pointing out that oversimplification. Suggested readings at specific points along the way would have helped.

Reviewed by Brian O'Laughlin, 1903 W. Evergreen, Chicago, IL 60622 — briano@tezcat.com

Foundations of Software Measurement

Martin Shepperd

Foundations of Software Measurement is written by Martin Shepperd, and published by Prentice Hall 1995 (paperback), ISBN 0-13-336199-3, 234 pages, $38.00

Software measurement is a growing interest in the software community: There is a call for votes on Usenet for the proposal to begin a new newsgroup, "`comp.software.metrics`". Major journals (such as "*IEEE Trans. on Software Engineering*"), frequently have contributions on the topic. A newcomer to such articles may find them quite difficult reading. Martin Shepperd's "*Foundations of Software Measurement*" offers the necessary background for such readers.

The book has two parts: "*Foundations*," introduces the reader to measurement and measurement theory –the basis for evaluating measures. He introduces measurement types (nominal, ordinal, interval, ratio, and absolute), and attributes (internal v. external). While only addressing introductory level measurement theory, Shepperd manages to show its practical usefulness by applying it to the popular McCabe Complexity metric. He finds two problems. First, the attribute "complexity" is problematic, since the meaning of complexity is unclear. What's complex to me may be simple to you, and vice versa. The second, and lesser problem has to do with how complexity is measured: McCabe measures decision counts, but the measurement rules need to be very explicit: for example, in a switch statement, it is unclear if each "case" statement should be counted as a separate branch. Shepperd concludes that under the facade of McCabe's Cyclomatic complexity measure lies a powerful idea: control flow branching is a worthwhile measure in its own right, and is useful in predicting test effort.

A distinction the reader gets from this examination is that measures are different from prediction systems: Measures are used in prediction systems, but they are validated by reference to measurement theory. For example, "Lines of Code" is not a "bad measure." It is a good measure of program length, and a poor predictor of project maintenance effort or defects.

Since the 1970s, software architecture has been recognized as having a huge effect on software quality. Design choices may result in highly complex code, which in turn produces maintenance difficulties. Shepperd discusses the influential coupling and cohesion model. Module cohesion refers to the singleness of purpose of a module; module coupling is the degree of independence of one module from another. Optimally, the software designer should seek to maximize cohesion and minimize coupling. Having read the previous chapter, the reader will be asking by now, "How can we measure the 'purpose of a module'?"

Shepperd discusses the various efforts to quantify software architecture, focusing especially on information flow metrics. These metrics, which are related to coupling, measure how data flows between modules. Specifically, they measure data emanating from a module and data terminating in a module. Shepperd introduces these concepts and guides the reader through an in-depth example of a simple reactor controller. The metric points to a module that has a relatively huge number of information flows. He fixes the problem by delegating some of its responsibilities to new modules. The result satisfies the information flow metric, and reduces coupling. Coders will probably see some of their own coding experience flash before them as they read this.

Software measurement is a typical tool for those concerned with software quality. There are two types of quality metrics: diagnostic quality metrics and discrepancy quality metrics. Diagnostic metrics are used to identify and predict problems before they occur in systems. Examples are "cyclomatic complexity" and "module fan out". These identify problems in code, ideally prior to integration. On the other hand, discrepancy metrics count incidents or problems that actually occurred in the software, and are used in statistical process control. An example would be a simple count of defects. Discrepancy metrics are often used as feedback for the software development process.

Those managing software projects are typically asked to predict resource requirements for a project, to understand the current status of a project, and to deliver it on time and with the appropriate quality. Shepperd discusses effort prediction and productivity in depth. Software development productivity is problematic, since it isn't clear what the output unit of measure is. Just what is it that software engineers **DO**? While it's true that they write code, measuring, for example, lines of code per person month is riddled with problems: It penalizes parsimonious code, it is language dependent, and it can be manipulated. Function points offer an alternative, though they are geared towards commercial and information system products. Nevertheless, they are available earlier in a project than "lines of code". Shepperd discusses several such models of productivity.

The second part of the book, entitled "*Supporting Topics*", begins with a formal algebraic specification of a system architecture. Formal models allow a more precise study of specific areas of a system. They offer a means of specifying the counting (or "observing") of a system more exactly. Formal models can provide a theoretical basis for empirical research. If a measure is theoretically flawed, there's no point in spending the money to empirically validate it. And there's certainly no point in developing a tool to perform the measure automatically.

