End User Software Engineering: Auditing the Invisible

Joshua B. Gross
School of Information Sciences and Technology
311B IST Building, Penn State University
University Park, PA 16802
+1814 865 9838
jgross@ist.psu.edu

ABSTRACT
In this paper, I will describe the need for new tools to engage end users in the software engineering process, and then describe an example of such a tool in a brief scenario.

INTRODUCTION
In his seminal article on the problems of software development, Brooks [2] cited the essential invisibility of software as one of the essential or natural problems that could never be resolved. His point is accurate, but limited in its perspective. Work in research and industry has shown that visibility can be lent to software, but that visibility is largely a veneer; an attempt to use physical or mechanical metaphor to explain the processes described in software.

Unfortunately, this approach is inevitably limited by the value of the metaphor. New approaches to visualization are necessary, ones that rely not on metaphor, but on new, artificial languages that bridge the gap between how computers operate and how the human mind functions. These languages must also account for the pragmatic applications of the software; this aspect is perhaps the most problematic, but the most critical to bridging the gap.

THE SOFTWARE ENGINEERING PROBLEM – REDUX
It seems almost superfluous to speak about problems related to software engineering. The norm for software engineering projects has been late delivery of overbudget, substandard, incomplete products. This is for the lucky projects that deliver at all; the United States has attempted to replace its air traffic control software three times in the past twenty years, but despite the millions of US dollars spent, no such replacement is available.

Much of the problem can be traced to software engineering (SE) as a discipline. Many software development processes begin (implicitly or explicitly) with the statement "assume fixed requirements." Even if a process to capture such requirements were available, fixed requirements are a myth on the order of Sisyphus.

Numerous solutions to the problems of software engineering have been proposed, and inevitably they have offered some improvement. Some rely on tools (e.g. CASE tool, Business Rules), while others rely on processes (e.g. Extreme Programming and Rational Unified Process), and others on visualizations that allow for design and explanation (e.g. the Unified Modeling Language).

All of these do address some aspect of what Brooks referred to as "accidents" of software development, but none solve the problem. Several researchers and practitioners have proposed that software needs either a "paradigm shift" or "sea change" to completely rewrite how software is built. Unfortunately, none has yet been successful.

It is not the aim of this paper to propose such a change; the hubris required to attempt such (especially in a three page workshop paper) is beyond this author. However, there are clues that show how existing tools, processes, and languages can be integrated and extended to improve software development, or, at the very least, lend it additional visibility.

THE END-USER SOFTWARE ENGINEER
When we consider a profession such as software engineering, we must initially ask whether end users can perform this function. As mentioned above, there is no reason to assume that they would be much worse than trained software engineers.

However, we cannot reasonably expect non-professionals to perform certain tasks. Designing taxonomies, creating flexible architectural components, and building the unexciting, exceptionally invisible interstitial software that manages the tiers of a business application are tasks with limited rewards for anyone other than a professional developer. Building a small application (e.g. in a spreadsheet) is within the grasp of many end users, but building an enterprise application is not.

So if end users cannot be software engineers, and developers cannot be domain experts, we must meet somewhere in the middle. Perhaps the best metaphor would be that of a library. A patron cannot be expected to build and organize the library, but similarly no librarian can fully understand the content and import of each volume. A library is only partly a building filled with books and periodicals; it is a meeting of minds, skills, and interests.

A SOFTWARE MEETING OF THE MINDS

Eric Evans has suggested that users, domain experts, and developers must jointly form a new "ubiquitous language" [3] that is shared and used by all people working on building a particular system. This language creates the possibility of an artificial space in which many abstract problems of the domain can be made concrete and "solved", at least for the limited purpose of the application.

This idea is excellent, and shows a growing trend to incorporate the user more fully into the software development process. Another example can be found in Extreme Programming, in which an "on-site customer" is one of twelve required practices [1]. While these practices are growing in popularity, they often hit a roadblock due to disengaged and uninterested users.

As with Carroll & Rosson's "active user", the "engaged user" is something of a paradox, concerned with productivity, possibly at the expense of quality. The engineering gestalt, which emphasizes robust, reliable systems, cannot be expected to capture the hearts and minds of users everywhere.

THE NEED FOR CONVERGENCE

Despite potential limits of interest, we should not dismiss end-user software engineering. Unfortunately, we haven't sufficiently mastered software production in order to allow us to completely automate the process. The 'Big Red Button' idea that magically translates requirements to code is not yet a reality.

The question arises, then, what role end users can take in the software engineering process? However, a slight modification of the question is more interesting: how can we modify the software engineering process to accommodate end users and improve the overall productivity and quality of the product?

This question allows us to find a convergence: a place where the needs of the various stakeholders in the process and outcome of large-scale software development can come together. In theory, any such convergence is a good thing, but as discussed above, the different interests and skills make a positive outcome seem unlikely.

ANSWERING THE CALL

Since we cannot yet solve software engineering problems *en masse*, our interim question must be how to take advantage of this convergence of need. This is not a question with a single answer, but this paper proposes that at least one answer can be offered and developed into a useful practice.

Two recent laws enacted in the United States have changed how businesses use and view information systems. HIPAA (the Health Insurance Portability and Accountability Act) regulates how all medical data is transferred, and the Sarbanes-Oxley Act has made corporate officers legally responsible for misreported corporate earnings and other financial statements.

In both cases, the new laws force organizations to produce a level of traceability that they have never had to deal with before. In addition, because both civil and criminal penalties can be imposed, these new business practices must be taken seriously. Interestingly, software developers are largely immune from penalties, but as others (end users) are not immune, they are greatly concerned with ensuring that the systems they use function properly.

