Ten Principles for Creating Usable Software for Science

Lavanya Ramakrishnan
Lawrence Berkeley National Laboratory
lramakrishnan@lbl.gov

Daniel Gunter
Lawrence Berkeley National Laboratory
dkgunter@lbl.gov

***Abstract*—The volume and variety of scientific data being generated at experimental facilities requires the seamless interaction of the scientist's knowledge with the large-scale machines and software that is required to process the data. In the last few years, scientific software tools are being developed to address these increasingly complex workflow and data management needs. However, current approaches for designing systems and tools focus on the hardware and software of the machine and do not consider the user. Our experience shows us that user experience research needs to be tightly integrated with the software development life cycle for building sustainable software for science. It has become not just necessary, but critical, to consider the user interaction in the design of the entire system for data-intensive sciences that have complex human interaction with the data, software and systems. The dynamic nature of science projects and the complex roles of personnel in the projects makes it difficult to apply classical user research methodologies from industry. In this paper, we make three specific contributions towards improving the usability and sustainability of scientific software. First, we examine the software life cycle in science environments and identify the differences with commercial software development. Next, we outline ten principles we have developed to guide user engagement and software development and illustrate it with examples from our projects over the last several years. Finally, we provide guidelines to other eScience projects on applying the ten principles in the software development life cycle.**

***Keywords*—usability, escience.**

I. Introduction

Next-generation scientific discoveries rely on analyzing the massive data that is generated at supercomputing centers and experimental facilities. The key to maximizing insight from data is the seamless interaction of the scientist's knowledge with the large-scale machines and software that is required to process the data. Current approaches for designing systems and tools focus on the hardware and software of the machine and do not consider the user. Thus, current approaches to software design are not sufficient when designing next-generation scientific software to manage complex workflows and data for data-intensive applications.

Our experience shows us that user experience research needs to be tightly integrated with the software development life cycle for building sustainable software for science. User experience research has been used effectively for about 30 years in consumer electronics products and web interfaces. Although the sustainability of scientific software is subject to different constraints compared to commercial products, a critical factor in both cases is the adoption by the target user community.

In the past 30 years, a significant body of practice and knowledge has been developed on how to improve the quality of the user experience. Usability processes have been shown to not only improve user experience but also to heavily impact the design of the underlying frameworks. User research methods have been shown to result in gains in productivity, improve performance of underlying systems, produce cost-savings and reduce the cost of fixing software failures [1]. However, the dynamic nature of scientific collaborations and data analyses projects makes it difficult to apply classical user-centered design methodology to development of scientific infrastructure.

In the last several years, we have used user research methods in our research and software development projects. We have developed and modified practices to be suitable for the scientific community. We have used a variety of techniques – including participant observation, interviews, heuristic evaluation, and usability studies – to understand, collect, and refine user requirements and improve usability of software products. We have used these techniques in our projects to improve user interfaces for data exploration and analyses, workflow APIs [2], and building data management tools [3]. Our collaborations have included various scientific domains, including bioinformatics, materials, environmental, and climate sciences. These methods influence not only the user experience but also the design of the overall system and the software development life cycle. It enables teams to focus on user needs early and often and remove biases and preferences of the designers. Software based on user needs has a streamlined implementation path and has a greater chance of adoption.

In this paper, we outline ten principles we have developed to guide user engagement and software development. Our methods create a fundamental shift in the design and development of tools for next-generation scientific software by focusing on the user and the user experience. Specifically, we make three contributions in this paper.

- We examine the software life cycle in science environments and identify the differences with commercial software development.
- We outline ten principles for building usable software systems and illustrate with examples from our projects.
- We provide guidelines to eScience projects for applying the ten principles in the software development life cycle.

The rest of the paper is organized as follows. In Section II, we outline background on our projects, software development life cycle and discuss related work. We outline the unique characteristics of software for public science in Section III. We outline the ten principles and illustrate its impact in the context of our projects in Section IV. We discuss methodologies that can be used to apply the ten principles in the context of the software development lifestyle and present our conclusions in Sections V and VI.

II. Background

In this section, we detail the current software development life cycle practices and related work.

A. Software Development Life Cycle

The classic software development life cycle (SDLC) has at least three primary stages: *requirements, design, and implementation and testing*. These phases are present in all software development models from the Waterfall Model of the 1960s and 1970s [4], through the 1990s "Rational" Unified Software Development Process [5], to current agile and lean software process models. Some models have implementation and testing as separate stages, have additional stages for deployment and maintenance, and/or expand upon the requirements phase. But for the purpose of this paper, these three basic stages capture all the essential aspects.

A common mistake in our environment is to attempt to address usability issues exclusively during a single phase. For example, developers often assume they need to think about usability only in the design phase, i.e., to consider both requirements and implementation as relatively unaffected by usability concerns. Alternatively, some groups assume user interfaces can be built at the end. These practices cause two problems: (a) usability is considered too late to improve the requirements that drive the design, (b) usability is considered too early to help with feedback from the implementation and testing, which drive the next iteration. Usability should be considered at all phases of the software development life cycle and strongly influences the focus on user-desired functionality in the end product.

