Int J Softw Tools Technol Transfer (2003) 4: 261–265 / Digital Object Identifier (DOI) 10.1007/s10009-002-0105-2

The opinion corner

Platitudes and attitudes

Jeannette M. Wing

School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA; E-mail: wing@cs.cmu.edu
http://www.cs.cmu.edu/~wing/

Published online: 10 April 2003 – © Springer-Verlag 2003

Abstract. In October 2002 I attended the Ninth Monterey Software Engineering workshop held in Venice, Italy. This year's theme was titled "Radical Innovations of Software and Systems Engineering in the Future." In preparing my talk for the workshop, I thought hard about what I could possibly say on this topic that would not sound stupid. I certainly thought it would be awfully presumptious of me to predict *how* people will or should be developing software in the future. More easily, I could imagine *what* the systems of tomorrow will look like and *who* will be developing them, though anything I would say would sound like **platitudes**. I could also state some strong opinions about what matters and what doesn't in the process of software development. Stating such **attitudes** would at least provoke some discussion. Hence, what follows captures some of what I said at the workshop[1].

Keywords: Future – Computing systems – Complexity – Diversity – Science

1 Platitudes: what and who of tomorrow

Imagine the computing systems of tomorrow. What will they look like? Who will build them?

1.1 What

Systems of the future will be more complex than they are today. Users will expect greater capability without compromise to performance; at the same time users will demand increased reliability, security, customizability, etc. Systems will be even more heterogeneous, where diversity will be measured along many dimensions. Some systems will be stand-alone; others, highly interconnected. Some will be autonomous; others tightly controlled. Systems will be self-probing, self-monitoring, self-adapting, and self-healing.

One major source of a system's complexity arises from the constraints imposed by the system's environment. These external constraints may arise naturally or by fiat.

External constraints due to nature are unpredictable, or at best stochastic, if modeled mathematically. Consider an autonomous vehicle sent to Mars where the system's environment is completely unknown. Or, consider a computer that is under attack by an intruder deliberately operating the system outside its design assumptions. Natural environmental constraints include temperature, altitude, wind, air pressure, and distance below the planet's surface. Natural disasters include fires, earthquakes, and hurricanes. These events are difficult to monitor; their effects are difficult to control. Perhaps at best we can only hope to contain their damage once they occur. Humans also are part of a system's environment, and their behavior is also a source of inherent unpredictability. Noise in the environment further complicates knowing what the correct system response should be: has the sensor failed, or in the next cycle will it start sending valid signals again? Is this aberrant behavior part of an attack pattern or not?

External constraints by fiat include methods adopted by community practice (e.g., corporate culture), standards imposed by professional organizations (e.g., IEEE, IETF), and policies imposed by law (e.g., US export restrictions).

Here are ten other dimensions to consider when thinking about the increase in complexity of a system.

[1] Sections 1 and 2 are derived almost verbatim from my abstract published in the workshop proceedings: "Radical Innovations of Software and Systems Engineering in the Future: Workshop Proceedings," Martin Wirsing and Simonetta Balsamo and Alexander Knapp, editors, CS-2002-10, Universita Ca Foscari di Venezia, Dipartmento di Informatica, September 2002.

- *Functionality.* Systems will be built with more capability, more features, and more options.
- *Size and configuration.* Some systems will be small and nimble. Some will be large and monolithic. Many – small and large – will be connected together, loosely or closely. They will have to interoperate, synchronously or asynchronously.
- *Lifetime.* Some systems will be throw-away. Others will operate 24×7. More and larger datasets will be expected to persist, perhaps in outdated formats.
- *Openness.* Some systems will be self-contained or embedded. Some will be open and exposed.
- *Physical look.* Some systems will be built out of nanotechnology, maybe even biochips. Computer architects will use new types of materials to build new computing fabric. Computing devices will come in different shapes, sizes, and textures. They will be both pervasive and invisible.
- *Properties other than performance and correctness.* Demands for performance (space, time, weight, power) will continue, as will expectations that systems "do the right thing" (or in reality, that systems are "good enough"). Increased demands, however, for fault-tolerance, safety, security, dependability, verifiability, maintainability, etc. will force system designers to rethink the traditional tradeoffs between performance, correctness, and all these other -ilities.
- *Specificity.* Systems will be more custom-tailored. Systems will be personalized, not just to individuals, but to an individual's different persona. Educators are now envisioning that students graduate with their own massive, permanent, searchable and updatable, multi-media recording of all their individual campus experiences, inside and outside the classroom.
  Systems will be domain-specific; what works for automobiles will be different for hospitals, and different yet again for supermarkets, and so on.
- *Connectivity.* We can expect all modes of connectivity: disconnected operation, intermittent connectivity, full connectivity; ad hoc and mobile networks; point-to-point, broadcast, multicast; dynamic and static routing; multi-media physical and virtual links; and wide ranges in latency, bandwidth, frequency.
- *Heterogeneity and interoperability.* We can expect diversity in devices, networks, standards, operating systems, platforms, languages, software applications.
- *Expectations of the end user.* The population of computing users will continue to grow with each generation expecting more than the previous. The infusion of computing technology, transparent or intrusive, into modern society will continue.

