceptions of the system's value. With so many ruling factors involved in the software development lifecycle, the goal of system design is not robust systems, but rather the delivery of value to system stakeholders. In order to maintain value delivery over a system lifecycle, "either the system must change, or at least the perception of the system must change, in order to match new decision maker expectations" (Adam Ross, 2008).

As a team develops the system architecture, they must consider the inevitability of change in the future system. Robustness of the software can be added later on, in the development stages – in the design or implementation phase. In the development stages, a must have feature of the developed system is modularity. The software modularity may reduce change cost, though perhaps at a higher initial cost. Due to knowledge gained as a result of individual or group learning, a second system development may occur. This represents an improved and better engineered version of the old system. Usually, once the development of the smarter system started, many fixes and improvements, not included in the old system will be forced in, and the chain of changes will continue.

Embracing the changes and not fighting them will help progress, and will increase team competence and ability to assure product quality. Making mistakes is human, but learning from them will help us progress. The team will gain experience in assessing risk and preventing failures by thinking ahead in the development process. As system complexity increases, risks grow proportionally, but some of the failure cases can be caught earlier if knowledge is made available.

Most of the software products, developed for scientific research communities, are embedded in a network of complex dependencies (i.e. inter-products, inter-services and inter-suppliers). In this type of architecture, small changes in the system's input can produce unpredictable changes in the overall system behavior. Therefore, the software design problem becomes one of designing an 'ecosystem' where software components, project artifacts (i.e. design documents, requirements documents, bug tracking reports, schedules, etc.) and human resources are considered agents. The development of engineering models and methods for assembling software systems that can dynamically adapt to context and account for themselves is a difficult task and should be treated on project by project basis.

5. Roles and Responsibilities

As software practitioners, understanding of the roles and responsibilities within the team framework can affect the team productivity and communication. Trying to develop a solution outside the carefully monitored framework may lead to disaster. In the scientific research projects staffed by multi-disciplinary teams the roles and attributions are a bit different, displaying deviation from the standard. The team is governed by a sense of product ownership – every team member contributes creatively and has some control over the final product. All the efforts should be focused towards maximizing customer value.

The success of any team is based on communication, team-feedback, information exchange, cooperation, and self-organization. In our scientific development project, the team grown into a great organic team that is self-organized, established and assured an environment of good communication:

- The software engineer was not only the developer, but also the business analyst, the tester, the architect. Real life data are often noisy, corrupted by error, or awkwardly formatted. To work with these data, the software engineer needs not only to translate data from one format to another, but to do some amount of cleaning (or scrubbing) before usage. In many cases, the software engineer was also the product advocate. Incorporating customers' and teammates' feedback in the development process allowed the team to gain experience in assessing risk and preventing failures. By becoming the product user, the software engineer redefined the friendliness of the product, made it more intuitive and easy to use.

- The scientist was not only the field expert, but also the tester, the end-user and the customer. By the decisions that the scientist was making, his role was more of a lead. The scientist paired with the software team and helped in the modeling, optimization and validation of the software. This type of knowledge sharing can be viewed as a way of saving other's time and a direct way of improving software quality. The quality of the code improved, as a result of multiple peer reviews, and corner-cases testing.

- The manager was not only the manager, but the person who kept the team focused, who communicated a vision between the team and the customers. Management also determined metrics around value in the form of test coverage, team velocity, Agile adoption assessment, defect counts, and definition of done and so on. In this effort, the management tried, but was not always successful in avoiding the project management traps.

The size of the team is also important. In our scientific project development, the team has a small size, and adoption of new methodologies did not constitute a draw-back factor. There was a point in the product life-cycle when management tried to speed up development by adding more manpower. It is well known that a bigger team will require more channels of communication; people need time to get up to speed and their productivity is different from person to person. There is no linear dependency between the number of workers and their productivity. In the case of this simulator, the increased manpower represented a "surge," and the long-term picture was one of decreasing manpower.

