Editor: Warren Keuffel ■ wkeuffel@computer.org

A Road Map to Java Software Development

Carlos H.C. Duarte

Java Application Frameworks *by Darren Govoni, Wiley Computer Publishing, New York, 1999, 0-471-32920-4, 410 pp., US $44.99*

Application frameworks are collections of related software artifacts that designers and programmers can instantiate, compose, or customize for specific purposes. They are developed to attempt to facilitate reuse by grouping together artifacts that encapsulate either the knowledge concerning an application domain or best-practice implementations. A set of component specifications for file system handling, written in an object-based programming language, defines a framework. *Java Application Frameworks*, a compendium of Java frameworks, related object-oriented notions, and heuristics, provides excellent coverage of this field.

Your Guide to Software Reuse

There is no goal in software engineering more compelling than effectively reusing software artifacts. The object-oriented approach is one of the best attempts to achieve this objective, and Java is currently the most successful programming language supporting this approach.

Nevertheless, software reuse based solely on programming-language constructs has been disappointing, not only because of the exponential complexity increase when the number of artifacts that could be reused grows, but also because theoretical results show that it is impossible to automate the entire reuse process. Some methodological guidance—as Darren Govoni provides in the form of structuring notions and heuristics—is required to increase the process' granularity level, thus making reuse more effective.

Reuse via Frameworks

Java Application Frameworks is well organized, starting with a description of object-oriented software development that focuses on frameworks. Govoni is pragmatic, presenting framework classification, composition, and complexity without relying on a purely definitional style. Along with the set of presented notions, he discusses issues of practical concern, such as whether to develop or buy a framework and the behaviors that emerge in some complex object systems due to the interaction pattern of their defining components.

Govoni then describes specific Java frameworks, gradually shifting from fine grain, low-level structures to higher-level ones. This is unusual in software engineering books, where the standard approach is to study software artifacts from a high-level perspective and proceed by applying some sort of functional decomposition. Govoni thus successfully orders his chapters, recognizing that understanding object-based notions and low-level frameworks is a prerequisite for reading the succeeding chapters.

Object-Based Notions for Everything

The main body of the book clearly explains how Java packages, frameworks, and software architectures support many standard ideas in software development, such as entity collections, patterns, components, the separation of user interfaces from the application, remote invocation of functionality, and persistence. Govoni addresses these topics, respectively, with the JDK Collections Framework (Chapter 2), the Gang of Four set of patterns (Chapter 3), the JavaBeans model (Chapter 4), the Swing framework supporting the model-view controller paradigm (Chapter 5), and Remote Method Invocation and Java Database Connectivity (Chapter 7). Suggesting we should adopt frameworks as the units of reuse when developing complex software systems, Govoni also presents a clear and uniform view of the built-in Java 2.0 class hierarchy and of other frequently used Java artifacts that third parties supply.

Good-Looking Technicalities

Another strong aspect of this book is its design—the three artists aren't acknowledged in the foreword for nothing. The humorous art makes reading the book—fully fledged with technical content such as code fragments—a lighter task, and the diagrams help readers understand the complex structures the book discusses. In addition, cross references alert readers to more information on the current and related topics, and the index lets readers use the book as a reference. The publisher's Web site (www.wiley.com/compbooks/govoni) also provides information related to the subject.

Falling Off Track

Unfortunately, readers won't make it through the book without experiencing a few difficulties. At times, such as during the informal definition of the framework notion, it is not clear if some of the properties Govoni discusses are inherent (such as customizability) or desirable (such as extensibility). It also would have been helpful to include a brief explanation of the UML notation with the diagrams. Fortunately, the author deals with these difficulties by providing numerous examples and extensively using the notions he introduces throughout the book. Thus, although the meaning of a few of his informal definitions might not be obvious at first, he clarifies them eventually.

In addition to this, the reader could certainly benefit from more connections with software process and architecture notions. This would offer a better understanding of how frameworks relate to these other notions and could potentially increase the granularity level of supported reuse.

