

I’m writing in response to Warren Harrison’s “Best Practices: Who Says?” January/February column. As usual, he’s laid a very controversial issue on the table. Determining best practices is like judging what’s good and what’s not. How does one know what’s good and what’s bad? Looks like a straightforward debate, right? Whatever the law deems good is good and whatever the law says is bad is bad. The problem is, there are many things the law says nothing about, such as coffee, salt, and smoking. What about computer games or action movies? Are they good or bad? There are myriad other things that can be good or bad depending on who you ask and the context. Software best practices are no exception.

Software licensing requirements will become a reality with serious impact on the consumer. When that time comes, our profession better be prepared to reason about software malpractice. In their article for the September/October 2002 issue, Thomas Hilburn and Watts Humphrey asked, “Can the software industry solve its own problems in time to prevent a serious public disaster, or must the government solve our problems for us?” They continued with a troublesome warning: “The fact that a governmental body doesn’t know how to solve these problems any better than we do won’t deter politicians. If the problems are serious enough, and the public concerned enough, there will be action—whether it’s effective or not.”

When this becomes a matter of law, peer companies, a panel of renowned researchers, or even SWEBOK won’t be able to establish what best practices are. I doubt a software startup with 10 developers that’s developing a 50,000-lines-of-code product would agree to follow NASA-established standards designed for software with two million lines of code.

Regardless of how they’re defined and who defines them, best practices must be documented, standardized, and incorporated in licensing requirements. They must become standard practices. Licensing bodies can and must enforce them. This would greatly improve consumer protection and increase confidence in software products. As long as it’s acceptable for software products to be provided as is, without warranty of any kind, we can’t expect to use or enforce best practices in a legal setting.

We should be able to borrow ideas from other professions regarding licensing, quality of service, and the definition of malpractice, and I would welcome an entire IEEE Software issue that would discuss this topic.

Magdin Stoica
EngPath
magdin@engpath.com

I just read Warren Harrison’s editorial on best practices and have a few thoughts about the term and its use in the computing community.

Although Warren’s analysis of the Ameri-
can Society for Quality’s statement is thorough and insightful, it might not be that relevant. People wildly overuse the term best practice, almost always without justifying why it’s the best or even what it’s best at doing. But even so, it’s not what the ASQ or the software community thinks is a best practice that matters, it’s what the law thinks is a best practice.

Warren mentioned that his law enforcement colleagues disliked the term best practice because failure to follow one could lead to case dismissal. This fact exposes “best practice” as a misnomer: if you lose because you don’t follow the practice, it’s not really a best practice; it’s a required practice. This will separate the sheep from the goats. If this criminal-law definition has a parallel in civil law, then failure to follow best practices could be considered negligence and have a devastating impact on the industry.

That explains how to identify best practices. If a best practice is a required practice in a legal environment, you’d have to be soft not to perform it. This would imply that everyone in the industry should be performing that practice. There’s no need for comparing organization size, reputation, outlook, and so on. You can determine what’s a best practice by looking at how people develop software. If it isn’t widely implemented, you can argue that it may be a good practice in some instances but doesn’t have the strong, broad positive impact on performance we’d expect of a best practice.

This also has some implications for IEEE Software. You’ve made a good start by restricting the use of “best practice” in submissions to your magazine. An innovative technique should never be labeled a best practice because it hasn’t withstood the test of time; this is why I find the ASQ definition unacceptable. It would help members if someone could determine if a civil-law definition of best practice exists that demands a certain performance level from software organizations. Then developers could avoid terms that might expose them to unwarranted legal action. It would be a nightmare to be sued for negligence and have back issues of Software with “best practice” definitions entered as evidence against the developer.

Dale R. Sinclair
Hedgehog Technologies
dale@hedgtech.com

An effective metaphor

While catching up on my reading, I ran across Alan M. Davis’s column on genotypes and phenotypes as a metaphor for requirements and design (Requirements, July/August 2003). Usually when I see an article on requirements, I try to keep from rolling my eyes and start to wade through. I usually never reach the other bank. But this article was a joy to read. When I was earning my MBA, in a management information systems class, I was fool enough to proclaim that systems should be designed “from the glass back,” meaning that the most important thing was how the system looked and behaved from the user’s point of view. I also advocated the “pizza and beer” simulation, in which everyone relaxed with pizza and beer and played acting using the system in an attempt to uncover hidden requirements. People looked at me like I had two heads, and I chose to remain silent thereafter. Well, almost.

Imagine my surprise when someone who knows what he’s talking about (Alan Davis) actually advocated focusing on the system’s external behavior! Because I have an undergraduate degree in biology, I immediately understood the concept he described, but I would never have thought to apply it to requirements definition—very nice, and very effective in distinguishing the disparate roles of requirements and design.

