Insights from the Software Design of a Multiphysics Multicomponent Scientific Code

Anshu Dubey[1]

[1]Affiliation not available

March 23, 2021

Abstract

Using simulations for scientific discovery requires that the software used in the simulations undergoes a rigorous design and development process similar to that of the lab instruments in the experimental sciences. To devise a good design methodology it is critical to understand the requirements, constraints and challenges. This article describes insights from the long-term stewardship of a multiphysics multicomponent software, FLASH, that was designed more than 20 years ago for astrophysics, now serves multiple communities, and has been successful in adapting to the changing world of high-performance computing.

Introduction

Over the last two decades the advances in computing hardware and computational mathematics have transformed the methods of scientific discovery. Computational advances in engineering have been instrumental in reducing, and sometimes eliminating, the cost of experiments in product design. While the benefits of computing are appreciated, the equivalence between scientific software and laboratory instruments has not been fully realized. Experimental scientists understand that the quality of their science depends upon the quality of their laboratory instruments. A similar understanding about software quality has not emerged in the computational sciences. Therefore, it is still difficult to persuade domain scientists that upfront investment in robust software design is in their best interest, and that it is an investment whose rewards are reaped for years (A. Dubey et al., 2018). The return on investment is tremendously valuable not only in terms of the quality of science they produce, but also in terms of the scientific productivity of their team members.

In the world of scientific discovery, software can be developed for many different purposes. Best practices for designing software may differ depending upon their use target. Here, I am going to focus on a design methodology that I found to be very useful in developing a multiphysics multicomponent software, FLASH, meant to be used as a community tool in astrophysics (A. Dubey et al., 2009; *Evolution of Flash , a Multiphysics Scientific Simulation Code for High Performance Computing*, 2013), that then went on to become a community tool for several other domains because its design enabled relatively easy customization for these other domains.

Design Challenges

Before going into the details of the design methodology, it would be good to explain what is involved in science through simulations and the challenges involved. The process of science through simulation roughly follows the flow shown in Figure 1. The phenomena of interest are captured in mathematical models, which

are then discretized so that numerical methods can be applied to obtain their solution. **Verification** is the process of ensuring that the model is implemented correctly. This is achieved through testing the software for correctness, stability, and convergence. The process of **validation** is meant to ensure that the devised model adequately captures the phenomenon of interest by comparing it with experiments and/or observations. Various feedback loops in the figure represent steps in the development process where it is important to do sanity checks and ensure that the progress is consistent with the objectives of the science being done.

Although Figure 1 captures the stages of development, it says little about the challenges at each stage. The entire design process is a balancing act between competing concerns. Well-known and understood principles for software design can be at cross purposes with the demands that are placed on the control flow by the requirements dictated by nature. For instance, the most basic good software design principles are modularity and encapsulation. However, real world is messy, and the model capturing its behavior may not lend itself to easy modularization. Similarly, one would like to design data layouts that minimize rearrangements in memory and maximize spatial and temporal locality for good performance. However, often different solvers have different optimal data layouts, and one is forced to consider trade-offs between the cost of rearranging data versus the slowdown caused by suboptimal layouts.

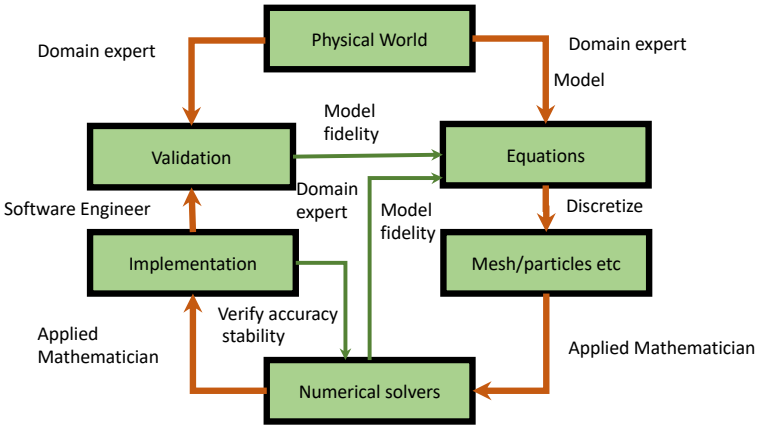

Figure 1: A schematic of how simulations are typically used in scientific discovery

Methodology

The map of expertise needed as shown in Figure 1 brings out another challenge; the development team members are likely to come from diverse fields with different expertise. The design needs to be cognizant of the technical necessities for an interdisciplinary team to be productive. One key principle is enabling people

to focus on what they know best without having to learn all aspects of the software; or in other words, **Separation of Concerns**. The term implies that software is developed in a way that different aspects of software do not interfere too much with one another. For example, in writing a parallel code for distributed-memory machines, a good practice was to separate all the local calculations from those that needed communication between processors. If the code is organized this way, for example, the applied mathematicians can do their algorithm development largely independently of the parallelization, while performance engineers can focus on optimizing scaling without having to know all the math.