1.2 Who

Systems of the future will be developed by teams of people. These teams will be cross-cultural, cross-disciplinary, cross time lines, and involve different skill sets.

Not all team members will or need be savvy with computers. There will be artists, astrophysicists, linguists, psychologists, policy makers, social scientists, statisticians, etc. We will not try to turn computer scientists into biologists or biologists into computer scientists; rather, we will draw from their different expertises in creating a computational biology team.

2 Platitudes: toward a scientific or engineering discipline

While we could build the systems of tomorrow using today's methods, can we do better? The short answer is that there is no silver bullet. However, let that not be an excuse for taking an unprincipled approach to software engineering.

2.1 Mathematical models and theories

What we need first and foremost, as for any scientific discipline, are mathematical models and theories. We have many already. For the foreseeable future, we will continue to have many and more. No one model, one theory, or one method will enable us to build the diverse range of systems of tomorrow. We will need to rely on many different models, theories, and methods. The first challenge is to explore different models to address new domain-specific problems or new technology-driven questions. We should not be afraid to invent new mathematical theories, if no existing one suffices to explain the artifacts we want to build. For example, hybrid systems allow us to talk about continuous dynamics and discrete mode changes, all within the single context of state transition systems. A second challenge is to figure out how to integrate the different models in a semantically meaningful way. We should expect to rely on the mathematical principles of abstraction, composition, and hierarchy to define the results of these integrated models.

2.2 Experimental and empirical methods

We must not only develop theories, but we must also conduct experiments, following sound and scientific methods. These methods must allow us to set up controlled experiments, build prototypes, take measurements, and validate theoretical models. Other scientists must be able to replicate our experimental results.

Experimentation and measurement allow us to improve the quality of our products. Quality is hard to quantify. We can count the number of lines of code, the number of bugs found, or the number of test cases passed. Are these the appropriate metrics? How do they relate to quality of the end product? We should think of other measures and of ways to instrument our code and designs such that we can determine improvement in software quality. In principle, we can formally verify that code meets a given specification, but we are still far from making this a practical method for determining quality; moreover, we will always be limited by what the specification captures.

The laboratories in which we conduct our experiments will undoubtedly rely on automated tools, which we ourselves will have to build initially. These tools should allow us to experiment on or to simulate real systems: large-scale testbeds, legacy code, and working prototypes. These tools should facilitate analysis of our systems, giving us predictive power that complements our theories. These tools should allow us to measure the quality of the artifacts we produce (more on tools below).

3 Attitudes: tools, processes, people, analogies

While mathematical and experimental methods are necessary, we know they are not enough in addressing the longstanding challenge: how can we *better* develop *better* software?

While I dare not predict *how*, I will state some observations and opinions about what matters and what doesn't. Some things that matter go beyond math, science, and engineering; they are non-technical issues, for example, having to do with processes and people.

3.1 Tools are essential

For readers of this journal, it is a given that tools are essential. But what kinds of tools?

In practice, 80% is good enough. A tool that helps a software developer find bugs quickly is more likely to be used than any full-fledged verification tool. A tool that finds one "serious" bug is better than one that finds many less "serious" ones. The tool can be incomplete – missing cases are ok. The tool can be approximate – false positives are ok.

There is a lot of "low hanging fruit" waiting to be picked using today's tools such as static analyzers, model checkers, and theorem provers. We should use them all! Finding 60 bugs in 100 device drivers[2] is better than doing nothing; besides, since these drivers run on millions of workstations and laptops around the world, the effect of fixing any of the serious bugs found can have a dramatic impact on a huge customer base.

