courses. I liked the book, although it is not exactly light reading. I would recommend it for practitioners who are exploring software engineering methods.

— *N. R. Mead,* Pittsburgh, PA

REFERENCES

[1] WITT, B. I.; BAKER, F. T.; AND MERRITT, E. W. *Software architecture and design: principles, models, and methods.* Van Nostrand Reinhold, New York, 1994.

GENERAL TERMS: DESIGN, LANGUAGES, VERIFICATION

D.2.2 Tools and Techniques

See: 9709-0628 [D.1.5]; 9709-0647 [D.3.2—*Eiffel*]

Computer-aided software engineering (CASE)

See: 9709-0632 [D.2.1—*Methodologies*]

Modules and interfaces

HANSON, DAVID R. (Princeton Univ., Princeton, NJ) 9709-0633

C interfaces and implementations: techniques for creating reusable software.

Addison-Wesley Longman Publ. Co., Inc., Reading, MA, 1996, 544 pp., $40.83, ISBN 0-201-49841-3. [Addison-Wesley professional computing series.]

Hanson's book will teach experienced C programmers how to code reusable modules in C. It includes some well-known data structures but is not a data structures and algorithms text, nor does it use some computer science ideas that are rare in the C programming community. You could use this book as a litmus test for computer professionals. If they like it, they are really C programmers. If they turn pink and throw it out, they are really computer scientists.

The book covers interfaces and implementations (1 chapter), exceptions and runtime error checks (1 chapter), memory management (2 chapters), arithmetic (3 chapters), threads (1 chapter), and some data structures (11 chapters). The data structures covered are lists, hash tables, sets, dynamic arrays, sequences, double-linked deques, bit vectors, and various kinds of ASCII character string (four chapters). There is a useful set of appendices, a bibliography, and an index. Each chapter includes examples and exercises, which make it suitable for classroom use.

A properly coded C module should have a shared public interface (a ".h" file) and a private or hidden implementation (a ".c" file). Each chapter in the book presents a module with these two parts. First, Hanson gives the interface. Second, he uses it in an application program. Third, he presents an implementation. Other implementations and applications are set as exercises. This pattern is one of the book's strengths.

Hanson is a superior C programmer who uses literate programming. This process generates both testable programs and readable narrative from a common source. The goal is to create an easy-to-read explanation of the code that also defines the code. The code in the book has been extracted, "tangled," compiled, and tested. Notice, however, that the code itself contains no comments. It is split into small chunks that are embedded in a large commentary. The readability of the narrative is another of this book's strengths.

The book's strengths lead to weaknesses. Hanson uses many tricks of the C trade: do-loops that do not loop, switches that do not switch, macros, unions, *void**, *longjmp*(), and so on. But even given his skill with C, Hanson fails to write statically checkable generic modules or runtime polymorphic functions. C's *void** is used as an opaque type instead. This is the C tradition. It works when application programmers follow the rules, but is unpredictable otherwise.

Hanson follows the C tradition of limiting interfaces to C macros, declarations, and function prototypes. Modern computer science texts include formal rules defining the use of each function (pre- and post-conditions). So do the interfaces defined in the C++ Standard Template Library. Hanson does say when a call would cause a "checked run time error," and when it may not work because of an "unchecked error." You must study the informal commentary and the implementation to try to figure out any other rules.

A skilled and literate C programmer creates plausible and testable code that can have hidden assumptions. If these assumptions are false, the programs can have bugs and redundancies. Here are the examples I noticed in this book.

On page 203, the code for calculating the number of "unsigned long int"s that can store a given number of bits can underestimate the correct answer when the number of bytes is not a power of two. The code on page 122 looks as if a function call is misplaced. I spent 30 minutes convincing myself that the code is incorrect. Three days later, I saw the trick and spent 30 more minutes convincing myself that the code is correct. A comment would have saved me 60 minutes.

Literacy is no guarantee of clarity. On pages 79 and 80, and on 94 and 95, a pointer is carefully aligned with the union of all the elementary data types plus *void** and pointers to functions. It looks as if the memory allocation routines are designed to work only with the data types listed in the union. This should be clarified by a more rigorous narrative or comments. I did not notice any other problems in the code. This defect density is much lower than in most of the C source code I have debugged over the years.

This is a good book for practitioners, but it will not work as a textbook in a modern baccalaureate program in computer science. It is too advanced for beginners, and it is too incomplete for data structures and algorithm courses. The analysis of algorithms, and several classical sorting algorithms, are missing. Trees appear only as an alternative to hashing in an implementation of the "table" data type in Exercise 8.2. The book does not have enough on analysis, design, and process to be a software engineering textbook. Instead, I will be recommending it to people who ask for a book to improve their C programming.

— *Dick Botting,* San Bernardino, CA

GENERAL TERMS: DESIGN, LANGUAGES

User interfaces

See: 9709-0631 [D.1.5]; 9709-0636 [D.2.10—*Methodologies*]

D.2.4 Program Verification