There’s one difference between genotype/phenotype and requirement/design that you didn’t list that might be important in some cases. An organism’s genotype defines not only how it behaves but also what it is—that is, it contains the information necessary to build the organism. This would be like the system requirement specifying how to build the hardware. It’s a minor point, but I suspect that in most cases the specification usually assumes existing hardware.

Once again, thank you for a very satisfying article.

Dale R. Sinclair
Hedgehog Technologies
dale@hedgtech.com

Alan M. Davis responds:

Your letter convinced me that the software program itself is equivalent to an organism’s genotype. Interesting that we call both code—program code and genetic code. And yes, the program code does contain enough information to “reproduce” the program.

Thanks for the Requirements column by Alan M. Davis (July/August 2003)—I enjoyed it. A couple of thoughts came to mind as I was reading and I thought I’d shoot off my two cents worth.

Near the end, Alan points out that, in systems development, the phenotype comes first. True—but I wonder about the postconstruction phenotype’s relationship to the preconstruction phenotype. The postconstruction phenotype represents the system as it’s finally delivered; it differs from the preconstruction phenotype in at least two ways.

First, it contains characteristics that aren’t represented in the preconstruction phenotype. For example, pressing the buttons in an unspecified sequence
might cause the program to fall over or have an “easter egg” effect (both common in desktop programs). Are these then bugs (not all the time, I guess), or are bugs only features that were specified in the preconstruction phenotype but aren’t delivered accurately in the postconstruction phenotype? Common sense says I know a bug when I see one, but I’m not going to specify every possible abnormal condition up front. Hmm.

Second, the postconstruction phenotype is more like geneticists’ usage in that it can be examined to increasing levels of detail. This eventually blurs the line between phenotype and genotype because our “microscopes” let us look into a system’s genetic makeup.

So what’s the link between the pre- and postconstruction phenotypes? What happens to a requirement once the system has been built and gone live? Does it have an ongoing meaningful existence? No startling observations here, I guess—just some disorganized musings.

Peter Houlihan
Charter Wilson & Associates
p.houlihan@computer.org

Alan M. Davis responds:

You certainly discovered one big difference between biological and software phenotypes, and it’s related to the timing. In the biological case, the phenotype is just an external manifestation of the organism’s true identity—that is, its genotype. In the software case, the phenotype, or the set of requirements, is sort of a “request” for a specific phenotype of the eventual system. So, as you pointed out in your letter, the “as-built requirements” are really more similar to the system’s phenotype than the original preconstruction requirements. Unlike the biological case where the in utero cells transform into a final organism via relatively predictable, relatively uncontrollable processes, the software construction process from conception to full-system status is relatively unpredictable and relatively controllable. Also unlike its biological counterpart, the “requested phenotype” changes throughout the development process—hence the differences between the original “requested” phenotype and the final as-built phenotype.

No need to fear

In his Loyal Opposition column, Bob Glass always takes interesting points of view and challenges “everybody knows that” positions. This month’s column about outsourcing development (January/February 2004) touched on foreign developers leaving backdoors in their software to create a binary 9/11 sometime in the future.

A very responsible and thoughtful argument indeed! Just think about all the non-US data-processing users who are limited to using US-made software (especially operating systems). Here in Germany we have a choice between IBM’s mainframe OS, Microsoft’s Windows, Apple’s Mac OS, and a range of Unix incarnations.

The one German contribution to that portfolio is the general-purpose mainframe operating system BS2000/OSD from Fujitsu Siemens. One would guess that German security-conscious corporations and government institutions would grab that opportunity and put all their mission-critical eggs in that basket. Unfortunately, this isn’t true in all cases, and (pseudo) economical reasons and the “me too” attitude fuel the desire to implement everything on Windows. Using Linux doesn’t disprove this argument because it’s a global rather than domestic OS.

So, dear Americans, don’t worry about what other countries’ software will do to you; instead, jump on the bandwagon and boldly share everybody else’s risks.

Juergen L. Hemm
Delivery Manager Mainframe Systems
T-Systems CDS
juergen.hemm@t-systems.com

Holes in UML

I completely support Stan Jarzabek’s comments in “Will MDD Fulfill Its Promises?” (Letters, January/February 2004). I’d just add a few notes as a software architect using various methodologies and models as well as UML notation in practice.

One well-known problem with UML is that it almost completely ignores both user interface and data models. OMG’s Model Driven Architecture states this explicitly (www.omg.org/mda): “MDA aims to separate business or application logic from underlying platform technology.”

User interface models are a crucial part of any user-oriented software system, including Web and desktop applications. Although they recommend using UML’s class diagrams, sequence diagrams, and so on to describe presentation tier, I rarely hear of anyone actually doing that. Class diagrams are too abstract to describe user interfaces. In many cases, objects located on pages should be considered as different views of business objects, such as parts of the same table that are shown on several pages—a concept that isn’t present in UML.

Another reality that UML ignores is databases. I can’t imagine an enterprise software system that doesn’t use thousands of distributed databases. One simple reason that UML ignores databases is that UML is object-oriented and databases are mostly relational.