Use of these tools must be a seamless part of the software development process. Software developers are unlikely to use a brand new tool unless they see value added in doing so; however, if a tool they already use, e.g., a compiler or linker, is enhanced slightly by doing some additional checks, then they would get these checks almost "for free."

[2] Ball T, Lichtenberg J, Rajamani S (2002) (private communication)

3.2 Process is important

While tools are essential, they are not enough. They are aids to help automate parts of the software development process. While the idea that process is important is not new, the idea of evaluating the *quality* of the process is.

As argued earlier, measuring quality of product is difficult. Measuring quality of process is even harder. Today, many software development processes rely on checking off boxes on lists, filling out forms, abiding by certification procedures, or following best practices. Many companies follow the only process they know that produces results: code, code, code. How do we determine that one process is better than another? Surely not by comparing the numbers of boxes checked off or the numbers of forms filled out!

Worse, we have no scientific basis for correlating quality of product to quality of process. Ideally, the better the process, the better the resulting product. This correlation, after all, underlies the Software Engineering Institute's Capability Maturity Model. However, some high-quality software products are produced by small teams that follow no strict development process; at the same time, adhering to a rigid process does not necessarily mean a better product. Processes that involve checklists and templates do not speak directly to the quality of the product – the end system. We need to understand better the relationship between process and product: what improvement in process translates directly into an improvement in product?

3.3 People are part of the process

There are three important ingredients in software development: product, process, and people. It is perhaps uncomfortable for scientists to talk about the "people factor" since unlike products and processes, people are even less controllable or measurable. If we are to improve the quality of software, however, we cannot completely ignore the people involved.

We do know that high-quality people can produce high-quality software, perhaps even independent of the process they follow. Since talent is limited, however, we need to define processes that can best use the people at hand – to exploit their individual and collective strengths and talents.

The reward system also has to be appropriate for the end goal. If we hire only people who are good programmers, then all we have are good programmers. We could end up with an implementation of a system whose architecture/design does not match the customer's requirements, resulting in a working, but useless system. If we promote people because they produce lots of lines of code, then all we have are lots of lines of code. The code may be buggy or hard to read, modify, maintain, etc.

In training future software professionals we must consider the science and the engineering ingenuity needed to build the systems of tomorrow. Computer scientists already agree on the importance of discrete mathematics and logic in undergraduate curricula. We now see a renewed demand for multivariate calculus, linear algebra, and probability and statistics. Back to basics!

Finally, given that systems of tomorrow will be built by teams of people with varying backgrounds, not only are mathematical, programming, and problem solving skills important, but so are good interpersonal communication skills. Being able to write and speak clearly can be critical to the success of a project.

3.4 Do not be seduced by inaccurate analogies

It is irresistible for the software engineering community to draw analogies when looking to our brethren engineering fields such as electrical engineering, civil engineering, or design engineering. Software, however, is not just like a circuit, a bridge, or a piece of furniture. The fundamental differences between our artifacts and theirs are that we can *instantaneously change* our systems and we can *instantaneously copy* our systems. Indeed, it is typical for a software developer to copy someone else's (or perhaps his own previously written) code because it does almost the right thing, modify it slightly, and install the result. At best, there is extraneous functionality or code; at worst, old bugs perpetuate.

In addition, while software engineering researchers worry a lot about maintaining code that evolves, in looking at systems of tomorrow, we need also to worry about code that is copied and distributed widely. This ability to spread code (and data) easily is the source of security and privacy problems such as viruses and spam. They may be only annoyances now, but they could be more harmful in the future.

One thing we can learn from our engineering brethren is the idea of "overengineering" software. In particular, we would like to be able to overengineer software without sacrifice to performance, e.g,. in using redundant runtime checks, or without sacrifice to usability, e.g., providing multiple ways to accomplish the same end-user task.

The speed at which we can change and copy our designs and implementations also suggests that the "software factory" analog to manufacturing processes and the "design-by-handbook" analog to engineering processes will break down unless undue discipline and restrictions are placed on the software engineer. The trend of open source software and even the new interest in extreme programming are examples of how these other engineering disciplines may even miss the mark.

