Having just read your article “Module Assembly” in IEEE Software’s March/April issue, I must object to the increasing complexity of the various “solutions” advocated. While your comments on modularity, polymorphism, dynamic binding, and interfaces are all well taken, the approaches presented beyond these appear as needless complexity. Consider the increasing complexity of each approach, from a user’s viewpoint:

1. Use object-oriented programming: Abstract data type + inheritance + dynamic binding—for example, Company/List/ArrayList. Here multiple implementations of a module interface (such as descendent classes, OpenClosed, Single Choice, Liskov SP, and so on) can all be linked into one executable that selects a specific implementation (such as Abstract Factory) based on user input (such as command-line arguments). This is the simplest solution: just a single executable file given to users to install anywhere in their existing PATH. (On rare occasions, you might need to make extensions without modifying even a single module, in which case you can use Pluggable Factories (C++ Report, Oct. 1999), but you should never use such added complexity gratuitously.)

2. Create shared libraries for each descendent. This leads to “DLL Hell.” Shared libraries get my vote for the worst “innovation” in software since the days of goto, common blocks, variant records, equivalence, and implicit typing. They are the bane of those of us targeting multiple platforms. Even on one platform, users must install not just a single executable file but also a collection of object files (a certain version, a certain location) and then set their RPATH (modify “dot files”), and so on. Needless burdensome. Shared libraries are for the most part a “solution” to a nonproblem—memory is large and cheap to add.

3. Add a module assembler application and an associated configuration file. Now we’re up to 3 + N (shared-object) files to install per host.

4. Make the configuration file XML. Now add the need for an XML parser (such as Expat). Overkill. 4 + N.

5. Make the configuration file a script program itself. Ugh. More logic to understand, test, debug, and so on. Add the interpreter (for example, Python) that the user must locate, download, install, and test on each platform and host. Oh, and hire a programmer fluent in that interpreted language, too. (Or users must become programmers.)

6. Finally, make the assembly process dynamic. Add requirements for a Web site connection, a service locator (remote Web services application, firewall modules), a database (“clearing house” or dumping ground for ever-changing, differently bugged implementations), and so on. Need help from system administrators now.

This exemplifies the trend toward increasing complexity and dynamicism that has a large negative impact on many software qualities: understandability, testability, robustness, ease of use, efficiency, and more.

While applications that warrant such sophisticated dynamic application configuration management might exist, they’re likely few and far between. Nonetheless, legions of programmers inspired by their interest in mastering these techniques will begin employing them on existing and future projects (just to learn how), thus burdening their users unnecessarily. Users would be
better served if programmers heeded Niklaus Wirth’s “Plea for Lean Software” (Computer, Feb. 1995). Respected experts such as yourself could help ameliorate the problem by including more advice on when and when not to employ the sophisticated techniques presented (with compelling examples).

Gratuitous complexity: just say no.

Todd Plessel, CSDP
HPC visualization specialist
plessel@computer.org

PS. If memory serves, I think you wrote an article on the abuse of patterns.

Martin Fowler responds:

You’ve pointed out a clear fault with the article. My early discussion of assembly options carried an assumption that the modules were developed by separate groups and needed to be assembled in a later step. While I did mention that later on, that context should have been stated earlier and more clearly. Furthermore, there should have been some discussion on the pros and cons of allowing implementations to be added to a system at configuration time.

Having said that, I don’t agree that applications that raise these assembly issues are few and far between. Even if all implementations are known, you still need some sort of module assembler to resolve the choice—the difference is that the assembler is also part of the single executable and is written in the same language as everything else. I think the general point, the separation of assembly from use, still holds.

I agree we must avoid unnecessary complexity. Sadly, my own abilities to understand and explain the right level of complexity are far from perfect. Thoughtful conversations like this help improve both dimensions.

MDA Revisited

I’ve just finished reading Bob Glass’s column “On Modeling and Discomfort” (Mar./Apr. 2004), and it’s great to see some sanity being injected into the model-driven architecture hype. I totally agree with you about models and about MDA’s limited applicability—I predict this is another “silver bullet” that will turn out to be lead.

In discussions with MDA zealots, I always ask two questions:

1. How does it help me configure the generic application system I’ve bought from supplier X? In my current work in healthcare systems, virtually all users are configuring systems, not developing them from scratch. I think this is also more broadly true of business systems. I have limited experience in this area, but the people I know are configuring SAP and Oracle systems.

2. How does it help me achieve my required levels of performance, availability, and reliability? The argument for MDA is that it’s good for long-lifetime systems with changing platforms, but in my experience these

Inherited 1MegaSLOC of MegaSLOP?

Our tools help developers understand, document, and maintain impossibly large or complex amounts of source code.

They parse Ada 83, Ada 95, FORTRAN 77, FORTRAN 90, FORTRAN 95, K&R C, ANSI C and C++, Java, Jovial, Pascal, & Assembly source code to reverse engineer, automatically document, calculate code metrics, and help your engineers understand, navigate, and maintain source code that has grown too large for one person (or even a group) to know.

Big projects aren’t a problem. The can parse and later manipulate very large amounts of code. 1,000,000 SLOC projects (and larger) are common among our customers.

We also focus on exceptional customer support based on rapid response from a real engineer, with bug fixes and new features incorporated into weekly builds.

Download and try on your code:
http://www.scitools.com/

All downloads are fully functional, just time limited.
systems’ key requirements are performance and dependability.

I haven’t had a good answer so far.

Ian Sommerville
Professor, Lancaster University
is@comp.lancs.ac.uk

I always get a big kick out of reading Bob Glass’s columns, and this month’s (“On Modeling and Discomfort,” Mar./Apr. 2004) was no exception. You cut right to the heart of the problem with much of what’s called model-driven development. I appreciate your pragmatic approach. I also went out and bought a copy of Eric Evans’s book after reading your article—it was his quote trashing the idea of code generation from UML models that sold me.

Martin Fowler (http://martinfowler.com/bliki/ModelDrivenArchitecture.html) has expressed some thoughts that dovetail nicely with what you presented. Much of the disconnect comes down to how we perceive the models’ purpose. Are they sketches, blueprints, or a programming language? I think they’re most useful as sketches and, on rare occasions, as blueprints. For programming languages, I prefer, well, programming languages—not UML. It sounds like you’re in the same camp.

Fowler also has some related thoughts on directing versus enabling attitudes. I contend that the modeling viewpoints you’re uncomfortable with are toward the directing end of the spectrum: spell out everything because we don’t trust the implementor.

At one organization where I worked, the software manager considered the holy grail of development to be creating UML models that were sufficiently detailed that we could generate all our code from them. In his mind, this meant that design and implementation would always be in sync. My (fruitless) argument was that his approach really just shifts from modeling the design to modeling the implementation. You no longer capture the design, in that you’ve given up the more abstracted views that facilitate understanding of basic relationships, structures, and so on. Another disconnect occurs when talking about what software “production” is. In the mechanical world, you give a machinist a detailed mechanical drawing before he starts cutting metal. In our software realm, is “cutting metal” analogous to writing source code in a programming language or compiling that code into an executable? In my opinion, the source code itself is that detailed implementation spec, but a lot of folks assume source code is the end result—the machined part. This divergence in viewpoint makes a huge difference in what you think you need models to do.

“Hear, heart” for airing the heretical viewpoint that so many of us have secretly thought to ourselves.

Brett R. Holt
bholt@atrcorp.com

READ SAMPLE CHAPTERS TODAY AT: www.awprofessional.com