This principle is well known and has been followed by many projects that have had success in their respective science communities (A. Grannan et al., 2019). The key to achieving separation of concerns relates to the need for modularization and encapsulation. And it brings back the question of lateral interactions between modules dictated by nature that can make encapsulation difficult. One way to achieve a semblance of encapsulation is to modularize on the basis of similar well defined functionalities, and provide explicit interfaces for lateral coupling if needed. Interfaces should be designed to achieve a good balance between adequate functionality and flexibility without unnecessary bloat. Figure 2 shows a workflow for achieving separation of concerns. The figure has two branches of development, one that pertains to the infrastructure and book-keeping part of the code, and the other is the part that implements the arithmetic of the computation. These two branches interact at a few points through interfaces where the first branch provides services needed by the second branch. Sometimes components in the second branch may undergo changes that need to be communicated to the first branch, hence the provision for *augmenting* the first branch.

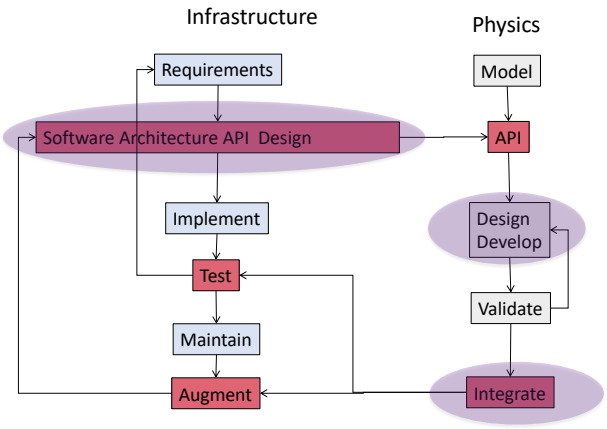

Figure 2: A design approach for achieving separation of concerns.

The other fundamental design methodology that is critical for long-lived nimble scientific code is implicit in the discussion above. It is based on recognizing that the design of the two branches ought to be treated

differently. The infrastructure, or the backbone of the code is the key to the robustness, performance characteristics, and extensibility of the code, and therefore its longevity. Being the service provider, this part of the code needs a thorough understanding of the design constraints imposed by the algorithms used in the science models. A careful exploration of the design space with prototyping and evaluation requires a non-trivial amount of upfront investment, but it is this investment that ultimately pays off. It is inevitable that at certain cadence even the infrastructure undergoes deep refactoring because both the hardware and the numerical methods constantly evolve. But, if designed well it should typically not need major overhaul through several generations of computing platforms.

The other part of the code, that implementing the model, should be treated as the client code with as close to a plug-and-play design as feasible. The evolution of science domains usually goes hand-in-hand with advances in model fidelity and the methods that implement the models. Because science campaigns typically take the code in uncharted territories for exploration, this part of the code is subject to continuous and rapid changes and should be designed such that these advances can be quickly assimilated. Ideally these advances and any customizations needed should either not involve the infrastructure at all, or at best require small tweaks to the infrastructure. Figure 3 captures the methodology for devising a software architecture that accommodates robust slow evolving infrastructure with flexible and nimble science solvers co-existing in the software.

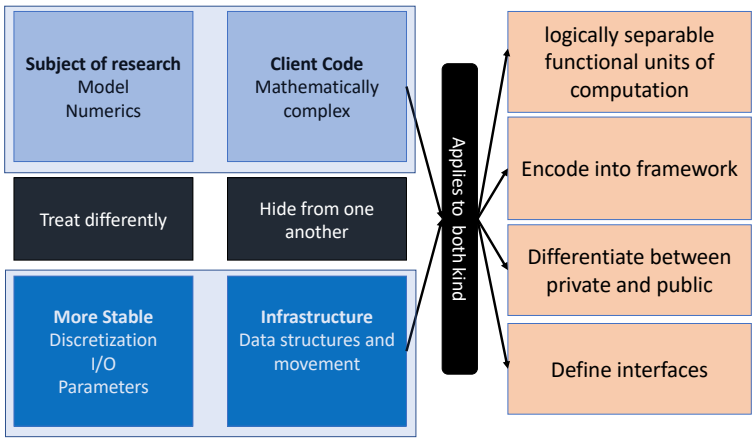

Figure 3: Overview of software architecture that accommodates slow evolving and fast evolving components