It is also tempting to look to other sciences such as ecology, entomology, and immunology for understanding the behavior of complex systems; however, while they can serve as inspirations for ideas and alluring metaphors, it's unlikely that their models directly apply to software and software engineering.

We are more likely to be able to borrow and adapt techniques from disciplines such as behavioral and organizational science, economics, and social and decision sciences in order to model and analyze software development processes – because of the "people factor." Understanding human behavior in general and how people make value judgments in particular would be useful.

4 Attitudes: research agenda

What follows are some ideas for possible short-term research projects and longer-term research goals.

4.1 Short-term research

- **Featherweight checkers.** The next step beyond "lightweight formal methods" [3] is featherweight checkers. These kinds of tools must fit unintrusively into a software developer's life. One approach is to include featherweight checks into existing tools, such as a type checker or debugger, that the developer uses already. Another approach is to build a tool that performs an extremely specific, but critical check, e.g., ensuring that procedure X is never called from procedure Y because it would open up a security vulnerability.
- **Programming language support.** Types give us a kind of verifiable code, albeit type checkers verify only simple properties of code. In support of writing code for verifiability, we should continue to push the boundary between what we can express in a type and what we can check automatically.
  Much of the tedium of producing a large software system is doing regular, e.g., daily, "builds." It would be nice to incorporate mechanisms directly in the programming language that would help in the monitoring and management of the "build" process as part of the design-code-test-debug cycle.
- **Certification processes.** Different organizations require different processes for certifying software. The Federal Aviation Authority uses something different from the Food and Drug Administration. Industry differs from government as well. We should examine the certification processes of today, understand where they fall short and why. We can then extract best practices, spread the word, and codify/systematize them. We should look more closely at the correlation between process and the quality of the end product.
- **Human behavior.** To better understand a given software development process, we should study the role of humans. In such a study we need to involve social, behavioral, organizational, and management scientists. We should look for opportunities to exploit

[3] Jackson D, Wing J (1996) Lightweight formal methods. IEEE Comput 29(4):21–22

any innate human advantage over computers (e.g., vision, speech, pattern matching, tolerating faults, real-time/priority task scheduling).

4.2 Long-term research

- **Evaluating design**[4]. Given that we understand the processes of coding and testing, and that we have some measures for these processes, how can we measure the "goodness" of a design? It would nice to be able to know how to evaluate whether a design is "good" or "better" and to able to predict the effects of a "good" design versus a "bad" one. We could then begin to think about building tools to help measure and evaluate designs, to teach people what makes a design good, and to reward and promote people for being good designers. Without evaluation criteria, preferably quantifiable, software developers and their managers will be reluctant to change, especially if the code-code-code process is the norm.
- **Revisit specification languages.** Specification is in vogue again, but not in phase with current technology or practice. We should revisit specification languages, logics, models, theories, and application-specific domains given the breadth and complexity of tomorrow's systems.
- **Lightweight verification.** We should strive to make verification (static and dynamic analyses) truly lightweight, like compilation.
- **Scale.** We still need to figure out how to scale everything up – through new abstraction and compositional techniques. While we should continue to understand how to build reliable systems out of unreliable components, why not also think about how to build reliable systems out of reliable components?
- **Probabilistic verification.** We should consider radical approaches to verification. One idea is probabilistic verification. Perhaps we can use approximation/randomization algorithms in either the verification process or in implementing verification tools. We can certainly use stochastic models (Markov Decision Processes, Bayesian networks) to help model the complexity of a system's environment. Open for exploration is the different interpretations of what "probabilistically true" or "probabilistically correct" might mean.

[4] By "design" here, I mean anything above the source code level, e.g., at the module level or architectural level.

5 Summary

The complexity and diversity of computing systems will continue to grow. We need to continue to develop the mathematical and scientific foundations so we do not have to put quote marks around the word "engineering" in the term *software engineering*. In the short-term more featherweight tools will help. In the grand scheme of things, however, we need to understand better – with a scientific basis – the roles of process and people as they affect quality of product. Perhaps one day we can look back and see that my attitudes have turned into platitudes.

Acknowledgements. This work is supported in part by the Army Research Office (ARO) under contract no. DAAD19-01-1-0485, the National Aeronautic and Space Administration under contract NCC2-1241, and the National Science Foundation under contract no. CCR-0121547. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the DOD, ARO, NASA, NSF, or the U.S. Government.