Why Explore Object Methods, Patterns, and Architectures?

STEPHEN J. MELLOR, Project Technology
RALPH JOHNSON, University of Illinois

These three technologies each offer the promise of a solution to the “software crisis.” Yet each camp advocates quite different means for achieving that solution. Resolving their conflicting views will likely require a re-examination of core assumptions.

Objects, patterns, and architectures have much in common. Each holds the promise of solving chronic software development problems: high development costs, even higher maintenance costs, low levels of reuse, unbelievable—and unrealized—schedules, and so on. Advocates have proposed excellent reasons—in some cases, elevating them to universal propositions—as to why their particular technology can “solve the software crisis.” All advocates stress domain analysis and reuse. The object methodologists and patterns people discuss architecture, and object-oriented programming is popular in both the architectures and patterns camp. They all use modeling notations, suggest processes for developers to follow, advocate tools for automating software development, and produce reference models.
PARADIGMS IN CONFLICT

In spite of their many similarities, the different camps’ proposals often conflict. Sometimes this is just because they use different definitions for words like “analysis,” “domain,” and “architecture.”

Everyone agrees that all topics are important, they just disagree as to which is most important.

This can lead to unnecessary misunderstandings, confusion, and even arguments when a listener assumes that a speaker has the same definitions, assumptions, and context.

Other times, people differ in what they consider most important. Each group tends to focus on some topics to the exclusion of others. In general, everyone agrees that all topics are important, they just disagree as to which is most important. This can affect the final conclusion.

Most object-oriented methods emphasize concepts, notions, and processes for developers. In contrast to the earlier structured methods, they use the same notation for both analysis and design, and try to reduce the transition from analysis to design by how they organize the models. Although tools are recognized as important, originally they did little more than help draw a method’s notation and navigate large models, and only later began to support semantics and code generation. Most methods target software development in general or focus on broad areas such as real-time systems. Reference models—the production of completed models that serve as a reference to others working in the same area—generally take a back seat.

In its purest form, a domain-specific software architecture is closely matched to a specific kind of problem. No one assumes that a DSSA be inherently object-oriented; indeed, an object-oriented architecture is just one possibility. Once the architecture has been selected, it provides the developer with a set of high-level abstractions in which to express the application, so the developer moves “backward” from the architecture to the problem specification. Notations and process both tend to be specific to the architecture, rather than universal. Tools may support specialized, sophisticated code-generation schemes. The software architecture is itself a reference model.

Work on architectural styles focuses less on the problem and more on the solution. Ideally, an architectural style can be applied to a broad range of problems. “Objects” is just one style among many. Each architectural style has its own notation that describes the system structure, vocabulary, and interaction patterns of systems built with that style. One consequence of being so broadly focused is that it is harder to make powerful tools and to specify complete processes, so less work has been done in these areas.

People who work with design patterns observe that certain patterns manifest repeatedly across applications, but in different guises. Contention for a resource, for example, appears in hotel reservation systems and in printer managers. This pattern can be described and reused in each of these systems, but with different names. The developer embeds the appropriate name into the generic logic of resource contention management. A developer working in this style connects a problem to its solution by recognizing that the pattern applies. Notations and processes are used eclectically. Reference models are all; emphasis is placed on documentation and literary style, rather than on tools or code generation.

Each of these descriptions is a caricature, but they show that practitioners and researchers in each area emphasized different issues. And in some cases, they conflict.

One example of a question with conflicting answers is “How should generic information be treated?” Much of the information in our software is generic to the problem domain, while other information is specific to the particular problem we are solving. How do we separate the two and encode the generic information so that it can be reused?

- Object-oriented developers tend to encode both generic and specific information in different objects and classes. Sometimes they use abstract classes to encode generic information and put specific information in the concrete classes. Other times the classes represent generic information and the specific information is in the scripts that instantiate classes, parameterize the instances, and connect them together.

- Domain-specific software architects encode the generic information in a DSSA supported by special-purpose languages, encode the problem-specific information using those languages, then translate from the special-purpose languages to the final code.

- People working with patterns view this as an unimportant question, because what is important is to discover the generic patterns and document them. The software developer can then decide whether to encode the patterns in objects, use a domain-specific language, or just use the patterns to write a single, specific program. The biggest gain comes from understanding the patterns and making them explicit.

Again, each of these points is oversimplified. Some object-oriented systems use special purpose languages and code generation, some patterns focus on how to separate generic information from specific information in an object-oriented system, and some DSSA articles focus on the patterns in
A Unified Object Topology, pp. 31-35.
William J. Tufteholtz and James Carver

This article offers a structure for understanding how kits, frameworks, and systems fit in with patterns, architectural styles, and domain analysis. The article applies the structure to show what information—specifically, patterns and architectural styles—are required to develop kits, frameworks, and systems, and how that information fits into a systems development lifecycle. The topology also identifies potential products and research areas.

Idioms and Patterns as Architectural Literature, pp. 36-42.
James Coplien

One of the leaders of the patterns community and the author of Advanced C++, Coplien describes his view of why patterns are important, and what they contribute to architecture. He argues that their value extends far beyond objects, that they are a powerful way of thinking about software development organizations as well as the architecture of the systems they develop.

Architectural Styles, Design Patterns, and Objects, pp. 43-52.
Robert Moore, Andrew Kampman, Ralph Melton, and David Garlan

This article shows the relationship between software architecture and object-oriented design, and shows that neither of them subsume the other because they address different (if overlapping) issues and use different abstraction mechanisms. The authors also argue that architectural styles and design patterns are complementary mechanisms for encapsulating design expertise, and that architectural styles are probably more directly comparable to object-oriented methodologies than to patterns.