B. Related Work

This paper addresses, broadly, the process of designing usable software, or "usability engineering", as it is called in the discipline of Human-Computer Interaction (HCI). Much has been written on methods, applications, and tools for usability engineering in HCI textbooks, journals, and conference papers. An entire volume of Springer's Human-Computer Interaction series [6] is dedicated to the interplay between usability, HCI, and software engineering. This volume addresses many issues touched on here in more depth, but is not targeted at the needs of the scientific community. For example, none of the chapters deal with the peculiarities of the software for science environment. Our principles attempt to be accessible to people from software engineering, scientific, and HCI backgrounds, and streamlined for application in a very dynamic environment.

User research has been shown to be valuable in all stages of software development life cycle [7], [8], [9], [10], [11], [12], [13]. User research has been applied to various scientific domains including experimental cosmology, energy consumption, and biological light microscopy projects [14], [10], [13], [15], [16]. The importance of taking into account human factors has been recognized in prior human data interaction [17], [18], [19]. Previous work places users into groups of data producers and consumers and discusses how the decoupled nature of scientific research is increasing the complexity of data integration [20].

In [21], Nielsen lays out a step-by-step guide to applying usability methods at various stages in the software life cycle. In [22], Gould distills four basic principles of designing usable systems: (1) Early and continuous focus on the users, (2) Early and continual user testing, (3) Iterative design, and (4) Integrated design (i.e., all aspects of usability evolving together).

More recently, the popularity of "agile" design methodologies has led to an examination of the effect shorter iterative cycles of development have on the usability engineering process [23].

In previous work [24], the authors provide ten rules for developing usable software in computational biology. A couple of these rules, and in particular "Rule 2: Collect Feedback from Prospective Users", is similar in spirit to the principles described here. However, the purpose of the rules in the paper are to provide concrete guidelines for people who lack formal training in software engineering, i.e. scientist/developers as discussed in Section III below, who are creating their first software for general use. Thus, the rules mostly focus on the mechanics of writing usable software and do not explore more deeply how to integrate usability into all stages of the software development life cycle.

Previous work has also investigated how software is developed for scientific applications and identifies the iterative process and some of the challenges with applying software development practices [25], [26], [27], [28], [29], [30], [31]. This paper focuses on ten high-level principles that are particularly important to consider in the constrained and specialized environment of scientific software.

III. Software for Public Science as a Unique Environment

The principles described below grew out of our experience developing open (i.e. non-classified and mostly open-source) software, under public funding, to support a scientific mission. This environment has unique aspects not found at corporations, universities, or classified research laboratories.

In this context, we prefer to use the term "software for [public] science" to the terms "scientific computing" or "computational science", to make it clear that we are interested in the broader body of scientific software and not just software used for simulation models. We are concerned with the body of software that is used to execute the algorithms, to collect, store, analyze, and disseminate results (from simulation, experiment, or observation), and to support collaborative science in its many forms [32]. Most of this software has user-facing components (e.g., graphical interfaces, command-line interfaces, or APIs), and is affected by usability issues.

Invisible profit models. User satisfaction or user adoption are not first-class software metrics in the scientific environment. The reasons for this are partially cultural but, more fundamentally, are economic. The projects operate on grant money, in which value of work is determined indirectly by funders' impressions of value to the users, and not directly by consumers in a market. Also, unlike commercial products, the software is often a small part of the overall budget for a scientific mission. Similarly, another pervasive but subtle aspect of our environment is that work to improve software usability is often not explicitly called out in project deliverables and budgets. The ability to persuade key decision makers of the value of a type of work is crucial to its inclusion in project budgets. Usability research and development suffers from its relative invisibility in two of the main metrics used for judging software: performance and research publications. Usability *does* strongly affect the overall design and the adoption, but without ground truth of sales and profits, adoption numbers are difficult to measure accurately.

Exploratory iterative workflows. Usability for scientific software must also take into account some of the distinctive aspects of the scientific workflow. In our environment, scientists typically need to perform exploratory research that involves a complex combination of data and modeling, possibly using the computational resources in batch-scheduled supercomputers or moving terabytes of data for analysis. Work is often driven by short cycles tied to publication deadlines. Individual researchers often perform variations of the same analysis over and over, developing work-arounds for software quirks. Different researchers may use different software tools for the same tasks. Reproducibility of results is an ongoing challenge. This creates an environment that is both dynamic (many tools and systems) and resistant to change (repeatability and deadlines). As a result, the boundaries of the software development life cycle are less clear. Requirements gathering, design, and implementation are not strongly differentiated. The release phase is often implicit since users exchange software pieces through informal methods including emails, etc. Sometimes even getting code to be put under version control requires convincing the developers that this is not equivalent to a release.

Diverse and intertwined teams and roles. Many scientific projects consist of teams ranging from a handful up to hundreds of people who have widely varying domain knowledge and software skills. Scientists may know how to program, and programmers may have domain knowledge due to former training or long association with a given domain. There is not often a clear distinction between tool creators and tool users, and particularly on larger teams the bulk of the user population may be the project participants. The usability testing and requirements process needs to consider the diversity of roles on a project, how the new software will accommodate or replace existing tools, and the political realities of how the developer/scientists on the project can be brought into alignment with that decision.

