FORMAL VERSUS AGILE: AN OXYMORON


Agile methods and formal methods have little in common. Or to put it simply: One size fits nobody, and agile fits very few.

Agile is good for very small projects and for throwaway code supporting research/experimentation. Formal methods should be used for any project that is nontrivial. The potential for useful overlap is miniscule.

The only place that agile fits, with a formal method, is at the front end of the systems architecture definition, where alternatives need to be explored. No amount of interchange will make people with real problems use agile methods except to support research about alternatives. And these days there is little that cannot be designed from real requirements without research that agile could actually be a help with. The exception may be to do drivers and test cases to evaluate hardware capabilities.

Nobody, at least nobody who is sane and rational, is going to give software folks a bunch of money just to see “what they can come up with.” That is not to say that there are not PHBs who can get snookered by technobabble, but with the current economy, their numbers are being thinned down.

In the real world, real (functional) requirements are required, not optional. They are fixed; they do not change rapidly, if at all. They describe the essence of a solution to a problem. The best alternative solution will be further specified by derived requirements, which are subject to occasional change and constrained by nonfunctional requirements such as size, weight, power, security, reliability, and so on.

It is unclear what problems change so fast that agile would be useful. “Requirements” that change are a sign that the real problem has not been defined and that a systems architecture has been bypassed to start design and development without knowing the framework that the “solution” should fit in.

Management has to work to a budget. They need products that are guaranteed to work and solve their problem. And they need them on time. Agile just does not do that in any but the simplest cases.

William Adams
williamadams@ieee.org

The authors respond:

We thank William Adams for his comments. Unfortunately, his viewpoint seems to reflect the thinking of decades ago and is no longer appropriate. These days, agile is used very successfully on many large-scale projects and certainly not only for “very small projects and for throwaway code.” For example, at the recent Agile 2008 Conference, Marcus Evans reported on the use of agile within the British Broadcasting Corporation (BBC) to develop the BBC iPlayer. The BBC iPlayer project has been described as a “project of the same importance as moving from B&W to color TV.”


Regarding the comment that “Formal methods should be used for any project that is nontrivial,” we are proponents of formal methods, but would never agree with this statement. Formal methods have their place, as do agile and other development paradigms in the modern world of software engineering.

In our article, we set out to get colleagues thinking about the big picture regarding practical software engineering in the 21st century. We wanted to challenge stereotypical, old-fashioned thinking. We are happy that we have achieved that aim.

SAAS LIMITATIONS

In “The Web as the Ubiquitous Computer” (Web Technologies, V.S. Pendyala and S.S.Y. Shim, Sept. 2009, pp. 90-92), the authors provide information about software as a service (SaaS). Technological trends are clearly going in the direction of having applications online. Zero maintenance results in a lower total cost of ownership; hosted offerings are designed and finely tuned to scale seamlessly for large numbers of simultaneous users; and upgrades are made frequently and, for the customer, effortlessly.

However, there are several significant limitations that users must bear in mind before they select a hosted application. The sad truth is that computers can fail, resulting in service disruption, which has the potential to be catastrophic. Users are not only subject to network outages, they also must always be tethered to the Internet.

Sensitive data also may be vulnerable when hosted on someone else’s server. In this case, users can’t modify the software and they can’t upgrade it. They are forced to accept changes that they might not even be aware of.
They lose control over what version of the software they are using. Although keeping up to date is one of the selling points for SaaS, in fact, too many new software versions may turn out to be problematic.

With SaaS, users keep paying. They can’t own a license and use it freely. Whenever the SaaS provider changes its rates, users must pay the increased fee or risk losing access to their information.

The cost savings with SaaS is a huge benefit. However, it is important to realize that many of these cost savings could just be upfront costs associated with SaaS implementation. In the long run, it could be more expensive to maintain SaaS due to subscription costs.

If software systems are absolutely critical to a company’s operation, a much better choice is to invest in a more reliable and comprehensive solution to support making guarantees to customers.

Hong-Lok Li
lihl@ams.ubc.ca

The authors respond:

Points of failure cannot be avoided in any setup, including local environments. As described in the column, cloud environments tend to have better reliability and resilience intrinsic to large corporate setups. As we also stated, the Internet is now a household commodity and abundantly available. This is in fact a fundamental premise for the paradigm shift.

Vulnerability and losing control of data were covered in the column. It is not clear how that point applies to versioning, however. Irrespective of the environment, developers should always be in control of versioning their software. If the comment refers to versioning of the software on the leased servers, that is the case even with local servers. The users are bound by what the server vendor provides, whether deployed on a cloud or in the local environment. Vendors can discontinue support to older versions or include features in the new versions that the user may not like. In view of the economies of scale that the cloud provides, there is a greater likelihood of accessing better features.

This point about potential problems with the fee structure is covered in the “Challenges” section of the column. As mentioned there, users “could feel trapped and helpless when providers change their terms of service or operational methods after some time.” Gains from economies of scale inherent in the cloud environment will help keep the costs down.

The new model is more likely to provide a reliable and comprehensive solution and make guarantees to customers than a homegrown environment. The investment and interest in this new model of computation itself is ample proof of this.

DEFINING COMPUTER ENGINEERING

Regarding “Defining Engineering” (Letters, Sept. 2009, p. 6-7): First, I contend that computer software development and programming has very little to do with mathematics. Computer software is a language solution to an automated world. A handyman with good communication skills will likely be as good at designing and developing software as an engineer who has memorized all the rules inside the box.

Many successful software developers have only modest ability in math—myself included—and so couldn’t take a computer science degree in the average university because of the odious math prerequisites. So one of the most creative fields available is blocking many creative people because of the misperception that it requires math.

Second, when programmers are conceived of as technicians the software architecture and design become brain-numbing bureaucratic functions; just produce the ambiguous diagrams and documents, argue about the document structure, get it all signed, and you’re done. Oh, and ship it offshore for programming—it’s already late.

Maybe the reason engineering can’t come to grips with programming is that programming requires craft and abstraction as well as science, and that knowing all the minutiae of the theory may have little to do with effective practice. Too often, an “engineered” software application has many impressive features, but is maddeningly “unintuitive” to use. Doesn’t engineering do “intuitive”? Even an airliner’s complex instrument panel is designed for optimal usability by the user (the pilot). This doesn’t just relate to screens and buttons—most levels of computer systems need to be understandable. Humans need to easily work on them, upgrade them, and interface with them without making errors of misunderstanding that cost time, money, and safety.

My education was in music and language. I have a BA, and I don’t do math—but I have been a successful programmer (and now software architect) since 1972. My own programming experience includes data communications, finance, compiler development, psychomotor testing, animation, graphics, the Web, databases, diagnostic systems, user interfaces, and much else. I have software patents granted and pending, and I authored a successful college textbook on computer programming.

Harry Gilbert
harrymgilbert@yahoo.com

We welcome your letters. Send them to letters@computer.org. Letters are subject to editing for style, clarity, and length.