Using Patterns to Improve Our Architectural Vision, pp. 53-59.
Nayyir K. Woha and Ward Cunningham

Christopher Alexander is an architect who invented the term “pattern” for use in building towns, homes, and offices. K. Woha and Cunningham show how concepts in patterns and architecture for building may be applied in software. They explore the relationship between objects and architecture and propose that Alexander's concept of a "pattern language" can be used to fill the gap between the two—especially as project size increases.

Recursive Design of an Application-Independent Architecture, pp. 61-72.
Sally Shier and Stephen Melton

This article defines a software architecture as a virtual machine that underlies a system. The authors use an object-oriented analysis method to model both the application and the architecture, and they use formalized patterns to describe how to map application concepts to the architecture. The result has the code generation ability of the tools for a domain-specific software architecture, but is based on general notations. The authors use the development of the Timepoint application to illustrate how their method works.

FIVE ATTEMPTS AT SYNTHESIS

And so to this special issue. We have selected five articles that each take rather different perspectives. However, all of them relate objects, architectures, and patterns. We asked the authors to focus on defining their assumptions and definitions so as to make more clear the reasons that they draw the conclusions they do. We believe that the articles make sense within the context that each defines. But we also believe that these articles are nonsense if you don't hold at least some of the assumptions that animated each article in the first place. For a summary of each article, refer to the box above.

So we ask you, as you read these articles, to focus your attention on assumptions, definitions, and context. When you read something that seems peculiar, or even absurd, think carefully about what assumptions and definitions would make this statement sensible in the mind of the authors.

To kick off this exercise, consider the questions in the box on page 30. Write down your answers on a separate sheet of paper. When you've finished reading the articles, come back and consider the questions once more. Compare your answers. At the least, we hope that you will find the questions more meaningful.

We began work on this special issue with the assumption that each of these topics holds out the promise of solving the chronic problems of software development. We further had the impression—and a few striking examples—that indicated that the paradigms were in conflict. Good fodder, you might think, for a special issue.

But as co-editors, we soon found that we had trouble understanding each other. Coming from different backgrounds—object methods versus patterns—had given us different vocabularies and we did not have an easy way to reconcile our different interests and concerns. Over time, we came to realize that a synthesis of these three topics will require that we break down core assumptions and look at why an assumption is so deeply embedded in a particular camp.

We hope that this issue's articles will help you understand both the similarities and differences in these fields and will help you appreciate and use work based on assumptions different than yours. 

IEEE SOFTWARE
DEFINING OBJECT METHODS, PATTERNS, AND ARCHITECTURES

1. When you see the term architecture, what do you mean?
   A. The high-level design of a system
   B. The structure of the lower-level, more architectural objects of the system, such as subprograms, databases, etc.
   C. The structure of a particular system layer, such as the architecture of the network interface on the architecture of the lan
   D. It also refers to the architecture of the system
   E. An architecture outside the structure of any system

2. When you see the term architecture, how much detail is implied?
   A. Not much because architecture is high-level
   B. Not at all because architecture is a high-level concept, it is not a detailed design
   C. A great deal of detail because it is a high-level concept, it is not a detailed design
   D. A great deal of detail because it is a high-level concept, it is not a detailed design
   E. It depends on the context

3. How do you think about architecture during the development lifecycle?
   A. Limit the number of people who are responsible for the particular project going on
   B. Think about it on a case by case basis, but try to use the same architecture principles across all architecture descriptions
   C. Think about it on a case by case basis, but try to use the same architecture principles across all architecture descriptions
   D. Think about it on a case by case basis, but try to use the same architecture principles across all architecture descriptions
   E. It depends on the context

4. When you see the term architecture, how do you think about it?
   A. To understand the overall system's design and how it interacts with the environment, that is, the context for the system
   B. A complete and detailed view of the system's design, including the environment
   C. The key components of a general purpose design that can be applied to other systems
   D. The key components of a general purpose design that can be applied to other systems
   E. It depends on the context

5. How do you think about an object in a greater context?
   A. Derive primarily from the name and function of an object
   B. Derive primarily from the name and function of an object
   C. Derive primarily from the name and function of an object
   D. Derive primarily from the name and function of an object
   E. Derive primarily from the name and function of an object

6. Should there be a seamless transition between analysis and design?
   A. Yes, our notation should map from the problem specification to the software
   B. Yes, our notation should map from the problem specification to the software
   C. Yes, our notation should map from the problem specification to the software
   D. Yes, our notation should map from the problem specification to the software
   E. No, the structure of the application and the structure of the software do not necessarily match each other

---

Stephen J. Mellor teaches and does research at Project Technology. He began his career at CERN in Switzerland, where he supported accelerator control systems. Subsequently, at Lawrence Berkeley Laboratory, he led a team responsible for providing control systems for several real-time applications. Mellor received a BA in computer science from the University of Essex in the United Kingdom.

Ralph Johnson is a senior research scientist at the University of Illinois. His research interests include object-oriented design, especially design patterns and the design of object-oriented frameworks. He is a co-author with Erich Gamma, Richard Helm, and John Vlissides of "Design Patterns: Elements of Reusable Object-Oriented Programming," and was one of the organizers of the original conference on Pattern Languages of Programs.

Johnson received a BS from Knox College and a PhD from Cornell. He is a member of the IEEE Computer Society and ACM.

Address questions about this article to Mellor at Project Technology, 2560 Ninth Street, Suite 214, Berkeley, CA, 94704; steve@projtech.com; or to Johnson at University of Illinois, Department of Computer Science, 1304 W. Springfield Ave, Urbana, IL 61801; johnson@cs.uiuc.edu.