Metrics of success. The metrics by which success of scientific software are measured are often not explicit, obvious, or even understood by the participants. As stated in a recent workshop on sustainable software for science [33], *it [is not] known if there is a common set of metrics for scientific software*. Some commonly used metrics are anecdotes from key projects or users, number of users/usages, number (and impact) of associated scientific publications, and technical measurements such as speedup or scalability. Unlike a commercial environment where number of users is always a major factor, in this environment the metrics are less clear. As we explained earlier, usability is rarely understood to be a separable important aspect of software. Nonetheless, human nature being constant, the usability of the software does have a large influence on the other metrics. The main difference in this environment is that the pathways by which this occurs may be hidden and unacknowledged.

Diverse user groups. Finally, a defining aspect of creating usable software for science is understanding the diverse user community. Even users within the same domain have varied requirements of the software (e.g., experimentalists and theorists). Scientific research is inherently specialized and fragmented, partially due to the depth of knowledge required and partially due to the exploratory nature of science itself – each researcher to some extent searches for their own niche in the field. The impact on software usability is that talking to a single user community may not actually help with the needs of their neighboring, and to the layman almost identical, researchers. Thus, the "guerrilla testing" model [34] of picking somebody at random to check the usability of the product will only get you so far, and to produce something meaningful you *must* think first about who you are trying to help, and how you are going to get feedback from all the sub-disciplines in that group.

IV. Ten Principles

In this section, we describe the ten principles that we consider important for developing usable scientific software and illustrate them with examples from our projects. These principles can be associated with phases of the Software Development Life Cycle (SDLC) described in Section II-A. This association is shown in Figure 1. In the requirements phase, apply the principles for understanding the users, their context, and their actual problems. In the design phase, focus on designing software that fits the users and use focused testing of prototypes to assess whether the design meets the user's metrics and trade-offs. In the implementation and testing phase, the principles point towards testing the software again with users, keeping in mind that the aesthetics are not as important as the mental model of the design, and prepare to carry what you have learned back to the next iteration of requirements.

P1 Solve the right problem first. Computer scientists often focus on the research problem that they are interested in, which might be a longer-term outlook. However, a good science engagement strategy will solve the right problem, the one that is affecting the users *today*, first.

For example, in a science engagement with users of LBNL's Advanced Light Source beamline, we were focused on learning their long term data problems. The goal was to modernize the data pipeline so users would be able to analyze the data in near-real-time instead of storing it and then manually feeding it to the analysis tools later. However, the user had a complex system of managing data on external hard disks and was simply running out of space. Losing experimental data, particularly when using large and expensive equipment, is very painful. In this case, it was critical for us to solve the current problem to engage the user as well as understand the other problems in the pipeline. Other challenges were masked by this overwhelming situation. Our initial work built confidence

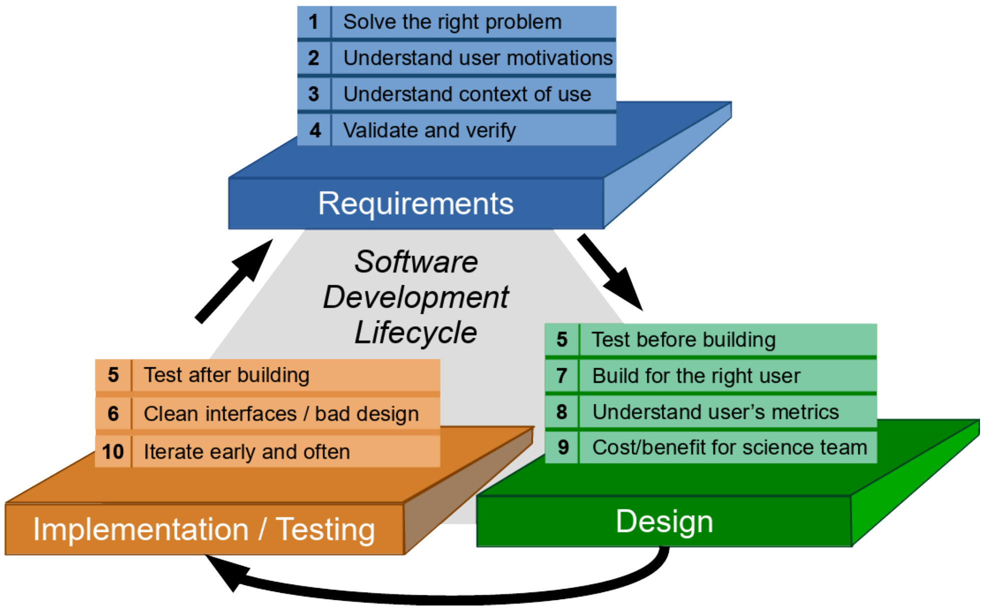
1 Solve the right problem
2 Understand user motivations
3 Understand context of use
4 Validate and verify
Requirements
Software Development Lifecycle
5 Test after building
6 Clean interfaces / bad design
10 Iterate early and often
Implementation / Testing
5 Test before building
7 Build for the right user
8 Understand user's metrics
9 Cost/benefit for science team
Design

Fig. 1. Mapping of the Ten Principles to the software development life cycle

for future engagement but also led to productive discussions about future work.