From formal models Shepperd moves into empirical analysis – the other side of metric evaluation. After a brief overview of relevant statistics like regression and factor analysis, he gives an example of using data analysis to measure development:

Can function points predict duration of a project and hours worked on a project? His data set includes estimated hours, actual hours worked, actual project duration, and software size in function points. Using various regression techniques, his resulting R-squared — i.e., the amount of variance in actual hours that function points explain — is rather low at .258. (If it explained all of the variance, the R-squared would be 1.00). He concludes that other factors must significantly influence actual hours.

The last section changes focus from software engineering products to the processes that create them. Suddenly, a rather large, non-deterministic element is introduced in the model: people. Nevertheless, the issue of having sound models with which to guide observation applies. Further, process is clearly an important factor, and there is consensus that quantitative analysis of process, though just emerging, is critical. Shepperd discusses several types of models, including Data Flow Diagrams and life cycle models.

This is an excellent introduction for students, researchers, or software professionals interested in software measurement. Shepperd makes his subject matter very accessible without oversimplification, and each chapter includes suggested readings and exercises, to further explore this challenging area. I highly recommend this book.

Reviewed by Brian O'Laughlin, 1903 W. Evergreen Ave., Chicago, IL 60622 USA — briano@tezcat.com.

The Object Primer

Scott W. Ambler

The Object Primer: Application Developer's Guide to Object-Orientation is written by Scott W. Ambler, and published by SIGS Books, 1995 (paperback), ISBN 1-884842-17-8, 248 pages, $35.00.

Object-oriented (OO) analysis and design techniques are the focal point of this book. Although the book is relatively short, it manages to present the basics in introducing the reader to the subject. Highly technical readers will most likely find the book lacking in depth, however, someone with little or no OO background who is ready to learn OO technology will probably find the book helpful without being overwhelming.

The first chapter of the book provides a comparison of structured software development with object-oriented software development. Each approach is described and then a short example is used to demonstrate how each development strategy would be used to implement the example.

The second chapter deals with the potential advantages and drawbacks of using OO. The standard benefits associated with OO are presented (Reusability, Extensibility, etc.) as are potential benefits such as increased chance of project success, reduced maintenance burden, reduction in application backlog and the ability to deal with complexity. Although listed as potential drawbacks, several of the items in this list could be viewed as potential benefits e.g. "Developers must work closely with users." I would probably promote this one as a potential benefit.

The next two chapters of the book deal with the actual "how-to" of gathering user requirements and then ensuring the requirements are complete. The author details CRC (class responsibility collaborator) modeling to the level of choosing the CRC team, running a CRC session, and even arranging the CRC modeling room. The author makes liberal use of examples and case studies, however, they are not always presented logically within the text of the chapter which lends to some confusion on the part of the reader. The second part of this section discusses Use-Case Scenario Testing. This section provides the "how-to" aspect as well as a section on why this is a necessary part of the OO development life cycle.

In the fifth chapter, basic OO technical terminology and concepts are introduced. Terms are defined using concise wording and again, liberal use is made of examples to assist the novice in understanding basic OO terms. Although the author uses his own OO modeling notation, basic concepts such as instance relationships, aggregation and collaboration are easily understood by the reader.

Building on prior chapters, the reader is introduced to class modeling in Chapter 6. This chapter is another "how-to" and draws the reader further into the more technical side of OO. A significant portion of the chapter is comprised of a case study which allows the reader to put into practice what has been presented previously.

The final chapter of the book, entitled "Putting It All Together: OO in Practice" covers several different aspects of the OO software process without providing a significant amount of depth on any particular one. It is the most introductory in nature of all of the chapters in the book. However, this does not prevent the author from presenting some valid and useful points to the reader.

Overall, I believe Mr. Ambler's book meets its objectives as a primer. His use of examples and case studies help the reader to grasp what will be for some an introduction to a completely foreign approach to software development.

Reviewed by Suzette Person — 102264,1242@Compuserve.com

Rapid Software Development with Smalltalk

Mark Lorenz

Rapid Software Development with Smalltalk is written by Mark Lorenz, and published by SIGS Books 1995, (paperback) SIGS Books ISBN 1-884842-12-7 Prentice Hall ISBN 0-13-449737-6, 210 pages, $24.

Software developers want to build systems quickly yet deliver results that have high quality. Methodologies such as Rapid Prototyping combined with software tools such as Smalltalk ought to assist in achieving these goals, and in the book