USE CASE DIAGRAMS
In “Qualifying Use Case Diagram Associations” (A. Dedek and B. Lieberman, June 2006, pp. 23-29), the authors propose the excellent idea of using a domain model to refine use case models.

The unified process also recommends this approach. For example, see Craig Larman’s book, Applying UML and Patterns (Prentice Hall, 2004).

The authors also propose extending UML use case diagrams to show data flows and data storage. Analyzing data flows in this way improves requirements specifications. They note that qualified use case diagrams strongly resemble the older dataflow diagrams (DFDs). Some obscure features in UML 2.0 express data flows. These features are information flows and streams (Object Management Group, http://www.omg.org).

A standard component diagram with information flows provides the same documentation as a DFD. Components have stereotypes for subsystem, process, and entity. UML actors can play the part of data sources and sinks. Entities and information flows should be matched with a domain model. Processes can be refined to any number of use cases since existing processes do not need design.

These diagrams show all the information of a qualified use case diagram or a DFD. They typically have fewer arrows than a DFD and fewer boxes than a qualified use case diagram. It is possible to use activity diagrams with stream connectors, but doing so requires supplying extra details.


Richard J. Botting
rbotting@csusb.edu

The authors respond:
UML 2.0 provides at least nine diagrams that could be used to present the structural (object, component, deployment, and class diagrams), behavioral (use case, sequence, activity, collaboration, and statechart diagrams), groupings (package mechanisms), and annontational (notes symbols) aspects of an evolving design.

Each of these diagrams defines a particular view of the design. For example, while sequence diagrams expose the time ordering of messages between objects, statechart diagrams emphasize the event-centric ordering of each object’s state machine representations.

The standard component diagrams that Botting refers to are very useful in the later stages of software development projects, where the emphasis is development of the interactions between objects that send and receive messages. The stereotypes of the component diagram, such as subsystems, processes, and entities, are meaningful to software developers but probably less useful for end users who help with the development of solution requirements.

We argue that, while all UML diagrams are useful for different phases of the software development process, not all are intended for the requirements development phase. An advantage of the qualified use case diagram is that it enables the business analyst to represent the business problem meaningfully to subject matter experts.

One of the more vexing problems in requirements engineering is representing the business domain data elements at the correct level of abstraction and indicating where to use this information in the business flow. This technique permits a visualization of those elements and the points where they enter and leave a use case definition.

Adenekan Dedeko
adenekan.dedeko@umb.edu
Benjamin Lieberman
blieberman@biologicsoftwareconsulting.com

OUTSOURCING CONSEQUENCES
Reading “The Data Doughnut and the Software Hole” (The Profession, June 2006, pp. 100, 98-99) reminded me of my involvement in the earlier days of the development of the Australian Customs Integrated Cargo System (ICS). As a sort of “super user” with an IT background (computer science), I made recommendations on the data and data integrity issues that were a concern with the extant systems, issues that had arisen because the systems were being used to process data in a way not intended in the original specifications and design.

The ICS was designed to replace a series of disparate and independent but highly functional computer systems that had been developed and implemented from the early 1970s onward. The older systems originally ran on IBM and Unisys hardware but were migrated entirely to Unisys in the late 1980s and early 1990s.

Needless to say, there was a huge amount of intellectual investment in these systems as they embodied Customs’ business rules. Sadly, this investment was largely ignored in creating the ICS. Part of the problem was a lack of documentation, but the other part was that Customs was among the first Commonwealth agencies to embrace outsourcing in the mid-1990s. Thus, it lost nearly all of its IT expertise to EDS, the outsourcer, and the rest of the private sector. It seems apparent that little thought was given to the longer-term implications of outsourcing.

As our outsourcer, EDS was responsible for the early development efforts, but after much wailing and gnashing of teeth, the contract to develop the ICS was eventually given to Computer Associates. During the protracted development process, little maintenance was done on the now-labeled “legacy” systems, which resulted in other problems emerging over time as there was a natural desire to minimize expenditure on the “old” and direct it to the “new.”
I’m convinced that the scenario I’ve described occurs regularly when proponents of the new don’t look to the past first.

Walter Hill
hill@ieee.org

DESKILLING SOFTWARE ENGINEERING

The software upgrade incident that David Grier mentions in “Across the Great Divide” (In Our Time, July 2006, pp. 8-10) reminded me of another article, “The Plot to Deskil Software Engineering” (Comm. ACM, Nov. 2005, pp. 21-24).

Regardless of all claims and efforts I have seen, diagnosing (or debugging) a problem remains an art. As Grier discovered, technical support nowadays appears to be deskilled, having been shipped off to wherever it is economically most attractive. Going through scripts and reading manuals is a deskilled task, but diagnosing a problem remains an art and requires skill.