P2 Understand user motivations. It is important to understand the user's motivation to participate in the science engagement: what do they want to change and why do they want it. Understanding a user's motivations for the engagement is a good indicator of their time investment in the project and also willingness for change.

For example, complex scripts to manage simulation runs may be quite tolerable in the short run if they are relatively static and there are more serious issues elsewhere in the workflow. On the other hand, scripts that fail regularly, causing postdocs or staff to spend many hours chasing down bugs (on one project it became a running joke that a postdoc needed to "get a cot in their office" for round-the-clock tending to the workflows), will be at the top of the list for refactoring. It may not be obvious to an outsider under which category a given set of scripts falls, as the elegance and aesthetics of the script code may correlate badly with the importance of fixing them.

P3 Understand the context of use. It is important to understand the context of use of the system to be designed. If we don't know when, where, and how the scientists are doing their work, the system we build probably won't be useful. For example, in one of our projects, scientists were collecting data in knee-deep water. In this scenario, the users need to use mobile devices often in precarious conditions with spotty wireless signals. This insight could not have been gained during interviews or discussions in a meeting room. Observing the user's work pattern in the natural surrounding was not only necessary but essential to ensure success of the front-end and back-end design of the system.

Another example is scientists using a high-energy beamline at a national laboratory. Typically a researcher has only a few days of "beam time" to collect their data, and then possibly months to analyze it. In this environment, simple analyses that provide real-time feedback and help them guide their experiment are much more useful than advanced analysis tools that require too much time and effort to produce results. Again, insights can only be gained if we one were to observe a scientist's work practice at the beamline.

P4 Validate and verify what you have heard. We often struggle with dealing with the chaos of conflicting user requirements, including, "what do they actually mean?". User research processes enable you to work through these requirements to come up with clear recommendations and priorities.

For example, when designing a new system for bioinformatics analyses, we talked to users about what they needed for their research. Some users gave examples of how they needed programmable and extensible systems for complicated workflows. Other users wanted to be able to add their own software to the system. Taken alone, these requirements would have led to a system that exposed much of the underlying complexity to the expert user. However, in other user interviews we found that many users wanted software that would simplify and streamline their current struggles with the command-line tools. By talking to different groups of users with structured user research processes, we were able to avoid the trap of building a system for (only) the first set of users.

Another example of this approach is a methodology we developed to validate the requested features of an API. We interviewed users about how they "felt" about each feature, asked them to rank the features in order from most to least desirable, and finally asked them to describe conceptual problems or missing features. The combination of emotional response and numerical ranking feedback provided much better context for discussing the features with the domain scientists than we had gained from informal and unstructured discussions alone. More details of this experiment are available online [35].

P5 Test before building; test after building. It is critical to test how an user will interact with a system before even building the system, and subsequently test again with early prototypes. It is important to test mock-ups of design as well as early prototypes.

In another project, we built a workflow scripting environment for scientists. We used innovative new procedures for usability testing of the APIs. The first round was based on just using a paper version of the API to write pseudo-code. The results of this process improved specific features of the API [2]. After developing a product, we performed a second round of usability study to study the effectiveness of using that API more thoroughly, and over a longer time period. This kind of testing goes beyond simply answering bug reports to understand the way experienced users fit the software into their workflows.

P6 Clean interfaces can't make up for bad design. Many interface designs start from the mistaken assumption that the main challenge is good aesthetics, i.e., a "clean" design. However, we all have experience with UIs, either in software or the physical world, that look nice but are difficult to use. To avoid this trap, user interfaces need to be an essential part of system design so that they end up working with, rather than against, the target users' mental model(s). Similarly, a bad interface may cause problems for a good backend design.

For example, we found in one case that users preferred a file level UI to a new UI that was "clean" but abstracted from files, since their mental model of the data layout was already based on files.

P7 Build for the right user. It is important to identify who the users of the system are and build for them. "What would I do?" may be extremely misleading, if you are not representative of the end user. You have to continually ask yourself whether you are really modeling a real user, or just targeting someone who is convenient for you to think about (like yourself!).

For example, in a research project, we discovered that since the tool was initially built for experimentation purposes by computer scientists, there was no clear mechanism provided to access the output data from the runs. When the tool started getting used by science users, this was a hurdle and needed to be fixed.

In another project, the initial development of a graphical interface for bioinformatics tools included a number of advanced options that were well-understood by the developers and their community, but not at all obvious to a large and important subset of the user community. In response to this realization, many tool options were re-classified as "advanced", given reasonable defaults, and hidden until a user explicitly decided to view them.

P8 Understand the user's metrics. When computer engineers think about porting codes to HPC, they tend to think about performance of the codes. This is only one part of the picture. Productivity, publication deadlines, etc. are often the user's metrics and needs to be taken into account when designing the system.

When we talk to users, they consistently worry about their own time – time waiting to run, time debugging, etc. – as the primary metric. If the extra overhead of running on HPC is wasting too much of a user's time, performance gains of the code itself may be eclipsed.

P9 Cost / Benefit for the science team is different from the development team. It is important to remember that the cost/benefit analyses that a science team might do is often different from the analyses that a development team might do.