Since scientific software is developed for the purpose of exploration, it is rarely used in exactly the same way more than once. Every new science project using it is likely to tweak, modify, and customize it in some way. Often whole new capabilities may be added for a new project. Therefore, extensibility and customizability are critical, but non-trivial to achieve because it is difficult to anticipate the direction in which it may be needed in future. An added new challenge is the increasing heterogeneity in the computing platforms. Not even the mid-size clusters these days are without some form of accelerators that provide the bulk of computing power. And these accelerators differ from one vendor to another, and from one generation to the next by the same vendor. This necessitates making the separation of concerns more concrete, and interfaces

more flexible. One way to balance these somewhat conflicting requirements is to design for hierarchical access to the infrastructure as show in Figure 4. Here, the details of the numerical method are known only to the fully encapsulated part of the physics in the figure. The wrapper layer choses the functionality that may be exposed to other physics components, and permit lateral coupling between them as needed. The infrastructure in turn exposes its functionalities at different levels of granularity and hierarchy to the physics components. This kind of layering allows a range of transparency to the developers of physics modules. A less demanding physics module can opt for more transparency and interact with the infrastructure at a superficial level. However, should the developer of a physics module wish for greater control over the use of resources, and exercise the advanced features of the infrastructure, they can opt to interact with the infrastructure at a deeper level. The trade-off is between the extent of knowledge and understanding of the infrastructure and possibility of better performance. Such a design has the added advantage that it does not eliminate the possibility of including physics that demands deep interaction with the infrastructure to be computed. A critical design principle, therefore, is that whenever there is choice between transparency to the end-user or flexibility, the software architect should opt for flexibility.

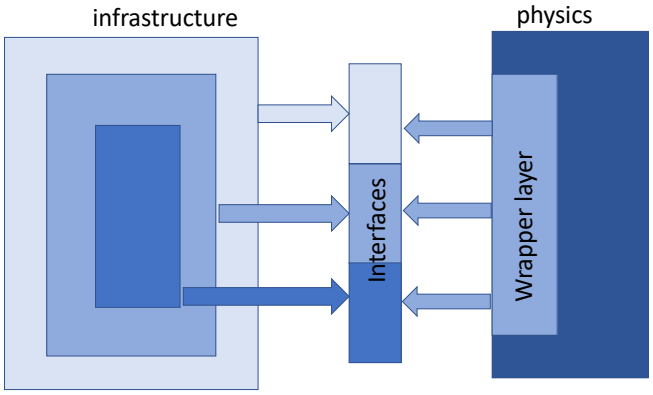

Figure 4: Interface design for layered access to framework features.

Conclusions

A very important question posed by the added heterogeneity in platforms is how much should the design process change to cope with it. Based on our experience in developing Flash-X (the new exascale code derived from FLASH) I have found that the basic design principles do not change, but the details get more complex. The boxes circled in Figure 2, combined with the layered interface design shown if Figure 4 was found to be sufficient for our purposes. The infrastructure has become a lot more complex (A. Dubey et al., 2020; *Orchestrating Data Movement and Computation for a Multiphysics Application At Scale*, 2021), but the basic principles of separation of concerns, modularity, flexibility, and well-thought-out interfaces still hold

Acknowledgement

This material was based upon work supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357.

Author biography. Anshu Dubey is a Computational Scientist in the Mathematics and Computer Science Division at Argonne National Laboratory. She received in Ph.D. degree in Computer Science (1993) from Old Dominion University, and her B.Tech degree in Electrical Engineering (1985) from Indian Institute of Technology, New Delhi. Her research interests are in the area of high performance scientific computing and software sustainability. Contact her at adubey@anl.gov

References

The Dividends of Investing in Computational Software Design – a Case Study. (2018).

Extensible Component-Based Architecture for Flash, a Massively Parallel, Multiphysics Simulation Code. (2009).

(2013).

Understanding the Landscape of Scientific Software Used on High-Performance Computing Platforms. (2019).

Distillation of Best Practices from Refactoring Flash for Exascale. (2020).

(2021).