I opine that deskilling is essential for the Vernon cycle’s maturation and standardization stages. However, certain elements can’t be deskilled. For example, I doubt that the skill (or art?) Grier used to diagnose the software problem that he encountered can easily be automated. It’s unfortunate that we don’t appear to be very good at determining what can and can’t be deskilled.

I am inclined to agree with what Grier appears to indicate: Although global trade does indeed destroy local industry, it generally gives the world a higher standard of living. Unfortunately, our think tanks and policy makers seem to suggest it is good for local industry too.

An additional observation: Necessity is the mother of invention. If tasks, and the associated problems, are shipped off to different parts of the globe, it is highly likely that the solutions—and associated inventions—will be coming forth from those same locations. That can’t be very conducive to local research or innovation.

Raghavendra Rao Loka
rao@acm.org

DESIGN PATTERN COMPONENTIZATION

I was disappointed to see that “Componentization: The Visitor Example” by Bertrand Meyer and Karine Arnout (July 2006, pp. 23-30) made no mention of the significant prior work in the C++ community on pattern componentization in general and the Visitor pattern in particular.

Andrei Alexandrescu’s Modern C++ Design (Addison Wesley, 2001) demonstrates how a number of classical design patterns (including Visitor) can be componentized in the form of C++ templates, and his subsequent columns in C/C++ Users Journal (http://erdani.org/publications/main.html) include further discussion of the topic.

For example, Alexandrescu’s Feb. 2002 column titled “Typelists and Applications” (http://erdani.org/publications/cuj-02-2002.html) demonstrates how to add visitation support to an unmodifiable C++ hierarchy—that is, one without an accept function.


As Meyer and Arnout point out, multimethod support makes possible different approaches to addressing the Visitor problem.

Scott Meyers
smeyers@aristeia.com

The authors respond:

We thank Scott Meyers for providing these references. Much interesting
research goes on in the area of design patterns and their implementation; given Computer’s format, we made no attempt to cite all related work. Andrei Alexandrescu’s book is cited in Karine Arnout’s thesis and also in our longer article on the Factory pattern, references 8 and 9 in the Computer article.

While the articles Meyers mention are relevant, they do not directly address the topic of our work: pattern componentization. Our guiding idea is to replace patterns, as schemes that each programmer must build anew for every applicable development, with directly reusable, off-the-shelf components, each available to any developer through a simple API. We described such a component for the Visitor pattern in our article and for the Factory pattern and a generalized notion of Observer in companion papers. The work that Meyers cites does not describe a systematic componentization effort, such as we carried out for all the examples in Design Patterns.

Regarding the underlying design and implementation techniques, we encourage readers to compare the results: for Visitor, Alexandrescu’s work and our Computer article; for Observer, Sutter’s article and our reference 10 (http://se.ethz.ch/~meyer/publications/lncs/events.pdf). We think the language mechanisms and design techniques do make a considerable difference in simplicity, elegance, and efficiency.

Bertrand Meyer
bertrand.meyer@inf.ethz.ch
Karine Arnout
kbezault@axarosenberg.com

BAND-AIDS OR BODY ARMOR?
Zoltán Ádám Mann fails to dig deep enough in “Three Public Enemies: Cut, Copy, and Paste” (July 2006, pp. 31-35) to arrive at the true root cause of the “problem.”

What he actually describes is a symptom not a problem. The true root cause is a lack of proper configuration management (CM), which should be a key process of the SE infrastructure supporting software development.

Valid CM needs to occur on the personal level as well as for the formal project and enterprise assets. Perhaps Mann’s technique could be a part of someone’s personal CM, but it can never replace a complete CM process.

A proper information architecture (IA) for both documentation and software would help avoid the temptation to misuse the cut, copy, and paste commands. But proper CM is still a requirement that the program management office places on the developers.

IA is, of course, one of the supporting technical architectures for the overarching systems architecture. I wonder how many of Mann’s motivating exemplars had any true systems architecture supporting the development. Or even an IA. Or an underlying set of SE processes as a key part of the infrastructure to support the development project.

Without paying full attention to the entire gamut of required SE processes, doing anything else is akin to putting a Band-Aid on a gunshot wound instead of wearing body armor.

William Adams
williamadams@ieee.org

Editor’s note:
The August 2006 Computer article by Paul Debevec titled “Virtual Cinematography: Relighting through Computation” (pp. 57-65) refers to the first photograph, Joseph Nicéphore Niépce’s “View from the Window at Le Gras.” To view the officially sanctioned version of this plate, produced at the Getty Conservation Institute, visit www.hrc.utexas.edu/exhibitions/permanent/wfp.

We welcome your letters. Send them to computer@computer.org.