For example, a development team might be willing to invest in performance improvements in the code. However, the time investment in this might not look beneficial to the scientist if it does not improve the science result in some way. The money spent on better performance or code refactoring could have paid for new features or more analysis.

For example, on one project in bioinformatics, developers spent months dramatically improving the speed of access to large data stored on tertiary backups. However, the project lead later explained that, while impressive, this represented only about 1% of their analyses, and did not help the other 99% of the data being stored on disk in databases.

P10 Be willing to iterate (early and often). It is important to be willing to iterate and engage the users early and often. It is critical that developers don't get attached to their designs and are willing to pivot the development process.

For example, in one project we planned to use Docker containers (portable execution environments) as the mechanism for releasing new versions of the software to the users. In early project meetings, we showed slides of the design and nobody objected. However, when we later asked the users directly if they would actually run the Docker containers, it turned out that all the users were "experts" who had no trouble updating their installation from Github directly. Moreover, they needed to integrate with non-free commercial software that could not easily be included in the Docker container. This conversation took about five minutes, yet saved a couple of weeks of effort wasted on refining Docker release mechanisms that nobody would use.

V. Applying the Principles in the Software Lifecycle

In this section, We provide a set of *processes* for applying the principles in each phase of the software development life cycle (SDLC), as described in Section II-A and Figure 1. Figure 2 connects the specific processes used, in each phase of the SDLC, to the corresponding principle(s). The diagram also categorizes processes by whether they primarily involve user interaction, planning and/or design, or programming.

A. Requirements Phase of SDLC

Principles 1 through 4 are important during the *requirements* phase of the software life cycle for understanding the users, their context, and their actual problems. In the requirements phase, the three primary processes are a) to identify the target users, b) perform interviews and participant observations, and c) create use-cases or mockups and validate these by discussing them with the target users.

Identify target users (P1, P2). It is critical to identify at the outset which scientists will serve as representatives of the intended audience of the tool. The challenges of doing this in a diverse project were discussed earlier. An approach that we have found effective is to ask project management for a high-level breakdown of the different sub-groups, and then to go into those subgroups and ask some very simple questions: (a) what are the main things you do, (b) how do you do them now, (c) if you could improve *one* thing, what would it be? The answers to these questions help us infer the targe audience for a new tool.

One thing that we learned *not* to do was to assume that project management can identify target users for us. Too often, we found that management's knowledge of what people did

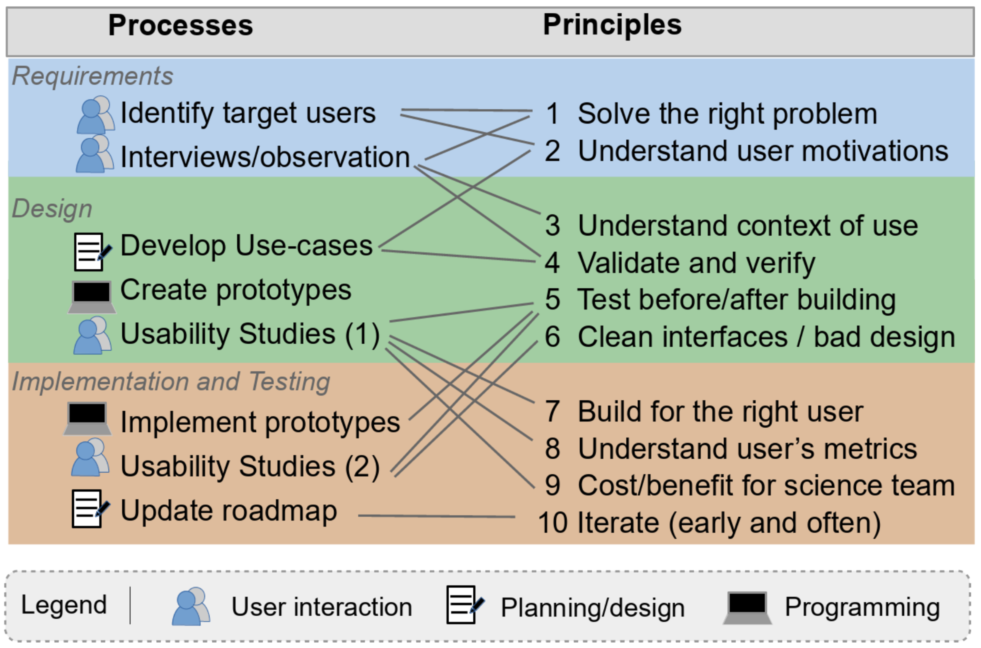
Processes
Principles
Requirements
Identify target users
Interviews/observation
Design
Develop Use-cases
Create prototypes
Usability Studies (1)
Implementation and Testing
Implement prototypes
Usability Studies (2)
Update roadmap
1 Solve the right problem
2 Understand user motivations
3 Understand context of use
4 Validate and verify
5 Test before/after building
6 Clean interfaces / bad design
7 Build for the right user
8 Understand user's metrics
9 Cost/benefit for science team
10 Iterate (early and often)
Legend
User interaction
Planning/design
Programming

Fig. 2. Application of the Ten Principles in each phase of the the Software Development Life Cycle

day-to-day was missing important details. Also, you will note that none of the questions asks the users directly whether they actually are the target audience. We have found that there are at least two problems with this approach. First, users often misinterpret what the new system is actually supposed to do and give misleading answers. At this stage, there is often a terminology and knowledge gap between the target users and the software developers. Second, people may skew their answers depending on whether they are interested in engaging with the software design process or not. Asking people directly about their work removes these potential filters that can bias the answers.

