The end of the book offers recommendations and suggestions for running and documenting experiments in SE. The recommendations are grouped in the last chapter to provide a quick reference and reminder for readers.

As it is conceived, both researchers and software developers can use this book. Researchers will be able to use the book to formally test the features of new software development artifacts generated as a result of their research. In this manner, they will be able to rigorously study the behavior of their discoveries under different situations and define the best conditions of applicability. They will then be able to demonstrate to industry what benefits these artifacts offer, something for which the software industry has been clamoring for some time. Many software development organizations find they must choose between development artifacts—without quantitative data to determine each one's benefits. This book can aid decision makers in putting together experiments that output the data they need. Developers can also use this book to analyze the impact of innovations in their software development practices.

Finally, I expect the topic of experimentation will be included in SE educational programs sooner or later, and a publication of this sort would make a good textbook.

Marta Lopez is a competence manager at the Fraunhofer Institute for Experimental Software Engineering in Kaiserslautern. Germany. Contact her at lopez@iese.fhg.de.

Generative Programming

Arie van Deursen

Generative Programming: Methods, Tools, and Applications, *by Krzysztof Czarnecki and Ulrich W. Eisenecker, Addison-Wesley, 2000, ISBN 0-201-30977-7, 832 pp., US$44.95.*

Does software engineering resemble car manufacturing? Is it possible to construct software from components in the way cars are assembled from parts, thereby inheriting such benefits as reduced costs and shorter time to market? Generative programming addresses these questions, concentrating on component composition—that is, the way in which applications are constructed from reusable components. In *Generative Programming*, configuration knowledge is made explicit through domain analysis and implemented via various software generation technologies. This book presents the authors' generative vision as well as comprehensive surveys of the underlying methods and tools, which mostly have their origins in research in systematic software reuse.

To understand the nature of configuration knowledge, the authors use a factory metaphor. In a car factory, configuration knowledge is the difference between a pile of car parts and a ready-made car. This knowledge is embodied in the assembly line. It not only includes one-of-a-kind production but also all sorts of customizations and special combinations. As an example, a single Mercedes-Benz assembly line in Sindelfingen, Germany, produces all the C-, E-, and S-Class models; the E-Class alone has more than 10,000 seat variants and 8,000 cockpit variants.

Using the car production metaphor, generative programming looks at software development as the production of application systems tailored toward a particular problem domain. We can express variability within that domain concerning, for instance, application-specific features using a domain-specific language. Code generators take care of translating the resulting feature combination specifications into executable code by deriving, specializing, extending, and assembling library components.

Explanation and example

The first part of the book covers domain engineering—the activity of organizing past experience in building systems in a given application domain. The result is a feature model listing the common and variable features as well as dependencies between the variable features. With this feature model in hand, the domain engineer can design a domain-specific language for configuring new systems in the underlying application domain.

Several chapters then explain implementation techniques that can deal with configuration requirements not easily expressed in traditional object-oriented approaches. As an example, the chapter on aspect-oriented programming covers cross-cutting features, such as persistence, monitoring, and secu-

To understand the nature of configuration knowledge, the authors use a factory metaphor.

rity, which potentially affect all classes in a framework. Aspect-oriented programming uses code generation to express such cross-cutting features concisely. Another implementation technique covered is static metaprogramming in C++. Template expansion in C++ is Turing complete, and the authors advocate using this to realize all sorts of domain-specific optimizations and specializations at compile time. The 100-page chapter covers several clever encodings but is hard to grasp without a good understanding of C++'s template mechanism.

The final part contains three case studies in generative programming. The most involved one deals with matrix manipulations, where template metaprogramming is used to arrive at a matrix manipulation library that is highly flexible, supporting all sorts of operations and matrix types, while still achieving the efficiency of hand-crafted Fortran libraries.

Strengths and weaknesses

Generative Programming makes a number of important contributions. The individual chapters are comprehensive overviews of such areas as feature modeling, generic programming, aspect-oriented programming, and static metaprogramming in C++. Several of these surveys are more accessible than the original publications they are based on. Moreover, the rich list of references makes it easy to find the original material if needed.

Furthermore, the book provides a grand vision of how the methods and tools discussed are connected, putting existing technologies in a new perspective. The authors coined the notion of generative programming, giving their vision a prominent position on the software engineering research agenda.

However, the book might be hard to use for teaching purposes. It covers many different subjects, but some chapters are quite large (75 to 100 pages), and the book does not include exercises. Moreover, several chapters require advanced C++ knowledge. This is unfortunate, because some of the techniques covered are essentially independent of C++ and could easily have been discussed at a more abstract level.

Another problem is that the book includes several chapters in which the authors get carried away by their enthusiasm, failing to keep proper distance from their subject matter. An example is their discussion of the

> **The authors coined the notion of generative programming, giving their vision a prominent position on the software engineering research agenda.**

Intentional Programming system. The authors herald IP as achieving "natural notations, great flexibility, and excellent performance"—and this is just the first of 10 IP benefits listed. Moreover, the authors do not provide a similar list of IP's disadvantages and problems. Also missing is an overview of earlier attempts to achieve such goals, for example in the Lisp, artificial intelligence, and programming-environment research communities. The authors do not explain why these earlier attempts failed and IP succeeded. Several chapters needed a more critical discussion to be more convincing.

Last but not least, after having studied more than 700 pages of generative programming technologies, I deserved a summarizing concluding chapter. The book ends with the application of generative programming principles to the (complex) domain of matrix manipulations, requiring considerable C++ knowledge from the reader. I would have liked to have seen a summary of what I should have learned from the book, an evaluation of the strengths and weaknesses of generative programming, an analysis of potential impediments for adopting generative programming in industrial practice, and an agenda of research questions that should have helped to advance the theory and practice of generative programming.

In short, then, is this book worth reading? The authors present generative programming as a paradigm shift, and Jim Coplien, in his foreword, characterizes the book as "a harbinger of a broader enlightenment that opens the door to a new age." You might want to buy the book just to find out whether or not you agree with this point of view. Another reason to buy the book is to learn about the methods and implementation techniques of generative programming. You will learn about high-tech approaches that you might not need in every project, but with this book in hand, you are in a position to make informed decisions about the cost-effective use of generative solutions. If you take software reuse seriously, you can't afford to miss this book.

Arie van Deursen is senior researcher at CWI, the National Research Institute for Math and Computer Science in the Netherlands. His interests include reverse engineering, domain-specific languages, and object-oriented programming. Contact him at arie@computer.org.