Stevens Lecture, presented at the IEEE"s 12th (WCRE) and IFIP"s 5th Working Conference on Software Architecture (WICSA), pp.11-14.

Sparking Research Ideas from the Friction Between Doctrine and Reality

Stevens Award Lecture

Mary Shaw
Carnegie Mellon University
mary.shaw@cs.cmu.edu

Abstract

Good research ideas often arise from critical observation of inconsistencies between researchers' assumptions about software development and practical reality. This dissonance creates a kind of friction that can spark research ideas. This is the text for the Stevens Award Lecture on November 8, 2005. The Stevens Award was created to recognize outstanding contributions to the literature or practice of methods for software and systems development.

1. Introduction

First, I'd like to thank the Reengineering Forum for recognizing me with the Stevens Award. This award is particularly significant to me because it is sponsored by an industry association, and it recognizes impact on practice. Since most of my work is basic research, several steps before reduction to practice, it's rewarding to see that my ideas remain recognizable all the way into application. In this research I have enjoyed a long and productive collaboration with David Garlan and briefer collaborations with a number of other colleagues and students, many of whom are here tonight. My work is inseparable from a large body of software architecture work at Carnegie Mellon, both in the School of Computer Science and the Software Engineering Institute. So it's fortunate that we're in Pittsburgh so that many of these colleagues can share in this event. It is especially fitting that we will shortly finish up this celebration in the Carnegie Museum's Hall of Architecture.

I have been working on software architecture for between 15 and 20 years. In the usual model of technology maturation this is about the expected time from basic research to popularization. Paul Clements and I have been looking at that timeline recently, as Paul mentioned in the panel today, and we think software architecture is right on schedule.

Tonight I'd like to reflect on the way this work got started. Actually, I'd like to tell you a little about where many of my research ideas have come from, including some of the ideas that have actually panned out.

2. Friction between doctrine and reality

I have been an academic researcher for quite a while now, and over the years I've noticed that good research ideas often appear if you pay attention to a certain type of dissonance between things you're saying and what actually happens in the real world. Naturally, research should aim to change what goes on in the world, but we should ground our work in reality. Sometimes, though, we tell ourselves things about how the world works that don't match the world very well, things like "if a program isn't correct, nothing else matters", or "a component should have a complete formal specification that says what the component must do". When we say these things, we should feel a certain sort of friction, a little voice saying "how many of the programs you use every day are actually correct?" or "when did you last see a genuinely complete formal specification, or an informal one for that matter?". If you listen to that little voice scratching away at your assumptions, insisting that you're saying things that don't match up with reality, you can sometimes spark little epiphanies that open new avenue of research.

To make this concrete, I'm going to walk you through a few of the "aha"s that have changed my views on good research problems. So here are some things I once believed, and why they might have been good ideas at the time, and why they became no longer good ideas, or at least needed to be more nuanced. I hope that some of these are no longer surprising, because they are two or three decades old. Time has given us perspective to reflect on how the friction in these examples was able to strike sparks.

3. The assignment axiom

The assignment axiom says, in effect, that if you assign a value to a variable, say *x*, the value of that *x* won't change until you make another explicit assignment to *x*, by that name and no other. The assignment axiom prohibits changing a variable's value through more than one name, for example through parameter aliasing or global variables. It is a cornerstone of program verification. It was originally formulated to support proofs, and it had the effects of improving the understandability of programs, reducing abuse of space-sharing features like Fortran common, and reducing the likelihood of a class of errors related to name conflicts in nested scopes.

I subscribed to the assignment axiom for a long time, but examples kept interfering with my faith. For example, in-place heapsort treats the same block of storage as both a tree and a vector, and the careful choreography of the algorithm makes this safe. Likewise, some (but not all) matrix operations can use one of the inputs as the location to construct the output. As a second example, communication of processes through a shared value is a legitimate and controllable form of unsynchronized communication. Similarly, property sheets in Visual Basic provide configuration parameters through global variables.

So at some point I renounced my allegiance to the assignment axiom in favor of a more nuanced position. For the most part, I still regard aliasing as dangerous. But I now understand that with the proper restrictions and protocols, data can be shared between processes or procedures with good effect. This resolved the friction between reasoning about variables and sharing them between processes, and it sparked some of the ideas that led to recognizing shared data as an architectural connector.

4. Functional correctness as the prime objective

I used to chime right in with people who said, "If the program isn't correct, nothing else matters". We meant, of course, functional correctness – that the program matched its pre- and post-conditions. This is a fine sentiment, for it's hard to say that anything less is acceptable. This position has a certain purity, it leads to clean elegant analysis, and it keeps you out of messy discussions about how much less-than-perfect quality is acceptable. We later extended our position to say that other properties, such as performance, reliability, and usability also matter, but we stuck to the principle that correctness matters most.