Interviews and observation (P3, P4). The team must understand the work practices, work goals, what the scientists would like to achieve, and current similar tools. This can be accomplished by interviews and participant observation (i.e., watching people work).

We have found that scientists are usually eager to brainstorm about ways to improve their current setup, i.e., to engage in design discussions. Sometimes, though, you find that the team repeatedly cancels or postpones discussions to deal with "other issues". Rather than interpreting this as a rejection of the process, it is more likely that you are not focusing on the right problem (**P1**), and you should probably stop and re-evaluate the issues being included in the design discussions.

Develop use-cases (P4). The use-cases are high-level descriptions of important goals, that come from the interviews and observation. Typically, the designer and developer comes up with the use cases, but another approach is to try and get some members of the science team to write down the use-cases from their perspective. Either way, it is very important to try and circle back to a majority of the users you talked to in the previous stages and talk through the use cases with them. Inevitably, there are misunderstandings in terminology and goals that are revealed.

When introducing ideas for new software into the existing workflow, we are essentially asking for scientists to spend their time on our task instead of theirs. Learning a new software tool may seem fun to a computer scientist, but is often seen as a necessary evil by a user. Scientists, like developers, are acutely aware of aspects of their workflow that either waste time or reduce quality of results. Scientists will be more willing to try a new tool if it addresses a problem they have with their current workflow.

It is also important to remember that part of the value of user research to the team, in addition to the technical skills, is as a force for change. Do not succumb to "group think" and complacency about the current solution, even as you remain practical about what can be done right now. At the appropriate times, it is important to advocate the value of re-thinking and re-factoring design and usability aspects of the software, as a complement to short-term fixes.

B. *Design Phase of the SDLC*

Principles 5, 7, 8, and 9 are important during the design phase of the software life cycle for designing software that fits the needs of the users and assessing whether the design meets the user's metrics and trade-offs. In this phase, the two primary processes are to create prototypes and perform usability studies.

Create prototypes (P5. In the first iteration for a use-case, develop a low-fidelity prototype that is believed to address the work goals identified by that use-case. For a graphical user interface, this may take the form of a paper prototype or a quick "clickable" prototype. API prototypes can take the form of a document describing the function definitions. Documentation and example code aid in early usability tests.

In subsequent iterations, the prototype will be refined, e.g., moving from paper to a detailed wire-frame or a semi- functional HTML page. For API design, a high-fidelity prototype will take the form of partially functional code.

Conduct a Usability Study (P5). Conduct a usability study with target users identified during the requirements phase. Usability studies provide a methodology to conduct and learn from these tests. Traditionally, the methodology has been used for web interfaces [36], but we have found it is applicable to APIs as well. We recommend the following guidelines:

a. Preparation time: Allocate a few minutes for the test participant to absorb the material and scenario, particularly when testing an API.
b. Scenario: Ask users to work through the scenario, thinking out loud.
c. Feedback: Provide feedback on demand but do not intervene unless a participant cannot continue with the exercise. The feedback provided should be minimal and guide the scientist towards the solution rather than provide the solution.
d. Follow-up. The study should be followed up with an open discussion. The discussion should cover aspects of the tests the users felt comfortable or uncomfortable with. Additionally, ask followup questions to see whether the material was understandable and if the participants can see themselves using it for their work.

The usability test results should answer the questions of whether the prototype is built for the right user (**P7**), supports a user's metrics (**P8**), and is clearly worth the effort in terms of benefit to the science team (**P9**). The diversity of scientific users pointed out in Section III needs to be considered. Feedback from different classes of users will answer different kinds of usability questions – *Does it do what we wanted?*, *Is it sensible and clear to all users?*, *Does it make sense in the broader context.* The answers to these questions will in turn determine whether a redesign is required, project priorities need to be changed, or both.

Sometimes, there might be a need to completely redesign the software at this stage. It will be many times more costly to redesign the software after it has been implemented or deployed. In our experience, the development team tends to be more attached to their work than they want to admit. Pride in your work is good, but this is really a reflection of the different cost/benefit for the development and science teams (**P9**). Elegant software is not in itself a benefit for the science team, but it can lead to a series of rationalizations for not redesigning a component that really needs to change. However, sometimes redesigning software might be essential for software maintenance. If you *think* you might need to redesign part or all of the system, you need to seek out some other opinions, e.g., talk to one or more colleagues who are not involved in the project, and listen carefully to their feedback.

After evaluation, repeat the prototyping and testing cycle, increasing the fidelity and functionality of the prototype at each iteration.

C. Implementation and Testing

Principles 5, 6, and 10 apply in the implementation and testing phase of the SDLC for testing the software again with users, keeping in mind that aesthetics are not sufficient and the mental model of the design needs to be considered. The team needs to prepare to carry what you have learned back to the next iteration of requirements phase. In the implementation and testing phase, the primary processes are to implement the software, conduct usability studies, and update the roadmap.