So, if we’re talking about describing what a system should do and how it should be done, what kind of software can we expect to build from a model that’s missing descriptions of user interfaces and persistent data storage?

UML’s visual (instead of syntactical)
representations and semantics aren’t self-evident. To properly read and un-
derstand UML models created by soft-
ware architects or designers, develop-
ers will have to understand those visual
elements and semantics exactly, with-
out any ambiguity. (A filled diamond
has a different meaning than an empty
one; an arrow’s head means synchro-
 nous, asynchronous, or another type of
call; the thick border of class’s rectan-
gle differs from the thin one; and so
on.) Or else we’ll end up with a situa-
tion similar to that of formal models of
programming languages—only archi-
tects will be able to create, read, and
maintain MDA models and UML dia-
agrams. For project managers, program-
ers, and testers, architects will have
to produce informal descriptions.
UML provides some dynamic sem-
antics at the level of objects sending
messages to other objects—roughly, syn-
chronous and asynchronous method
calls. It also has control structures such
as loops wrapping those methods’
calls.
To make a truly executable model,
UML will have to provide lower-level
control structures and some standard
data types, such as integers and strings,
to introduce operators on those data
types and formally describe operational
semantics. Further, UML will probably
need some notation of variables to
have executable dynamic semantics.

Given some MDA’s metamodel and
some, say, Java implementation of that
metamodel, can we prove that those
two models will produce the same out-
puts for the same inputs? It seems the
answer is no. If that metamodel
isn’t intended to precisely describe
the system’s behavior, then why make it ex-
cutable, and shouldn’t they specify
the MDA’s scope or limitations (ex-
actly what it’s meant to describe)?

Kirill Fakhrotdinov
Senior Internet Architect

Stephen J. Mellor responds:

In the early days of our industry, pro-
grammers wrote in assembly code, se-
lecting registers in which to place vari-
ables and managing memory explicitly.
If we had magically provided these pro-
grammers with a Smalltalk compiler, they
might have asked, “How does this help
us select registers? How do we allocate
memory?” They might have concluded,
“We don’t want no stinkin’ Smalltalk!”

Old and new programmers are still
writing programs, but the technology to
achieve the goal has changed. When a
new technology is sufficiently different,
you can’t evaluate it in terms of the old
technology. Conversations about the tech-
nology are unsatisfying, too: “How can I
allocate memory in Smalltalk?” “It does it
for you.” “Okay. Where’s the function to
do that? And how do I say which loca-
tions I want allocated?” “Um ….”

Kirill Fakhrotdinov
Senior Internet Architect

Martindale-Hubbell
kfahrut@optonline.net

Stephen J. Mellor responds:

In the early days of our industry, pro-
grammers wrote in assembly code, se-
lecting registers in which to place vari-
ables and managing memory explicitly.
If we had magically provided these pro-
grammers with a Smalltalk compiler, they
might have asked, “How does this help
us select registers? How do we allocate
memory?” They might have concluded,
“We don’t want no stinkin’ Smalltalk!”

Old and new programmers are still
writing programs, but the technology to
achieve the goal has changed. When a
new technology is sufficiently different,
you can’t evaluate it in terms of the old
technology. Conversations about the tech-
nology are unsatisfying, too: “How can I
allocate memory in Smalltalk?” “It does it
for you.” “Okay. Where’s the function to
do that? And how do I say which loca-
tions I want allocated?” “Um ….”
Evaluating a new technology in terms of the old isn’t a good idea. Let’s take just one example. You say, “Another reality that UML ignores is databases. … One simple reason ... is that UML is object-oriented and databases are mostly relational.” In other words, “Another reality that Smalltalk ignores is memory allocation. One simple reason for that ignorance is, Smalltalk assumes infinite memory!” Clearly, some mechanism that allocates memory exists; it’s just done the same way as the old technology. In model-driven development, the object-oriented model is mapped to a relational model using explicit transformation rules, a key technology in model-driven development.

In your final paragraphs, you raise questions about UML’s executability. First, let’s be clear: UML is executable (see Executable UML: A Foundation for Model Driven Architecture, Addison-Wesley, 2002). UML does have lower-level control structures (ConditionalAction, for example) and some standard data types such as integers and strings. UML also lets an action model-compliant action language define operators, either primitive or complex.

That said, two fundamental styles of MDA exist. One style, an Agile MDA, is based around executable models. You can execute each model stand-alone and combine it with others (including a user interface model) to produce a system. The other style, an elaborative MDA, successively transforms models at various abstraction levels, elaborating some by adding code directly. Both styles fit under MDA’s umbrella, although any precise discussion should specify which style is being assumed.

Personally, I’m not a fan of the second style. Although I can see its usefulness as a way to improve programmers’ productivity in using models to describe code, that’s also precisely its problem: It perpetuates the myth that the model should mirror the software’s structure. That’s like saying Smalltalk should have a way to manage registers and allocate memory explicitly.