However, my little voice of friction kept pointing out that most of the software we use every day isn't actually correct, but it makes up the bulk of software running in the world. The problem isn't that software developers don't care about functionality – they do care. But they also care about cost, time to market, and a variety of other quality properties. The cost of making the software perfectly correct is extremely high, if we can even succeed in making it correct. Further, most everyday software doesn't cause catastrophes if it fails, and actual humans monitor it behavior to recover when it does fail.

So I have set aside the quest for perfect software. Certainly there is some software that must be as nearly perfect as we can possibly make it. But there is a vastly larger pool of software in everyday use that must be "good enough" for the task at hand. Its value to its user lies in a balance between the benefit of the software and its cost. For some critical software we can make the case that the benefit justifies quite high costs, but for much everyday software, "good enough" is good enough.

5. The abstract data type as the one and only architecture

In the 1970s, a substantial part of the programming languages community worked on languages for abstract data types. These captured Dave Parnas' ideas about information hiding, and some of them provided for formal verification of the abstract data types in a form proposed by Tony Hoare. They are one of several important precursors of objects.

Abstract data types provide a good way to structure many problems, especially problems that have physical models and problems where defining data structures and their associated algorithms is an appropriate way to organize a software system. Many of the abstract data type researchers of the 1970s claimed that this kind of organization was good for all problems, a claim that persisted into object-oriented organizations.

Unfortunately, I kept running into examples where abstract data types and procedure calls don't really match the application, such as streaming data and other problems for which unix filters provide a good base. I also kept finding problems that could be fit into the abstract data type mold, but doing so wasn't very useful, such as very loosely-coupled reactive systems such as publish-subscribe or event notification systems. One such example was the text editor. Recall that this was happening in the 1970s, before GUIs. So you could define an editor by saying, “Take an abstract data type, ‘text file’. Add a current edit cursor, a remembered cursor (to define spans of text), and some functions that correspond to commands such as copy, insert, delete, move cursor, save, and so on.” The definitions of the functions are straightforward, so this defines a text editor typical of that period. However, it completely misses the point. The design and definition of a text editor is really about the syntax of the commands and how they translate to the basic operations, not about the underlying operations.

The editor example was the direct trigger that led me to start investigating software organizations other than abstract data types. In the early 1980s I studied the structure of user interfaces, and in the late 1980s I took the plunge and started looking for generalizations about architectures of systems – which, incidentally, I tackled much as Grady Booch described yesterday, by taking a big pile of published system descriptions and classifying the architectures they described (by whatever name). I noticed that real developers had a set of shared ideas about system structure and an informal vocabulary but no systematic way to describe their ideas. I resolved the friction between “abstract data types solve all problems” and the profusion of other organizations by trying to codify the other organizations. This was the genesis of architectural styles, now better known as patterns. In the process, I learned not only that people conceive of their systems in abstract terms, but also that they conceive of the interactions between components in abstract terms, like messages, events, shared values, data streams, sessions, and so on. Of course, all of these are implemented in conventional programming languages, with procedure calls, so the paper that made the case for abstract connectors was called “procedure calls are the assembly language of software interconnection”.

6. Complete, formal component specifications as the definitive criterion for correctness

Conventional doctrine holds that specifications are sufficient, complete, static, and homogeneous. This provides a single point of definition, cleanly separates the responsibilities of the implementer and the user, provides a criterion for success, and lends itself to analysis. When specifications are also formal as well as precise, they encourage precision and provide the ability to reason about the component.

For system-level specifications, especially for software architectures and their components, conventional doctrine often fails to hold. The problem isn’t that they’re formal, and the problem is certainly not that they’re precise. The problem is the assumption that any specification of a nontrivial component can be complete. I looked at a product specification and found very little about functionality but a great deal about platform requirements, data sizes, and the like. In fact, people can depend on an unbounded set of properties, and the cost of collecting information for a long list of properties is huge. The problem of unbounded specifications can arise when properties other than functionality are critical, when not all properties of interest can be identified in advance, or when the cost of creating specifications is significant. That is, the conventional doctrine often fails for practical software components. David Garlan and his students wrote a lovely paper about their attempt to reuse – in a single system – three different reusable off-the-shelf software components. They found that even good-faith efforts to make software reusable fail to recognize all the assumptions a user might make about a component. Further, formal systems often don’t express extra-functional properties very well.