Implementation. The software to be implemented will be a continuation of the prototypes developed during the design phase. Although your implementation will draw on all the knowledge gained to this point about how users work and what the interface should look like, in reality there will be a number of ad-hoc micro-design decisions that still need to be made. It is not practical to conduct a full usability study for each of these, but it is very useful to have some users who are willing to let you informally bounce these decisions off them as a lightweight way of testing (**P5**) the usability as you go. In the usability studies, you will learn more about how well these friendly users correlate with the larger group, and improve your interpretation of their feedback.

Conduct a Usability Study (P5, P6). Once you have a working prototype, you need to repeat the usability studies that you conducted in the design phase (Section V-B). In addition to the feedback, we have found that it is very satisfying to the users to see that their earlier input resulted in real changes to the software.

When reviewing an interface with potential users, it is not enough to ask simple, idealized tasks; make sure you ask them how they would achieve *most complex* tasks they perform today, since after all that is the intent of the software. Also simply ask directly whether the software is actually easier to use than what they have right now. This approach will help to catch lingering problems with the assumed "mental model" (**P6**), i.e., the way in which the users' workflow and artifacts are connected by the software and its commands. Mental model issues can be missed with simplistic views of what the users do, and these can lurk in the steps skipped during testing of prototypes.

Update roadmap (P10). For any development process, you will have some form of roadmap that states where to focus effort in the next iteration. You should now be armed with information to choose a sensible focus. In terminology recently made popular by The Lean Startup [37], you are using what you learned from the "Measure" and "Learn" phases of the cycle to choose the next Minimum Viable Product (MVP) to develop in the subsequent "Build" phase.

As the lean startup approach emphasizes, a key ingredient to success is to not assume more knowledge than you actually have, and stay modest in the scoping of the next iteration. For example, just learning that users need a certain set of visualizations of their data does not mean you should drop everything and design a general purpose visualization framework. Instead, take the shortest path to actually creating the visualizations that are most needed (which may, of course, involve using an existing visualization framework) and use the next iteration to determine what kind of additional framework, if any, is necessary. We have found that scoping to an MVP in each iteration is just as useful and applicable in the scientific software context as it apparently is for software startups.

VI. Conclusion

The ten principles described in this paper draw on our experience with the creation of usable scientific software in dynamic science projects. Our experience shows us that usability principles and user experience research need to be tightly integrated with the software development life cycle for building sustainable software for science. This paper provides a foundational framework for incorporating users into current software development life cycles of scientific software.

Our broader goal is to shift the culture in scientific computing to make usability concerns an equal partner with those for hardware and software development. This will require a large community awareness and change. We believe this work provides a starting point for a conversation that can lead to discussion about workshops at eScience and including usability in the review criteria of proposals and papers (akin to recently introduced data management plan or reproducibility efforts).

Acknowledgment

This work and the resources at NERSC are supported by the U.S. Department of Energy, Office of Science and Office of Advanced Scientific Computing Research (ASCR) Energy under Contract No. DE-AC02-05CH11231, program manager Lucy Nowell. The authors would also like to thank Sarah Poon, Deborah Agarwal, and others for their valuable suggestions and feedback.

References

[1] T. Gilb, *Principles of Software Engineering Management*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1988.

[2] L. Ramakrishnan, S. Poon, V. Hendrix, D. Gunter, G. Z. Pastorello, and D. Agarwal, "Experiences with user-centered design for the tigres workflow api," in *2014 IEEE 10th International Conference on e-Science*, vol. 1, Oct 2014, pp. 290–297.

[3] D. Ghoshal and L. Ramakrishnan, "MaDaTS: Managing data on tiered storage for scientific workflows," in *ACM Symposium on High Performance Parallel and Distributed Computing (HPDC'17)*. ACM Press, 2017.

[4] W. W. Royce, "Managing the development of large software systems: concepts and techniques," in *Proceedings of the 9th international conference on Software Engineering*. IEEE Computer Society Press, 1987, pp. 328–338.

[5] I. Jacobson, G. Booch, J. Rumbaugh, J. Rumbaugh, and G. Booch, *The unified software development process*. Addison-wesley Reading, 1999, vol. 1.

[6] A. Seffah, J. Gulliksen, and M. Desmarais, *Human-Centered Software Engineering - Integrating Usability in the Software Development Lifecycle*, ser. Human–Computer Interaction Series. Springer Netherlands, 2006. [Online]. Available: https://books.google.com/books?id=NkMp0_spR5QC

[7] W. Albert and T. Tullis, *Measuring the user experience: collecting, analyzing, and presenting usability metrics*. Newnes, 2013.

[8] C. R. Aragon and S. S. Poon, "Designing scientific workflow management systems for data-intensive astrophysics projects," in *Designing Cyberinfrastructure to Support Science Workshop, CSCW 2008: ACM Conference on Computer Supported Cooperative Work*, 1998.

[9] S. S. Poon, R. C. Thomas, C. R. Aragon, and B. Lee, "Context-linked virtual assistants for distributed teams: an astrophysics case study," in *Proceedings of the 2008 ACM conference on Computer supported cooperative work*, 2008.

[10] T. Boellstorff, *Ethnography and virtual worlds: A handbook of method*. Princeton University Press, 2012.