Software engineering has an answer; software quality assurance (SQA), which is concerned with ensuring that software is validated (matched to requirements) and verified (technically correct). Unfortunately, SQA activities are seen as the least engaging, and while tools have improved (e.g. for requirements traceability and unit testing), we still have a problem that end users are probably unwilling to tackle.

I propose, instead, that we incorporate a new method of investigation, auditing, and create new tools to support auditing by end users.

To differentiate between auditing and traditional verification and validation, I will note several changes. First, auditing implies that someone external (in this case, to the development process) is performing the action; the end user is an ideal motivated auditor. Second, the distinction between verification and validation becomes moot; the end user does not care why software does or does not fail. Finally, the goal is different; the end user will not be concerned about the process that produced the artifact. The artifact itself is the only thing of interest. In other words, a piece of software may pass all validation and verification tests, but still fail an audit.

In order to properly audit software, however, we need new tools. These tools will be of use and interest to end users, but will probably enhance the development process. These tools must visualize how software is functioning.

A METAPHOR FOR MACHINES

We already have many visual languages in active use in software engineering. However, most (like UML) are designed to design systems, or, in other words, to explain how the system *will* work. At a much later point, a system is produced from the design, but the system may have little or no fidelity to the design. Also, even if the artifact is largely a product of the design, certain elements (often structural) never make it into the design.

So, what we need is not another design language, nor even an improved design language. Instead, we need a language and supporting tool that will allow an end user to trace aspects of the functioning system. This "auditing" tool might be seen as something like a debugger; it would allow the user to "open the hood" on a running process.

However, this is not a proposal for a visual debugger. The goal of a debugger is tracing, but an end user's perspective on what should be traced will be quite different than the programmer's perspective.

Additionally, the purpose of this tool is not to explain or explore the components (e.g. objects or functions) of the system, although those will be relevant. The purpose is to expose to the user those aspects that they believe are important. The scenario described below will explain one possible use.

A BRIEF SCENARIO: WHERE DID MY MONEY GO?

Jane is an end user involved in developing banking software. She has worked as a bank teller, personal banker, and business banker, and has been asked by the bank to participate in ensuring that the new banking software functions properly.

In order to perform this task, she has been given a new monitoring tool. The tool allows her to identify a variable of interest and follow it through the system. Jane has decided that she wants to see what happens to an amount of cash deposited into a checking account.

Jane begins by opening up the teller interface, and selecting the screen to enter a deposit. She identifies the deposit as cash, and selects the deposit amount using the monitoring tool. She then completes the transaction interaction.

At this point, the tool begins tracking the deposit amount. Because the new banking software is object-oriented, the amount is placed in a new instance of the Deposit class, and this object is presented to Jane in the center of the monitor tool screen. This object will remain at the center of the screen throughout Jane's interaction.

Jane uses the object as a launching point for her investigation. She follows a link from the Deposit object to the Account object, and verifies that the account information is correct. She then decides to watch the process continue.

The tool automatically stops whenever the members (instance variables) for the monitored object change. At one point, an instance of the Transaction class is created and placed in the object. When Jane sees this, she looks inside this object, and selects this as an additional object to monitor.

The tool later notes that the information from the Deposit class has been written to the database. At this point, Jane is concerned, because the transaction information has not been written. She again follows the link to the Account object, and verifies that the balance has been updated to reflect the deposit.

Now Jane knows something is wrong; banking regulations (and best business practices) dictate that a change to a balance cannot be recorded without first recording the transaction that caused it. Jane lets the tool complete, and notes that the transaction information is eventually written to the database, as well, but she still feels it should have been done first.

Jane immediately goes to talk to a developer to discuss this problem. The developer, Ludmilla, looks at the code, and says to Jane, "Oh, that's OK, it's all happening in a transaction." Jane is confused; to her, a 'transaction' is a business process, not a technical process.

Jane explains her confusion, and Ludmilla realizes the mistake. Ludmilla explains the nature and purpose of isolated database transactions, in which all or none of a specified set of database writes are allowed to occur. Jane and Ludmilla use the point of confusion to propose some new terms.

As a result, the group explicitly uses the terms "database transaction" and "financial transaction", and the class Transaction has been renamed FinancialTransaction. Jane also uses this point to send an email to the developers of the monitoring tool to indicate that the tool should note the boundaries (beginnings and endings) of database transactions.

FINAL THOUGHTS

Looking at a traditional debugger, one might conclude that it could be used in the scenario described above. However, the amount of information on the screen, the monitoring and step points, and the programming knowledge needed to use a debugger make this unlikely.

Again, the goal is not to develop a new tool for its own sake. The idea is to develop a means to allow an end user to understand what is happening inside the world of a software application, in order to support a variety of tasks that can be categorized as auditing.

The advantage of using a visual language (and supporting tool) comes from using a new, potentially unbiased means of looking at the auditing problem that is necessarily limited in size.

We cannot immediately turn the reins of software engineering over to the end user, but we can use novel approaches to engage end users in the process at a deeper level. Traditionally, users have been kept at arm's length from the software artifact, but new interventions can bridge that gap.

REFERENCES

1. Beck, K. *Extreme Programming Explained*. Addison-Wesley Professional, 1999.

2. Brooks, F.P., Jr. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, *20* (4). 10-19.

3. Evans, E. *Domain-Driven Design: Tackling Complexity at the Heart of Software*. Addison-Wesley Professional, 2003.