Not in every project the application of standard methods and methodologies will work. Trying to find a better fit between a team and the methods that they are adopting is a good thing to consider. The team might experiment in adjusting the methods until they find what works best for them; there is no pre-can methodology to apply for every project. At this point in the product life-cycle, the knowledge gained can be utilized to improve the project's state of art, practice, and the team's education and training.

In any project, scientific or not, the adoption of dynamic methodologies gives team members the opportunity to experience different roles (scrum master, tester, user, developer, analyst, architect or manager). This change in roles provides a better perspective and understanding of the task(s), as well as a better estimation of the work-effort. In the multidisciplinary team the *us versus them* mentality is a fact of life, and therefore adoption of methodologies that provide one with the chance to experience work from the point of view of the other group is very valuable.

In our project, adoption of Agile methodologies and methods created an environment of camaraderie, increased team's respect on each other work and provided a better understanding of the difficulties and project hurdles. The learning process was bi-directional – scientists were learning from software engineers and vice versa. All the team members were familiar with the state and the direction of the project.

Good communication between the team members transformed the software engineers into product advocates, participating in discussion and demonstration of the product to clients and users. Usage of version control tools made the software product easy to deliver to interested parties with different modules incorporated. In this way, the team scientific staff has been convinced that reusing existing frameworks save time and effort, and it is more efficient than building their own from scratch. Therefore, for the whole project team, it was clear that understanding and implementing software engineering principles and other best practices is just as important as specialized knowledge.

6. Conclusions

Scientific software systems are growing larger, and their level of complexity is increasing. Software implementation of such systems requires proper software engineering and collaboration between scientists, engineers, developers and customers. Interaction among the computational-science and software engineering communities becomes more visible as software engineers become product advocates, participating in discussion and demonstration with clients and users. More dialogue and studies will consolidate this process. The two cultures have a lot to learn from each other.

References

Book:

Segal, Judith A. and Morris, Chris (2011), “Developing software for a scientific community: some challenges and solutions”, In: Leng, Joanna and Sharrock, Wes eds. Handbook of Research on Computational Science and Engineering: Theory and Practice. USA: IGI Global, pp. 177–196.

Journal Articles:

John Ousterhout, “Scripting: Higher Level Programming for the 21st Century”, IEEE COMPUTER, 1998

Cristina Marinovici, Harold Kirkham, Kevin Glass, and Leif Carlsen, “Engineering Quality while Embracing Change: Lessons Learned”, 46th Hawaii International Conference on System Sciences (HICSS), 2013

Berrisford, A.J., “Smart Meters should be smarter”, IEEE Power and Energy Society General Meeting, 2012 Digital Object Identifier: 10.1109/PESGM.2012.6345146

M.C. Marinovici, H. Kirkham, K.A. Glass, L.C. Carlsen, “Modeling Power System Operation with Intermittent Resources”, IEEE Innovative Smart Grid Technologies Conference, 2013

Adam M. Ross, Donna H. Rhodes, and Daniel E. Hastings, “Defining Changeability: Reconciling Flexibility, Adaptability, Scalability, Modifiability, and Robustness for Maintaining System Lifecycle Value”, Systems Engineering, 2008

Web Sites:

Hilary Mason and Chris Wiggins, “A Taxonomy of Data Science“, September 25, 2010, http://www.dataists.com/2010/09/a-taxonomy-of-data-science/ (accessed April 9, 2013)

Maxwell, J.C., Report on the Cavendish Laboratory for 1876 from the Cambridge University Reporter, ca May 1877) In A history of the Cavendish laboratory, 1871-1910, https://www.google.com/search?q=Report+on+the+Cavendish+Laboratory+for+1876+&ie=utf-8&oe=utf-8&aq=t&rls=org.mozilla:en-US:official&client=firefox-a (accessed April 20, 2013)

Lee Henson, “Empowering Agile Teams”, 2012, http://agiledad.com/index.php/downloads/ (accessed June 27, 2013)