I learned that specifications for real software must be incremental, extensible, and heterogeneous. To support such specifications, our notations and tools must be able to extend and manipulate structured specifications. This led me to the idea of *credentials*, or documentation of the properties that have actually been established for a component. This is a property-list form of specification that supports evolving heterogeneous specifications, including metainformation about the level of confidence in the values.

I also learned that different users expect different things of a given component. For example, you and I might plan to use the same subroutine library, but you might care only about trig functions and I might care only about string handling. If the trig functions were inaccurate, I might be perfectly happy and you would find the library useless. Over the past few years my students have introduced me to many situations where different users have different expectations for accuracy, timeliness, reliability, and other properties. Sometimes these expectations are more relaxed than the specification (or whatever the specification would be if it existed), sometimes more stringent. So I now believe that a specification that considers only the component cannot serve the needs of a diverse user community, and I have been investigating user-centered specifications.

7. These are not isolated examples

The list could go on. I have learned from students, collaborators, practitioners in this and other disciplines, and from other experiences that …

- Program components and databases are only two types of resource for system design. Our systems can also incorporate data streams, sensors and actuators in the real world, message formats, systems of documents derived from a master source, and many other kinds of resources.
- The closed-shop form of software development, where all the elements are under the manager's control, does not solve all problems on modern distributed networks. We must also learn how to handle coalitions of distributed resources under independent management. These resources may be changed at will by their proprietors, and users of the resources must be prepared to respond to unexpected changes.
- Most of our software development methods make inadequate provision for a common practice of engineering design: sketching candidate designs, evaluating how they will perform in practice, and choosing a few to refine further. This engineering practice recognizes both the value of getting good information early in design and the cost of acquiring information. An obstacle for software designers is the lack of a suite of compatible predictors that will give good information about implementations of the candidate designs.

8. Current opportunities

You'll notice that many of these epiphanies arose not from researchers' deep misconception about software, but from a simplistic or absolutist interpretation. So a general lesson to take away is that nuanced interpretations often give you more opportunity or more insight than blanket doctrine, and that noticing the friction between what you say and what you do can spark this deeper understanding. Whenever you start to say "always", ask whether you should be restricting the scope of that claim.

Some of the ideas I've mentioned have borne fruit already. Others are ripening and are some of the things shaping my current work. These are

- Good-enough quality for everyday purposes
- Credentials as an alternative to complete specifications
- Coalitions of independently managed resources
- Value – that is, benefit net of cost -- as a design criterion
- Predicting value from design

Ideas don't want to be owned. I'm working on these ideas, but they are not mine exclusively. So if you are moved to work on them, come on in, the water's fine.

9. Footnotes

Use footnotes sparingly (or not at all) and place them at the bottom of the column on the page on which they are referenced. Use Times 8-point type, single-spaced. To help your readers, avoid using footnotes altogether and include necessary peripheral observations in the text (within parentheses, if you prefer, as in this sentence).

10. References

1. David Garlan, Robert Allen, and John Ockerbloom. Architectural Mismatch, or why it's hard to build systems out of existing parts. *Proc 17th Int'l Conf on Software Engineering (ICSE-17),* April 1995.
2. C.A.R Hoare. Proof of correctness of data representations. *Acta Informatica*, vol 1, no 4, 1972, pp. 271-281.
3. David L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm ACM* vol 15 no 12, Dec 1972, pp. 1053-1058.
4. Chris Scaffidi, Ashish Arora, Shawn A. Butler, and Mary Shaw. A Value-Based Approach to Predicting System Properties from Design, *Seventh International Workshop on Economics-Driven Software Engineering Research,* May 2005.
5. Mary Shaw. Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. *Proc. Workshop on Studies of Software Design*, LNCS, Springer-Verlag 1994.
6. Mary Shaw. Architectural Issues in Software Reuse: It's Not Just the Functionality, It's the Packaging, *Proc. Symposium on Software Reuse (SSR '95)*, affiliated with ICSE '95.
7. Mary Shaw. Truth vs knowledge: The difference between what a component does and what we know it does, *Proc. 8th Int'l Workshop on Software Specification and Design*, 1996, pp. 181-185.
8. Mary Shaw. Architectural requirements for computing with coalitions of resources, *Proc.First Working IFIP Conference on Software Architecture,* 1999.
9. Mary Shaw. "The Coming-of-Age of Software Architecture Research". *Proc 23rd Int'l Conference on Software Engineering*, 2001, pp. 656-664a.
10. Mary Shaw. Everyday dependability for everyday needs, *Supplemental Proc 13th Int'l Symposium on Software Reliability Engineering*, 2002, pp. 7-11.
11. N. Wirth, Program development by stepwise refinement, *Comm ACM*, v.14 n.4, p.221-227, April 1971.