[11] S. B. Davidson and J. Freire, "Provenance and scientific workflows: challenges and opportunities," in *SIGMOD 2008*. ACM, 2008.

[12] D. Nicolini, *Practice theory, work, and organization: An introduction*. Oxford University Press, 2013.

[13] J. A. Maxwell, *Qualitative Research Design: An Interactive Approach*, 3rd ed., ser. Applied social research methods series, v. 41. Sage Publications, Inc, 2012.

[14] C. Aragon and S. Poon, "The impact of usability on supernova discovery," in *Workshop on Increasing the Impact of Usability Work in Software Development, CHI 2007: ACM Conference on Human Factors in Computing Systems*, 2007.

[15] C. Macaulay, D. Sloan, X. Jiang, P. Forbes, S. Loynton, J. R. Swedlow, and P. Gregor, "Usability and user-centered design in scientific software development," *IEEE Software*, vol. 26, no. 1, pp. 96–102, 2009.

[16] J. Nielsen, *Usability engineering*. Elsevier, 1994.

[17] P. Ziegler and K. R. Dittrich, "Three decades of data integration-all problems solved?" in *IFIP congress topical sessions*. Springer, 2004, pp. 3–12.

[18] A. Halevy, A. Rajaraman, and J. Ordille, "Data integration: the teenage years," in *Proceedings of the 32nd international conference on Very large data bases*. VLDB Endowment, 2006, pp. 9–16.

[19] L. J. Seligman, A. Rosenthal, P. E. Lehner, and A. Smith, "Data integration: Where does the time go?" *IEEE Data Eng. Bull.*, vol. 25, no. 3, pp. 3–10, 2002.

[20] C. Goble and R. Stevens, "State of the nation in data integration for bioinformatics," *Journal of biomedical informatics*, vol. 41, no. 5, pp. 687–693, 2008.

[21] J. Nielsen, *Usability Engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

[22] J. D. Gould, "How to design usable systems," in *Human-computer interaction*. Morgan Kaufmann Publishers Inc., 1995, pp. 93–121.

[23] M. Brhel, H. Meth, A. Maedche, and K. Werder, "Exploring principles of user-centered agile software development: A literature review," *Information and Software Technology*, vol. 61, pp. 163 – 181, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584915000129

[24] M. List, P. Ebert, and F. Albrecht, "Ten simple rules for developing usable software in computational biology," *PLOS Computational Biology*, vol. 13, no. 1, p. e1005265, 2017. [Online]. Available: https://doi.org/10.1371/journal.pcbi.1005265

[25] D. Heaton and J. C. Carver, "Claims about the use of software engineering practices in science: A systematic literature review," *Information and Software Technology*, vol. 67, pp. 207–219, 2015.

[26] J. Segal, "Software development cultures and cooperation problems: A field study of the early stages of development of software for a scientific community," *Computer Supported Cooperative Work (CSCW)*, vol. 18, no. 5-6, p. 581, 2009.

[27] C. Goble, "Better software, better research," *IEEE Internet Computing*, vol. 18, no. 5, pp. 4–8, 2014.

[28] Z. Merali, "Error: why scientific programming does not compute," *Nature*, vol. 467, no. 7317, pp. 775–777, 2010.

[29] "How computers broke science and what we can do to fix it," http://theconversation.com/how-computers-broke-science-and-what-we-can-do-to-fix-it-49938.

[30] J. Segal and C. Morris, "Developing scientific software," *IEEE Software*, vol. 25, no. 4, pp. 18–20, 2008.

[31] J. C. Carver, "Software engineering for computational science and engineering," *Computing in Science & Engineering*, vol. 14, no. 2, pp. 8–11, 2012.

[32] M. R. Benioff, E. D. Lazowska *et al.*, "Computational science: ensuring america's competitiveness," *Report to the President, President's Information Technology Advisory Committee, Washington, DC*, 2005.

[33] D. S. Katz, S. T. Choi, K. E. Niemeyer, J. Hetherington, F. Löffler, D. Gunter, R. Idaszak, S. R. Brandt, M. A. Miller, S. Gesing, N. D. Jones, N. Weber, S. Marru, G. Allen, B. Penzenstadler, C. C. Venters, E. Davis, L. Hwang, I. Todorov, A. Patra, and M. de Val-Borro, "Report on the third workshop on sustainable software for science: Practice and experiences (WSSSPE3)," *CoRR*, vol. abs/1602.02296, 2016. [Online]. Available: http://arxiv.org/abs/1602.02296

[34] S. Krug, *Don't Make Me Think: A Common Sense Approach to the Web*

(2nd Edition). Thousand Oaks, CA, USA: New Riders Publishing, 2005.

[35] S. Poon. (2015) Evaluating the proposed capabilities to be supported by an api. [Online]. Available: http://lbl-udablog.blogspot.com/2015/12/evaluating-proposed-capabilities-to-be_1.html

[36] S. Krug, *Rocket Surgery Made Easy: The Do-It-Yourself Guide to Finding and Fixing Usability Problems*. New Riders Publishing, 2009.

[37] E. Ries, "The lean startup— the movement that is transforming how new products are built and launched," *New York: Crown Business*, 2012.