Final Destination

Without a doubt, this is a well-written, carefully designed book presenting an alternative view based on frameworks of the Java paradigm. It complements other works that address the Java programming language alone. It will certainly be a valuable acquisition not only for intermediate and experienced Java software developers seeking to understand by example how to better structure and reuse their work but also as a reference text concerning the most widely used Java application frameworks.

Carlos H.C. Duarte is a member of the technical staff at BNDES, the National Bank of Social and Economic Development in Brazil. Contact him at carlos.duarte@computer.org.

An Easy-to-Use Guide to Use Case Driven Software Development

Martin Fogarty

Use Case Driven Object Modeling With UML: A Practical Approach *by Doug Rosenberg with Kendall Scott, Addison-Wesley, Reading, Mass., 1999, 0-201-43289-7, 165 pp., US $31.95.*

Use Case Driven Object Modeling With UML presents a practical guide to using the Unified Modeling Language to capture user requirements and produce the code necessary to fulfill those requirements. Rather than delving into the many nuances of the notation, Doug Rosenberg proposes a simplified bare-bones usage. (Kendall Scott wrote the book based on conversations with and e-mails from Rosenberg. For simplicity, I won't make this distinction in the review). Rosenberg couples this simplified usage with a methodology that emphasizes getting results—rather than a slavish adherence to the letter of the law. In keeping with this, the writing style is breezy and no nonsense, and the book is mercifully slim. All of this, along with the regular use of enticing words such as *practical*, *stream-lined*, and *simplified*, ought to guarantee a readership among those used to the style of some of the weightier tomes on this subject.

Describing the Methodology

Rosenberg begins by giving his credentials and doing a certain amount of judicious plugging for his company, Iconix. For those studying UML and the Rational Unified Process for the first time, note that the ideas Rosenberg presents predate and, to some extent, foreshadow these approaches.

The basics of the methodology presented are

- identify the real-world domain objects in a domain model, advocating grammatical inspection as a technique to arrive at these objects;
- produce, in parallel, a use case model using the evolving domain model as input (Rosenberg identifies real or paper prototyping as a key enabler for this process.);
- use a process called robustness analysis for first-pass identifica-

tion of the set of objects operated on during use-case execution;

- use interaction modeling to document the detailed interactions between the objects using UML sequence diagrams; and
- capture the objects' dynamic behavior through collaboration and state modeling.

This methodology's framework is the production of dynamic (problem space) and static (solution space) models. The steps Rosenberg describes facilitate the incremental definition of static model class diagrams that ultimately let us develop the code. Rosenberg devotes individual chapters to each of the major steps, explaining them and suggesting how to achieve them. The pitch at all times is that this is a practical approach that doesn't get hung up on irrelevant detail.

Herein lies the book's strength. Being relatively new to this area, I found the book excellent in that it presents a complex subject with great clarity. Although I didn't verify them all, the practical suggestions Rosenberg makes ring true and, more importantly, are realizable. One of the biggest problems in this area is that people propose methodologies that are too arcane and difficult to implement. The average analyst and system architect have enough to contend with without the tools and processes of the trade adding to the burden.

The methodology's use-case-driven and iterative nature is particularly compelling. One of the most common ways for large-scale projects to fail, particularly those with an elongated life cycle, is for the developed product to be out of step with user requirements. We can usually attribute this to poor links between the analysis and design phases or a failure to revisit the analysis over time to ensure that the premises on which it was based still hold. This methodology directly addresses the linkage problem and provides a framework to allow efficient iteration of the analysis and design when required.

Evaluating the Methodology

So does the methodology have any shortcomings? It is biased toward GUI centric projects and, as such, probably isn't as obviously applicable in all cases to embedded systems. In addition, developers must follow the full methodology to realize the gains of some of the techniques Rosenberg describes. This is not always an option for project teams dealing with legacy code and processes. However, these are relatively minor points and Rosenberg does not make any claim that the methodology is some sort of panacea.

I recommend this book to systems analysts and architects new to UML and interested in using some of the best practices out there. It serves both as a good tutorial and a reference.

Martin Fogarty is a Requirements Engineer at Motorola Cork. Contact him at Martin.Fogarty@Motorola.Com.

Software Entropy: An Emerging Perspective

Robert C. Larrabee

Chaos and Complexity in Software, Challenging the Industry and the New Science *by Robert Bruce Kelsey, Nova Science Publishers, Commack, N.Y., 1999, 1-56072-669-5, 183 pp., US $34.00.*

Robert Bruce Kelsey takes an interesting approach to the topic of chaos and complexity in software. He starts by discussing historical works on Roman bridge building, speculating on why some bridges failed and others didn't—for example, Caesar's large bridge over the Rhine river didn't fail. Was there a gap in the understanding of civil engineering that required a different approach for successfully building a bridge?

Kelsey also draws heavily on the social sciences (history, psychology, philosophy, and even literature) in addressing software complexity—which is the latest trend. Current research in software engineering seems to indicate that the highest return-on-investment leverage techniques result from the management of technology (psychosocial, leadership, group dynamics, and so forth) rather than the technology itself.

The software engineering field exhibits an inherent chaos that is not yet quantifiable, and Kelsey's central question is, "Can chaos and complexity treatments used in other engineering fields be imported into software engineering?" His writing is interesting and broad in scope, but his work is not an academic textbook—rather it's an expression of his interest. It appears he wrote it for the curious, reflective audience, but it requires at least a journeyman's experience with software engineering. Technical readers will be just as interested in this work as managers and engineer inclined toward philosophy.

The New World

The book opens with "Prologue: The Brave New World," which describes how modern chaos and complexity have come to supplant the less-complete Newtonian perspective of the world and how nonlinear dynamic systems resist classical treatment and explanation. For some, understanding this new order seems to be a "transcendental impossibility." Kelsey argues that software falls into chaotic and complex categories, and any practitioner will likely be compelled to agree. He even suggests that statistical and empirical treatment of software quality (for example, statistical process control and the Software Engineering Institute's Capability Maturity Model) might be limited in what they can accomplish.

The New Paradigm in Software

In his chapter "The New Paradigm in Science," the new paradigm Kelsey describes is chaos theory as applied to software engineering. He explains that its benefit is the systems understanding it yields. Complex software systems are not amenable to deterministic treatment, and even probabilistic treatments are of limited utility. Kelsey argues that we need a sharper tool and suggests that this sharper tool might be applied at the system level, rather than at the small-pieces levels.

Every practitioner is aware of chaos in software, although it is unfashionable to use that term in a business environment. Kelsey suggests that we can't move to the industrialization phase of the software industry until we accept and even embrace chaos (Pogos' observed: "We have met the enemy, and it is us"). Kelsey presents a new way of pragmatically viewing chaos, which mighty be thought provoking to software architects and testers.

In his chapter, "The Science of Complexity, The Complexity of Science," Kelsey defines a systemic approach to complexity that sounds similar to Peter Senge's systems thinking (*The Fifth Discipline*, Doubleday, 1994). Each method looks at complexity through a new paradigm and uses unfamiliar constructs and graphs. For instance, the notations in these approaches do not resemble the familiar state transition diagram with which I am comfortable. Again, Kelsey argues that the chaos theory presents a macro model for complex software systems and yields a practical understanding not available from classical methodologies. Chaos and complexity, although known, are herein somewhat rigorously defined (just what *is* chaos, anyway?).

A Synthesis of Ideas

This book is a novel approach to a topic largely unexplored. It synthesizes the current, relevant works of chaos and complexity in physical nonlinear systems with philosophy and software engineering. You can view it as a prelude work to the quantification of software entropy, as Kelsey suggests.

In some regards, this book is similar to Samuel C. Florman's classic *The Existential Pleasures of Engineering* (St. Martin's Press, 1996). I found them equally stimulating and fascinating. My wife, a practicing psychotherapist, found Florman's book unremarkable but was fascinated by *Chaos and Complexity in Software*. Both books apply humanities concepts to the field of engineering, and Kelsey's book offers the software engineer or researcher stimulating ideas and a thorough top-level survey of the topic.

Robert C. Larrabee is a Senior Staff Engineer for ARINC Research. Contact him at larrabeerc@ieee.org.