Scientific Programming 17 (2009) 275–277
DOI 10.3233/SPR-2009-0284
IOS Press

Book Review

David H. Bushnell
ARC-TI, Metrica, Inc., USA
E-mail: david.h.bushnell@nasa.gov

Beautiful Code, by Andy Oram and Greg Wilson (eds), O'Reilly Media, Inc., Sebastopol, CA, USA, 2007, ISBN: 0596510047.

Introduction

"Beautiful Code" is an intriguing title for a book on software. As developers, we occasionally talk about beautiful code, but the idea of devoting an entire book to it may seem a bit questionable. After all, our daily responsibilities have more to do with our products' correctness and scalability, ease-of-use and salability than anything as tenuous as the beauty of something no-one but our fellow coders will ever see.

But consider beauty applied to some of the hardest, most objective areas of human endeavor imaginable: bridge design, quantum physics, and abstract mathematics. Among civil engineers and architects, John and Washington Roebling are famous for the graceful lines of the Brooklyn Bridge. Physicists talk of the "double slit, single electron diffraction experiment" as the most beautiful in physics (http://physicsworld.com/cws/article/print/9746). G.H. Hardy's famous book, "A Mathematician's Apology", is a paean to beautiful mathematics:

> The mathematician's patterns, like the painter's or the poet's must be beautiful; the ideas like the colours or the words, must fit together in a harmonious way. Beauty is the first test: there is no permanent place in the world for ugly mathematics.

But what of programming? Is beautiful code either necessary or desirable? Product managers certainly do not schedule for it. Customers do not care. It is the first thing that even programmers drop when a deadline looms. One person, when asked to contribute to "Beautiful Code", refused because:

> Yes, we all strive for beautiful code. But that is not what a talented young programmer needs to hear. I wish someone had instead warned me that programming is a desperate losing battle against the unconquerable complexity of code, and the treachery of requirements. (Jonathan Edwards, http://alarmingdevelopment.org/?p=79)

And yet, given a choice, is there a programmer alive who would be happier working on ugly code than beautiful code? Is there a programmer alive who has not at some point apologised, "This is an ugly hack, but the schedule was tight. . ."? Is a beautiful design or a beautiful algorithm not something we all would be proud of? So there is something to the thought that beauty matters even in programming.

Surveying the book

We often hear the complaint that practitioners in software development know something about how to design or code, but they do not seem to know much about creating code that others can read and appreciate. Since most code is read far more often than it is written, this is a problem.

Andy Oram and Greg Wilson have recognised this problem and created this book to address it. As Greg Wilson writes in the forward,

> Architects are taught to look at buildings and composers study one another's scores, but programmers – they look at each other's work only when there is a bug to fix; even then, they try to look as little as possible. We tell students to use sensible variable names, introduce them to some basic design patterns, and then wonder why so much of what they write is so ugly.

This book is our attempt to fix this.

"Beautiful Code" starts from a great idea, but ultimately disappoints with its delivery. The idea was to ask great software developers and computer scientists to describe in depth the most beautiful piece of code they knew. The results would be collected in a book and we could all learn to produce beau-

tiful code by reading the essays and applying their lessons.

Unfortunately, something was lost when this great idea was put into practice. What went wrong?

For one thing, the essays in "Beautiful Code" have no discernible order; they are a mish-mash. Code, design, and architecture follow each other without pattern. So there is no common theme or progression to help you as you move through the essays. What you learn in one has no relation to what you learn in the next. There is no opportunity to show how good architecture leads to good design and on to good code.

The haphazard arrangement and lack of progression are particularly apparent in the repetition. Some things (regular expressions, ruby, python) are explained several times. Other things (Perl in Chapter 11) are assumed as background.

The multitude of languages (C, Perl, Fortran, Lisp, Scheme, Haskell, and so on) also presents problems. This is not to say that a book must use only the popular language-du-jour. Learning new and unusual languages can stretch your mind. And sometimes (software transactions and Haskell) a particular language makes something especially easy or clear. But having too many in a book like this just distracts the reader. Very few of us are going to know them all and the authors must waste their allotment of pages on language tutorials unrelated to the purpose of the book.

The better chapters

So that is what is bad about "Beautiful Code". What is good about it?

What is good is that among the chapters are some of the best writing about software you will find anywhere. The better chapters start with an interesting problem, lead you to a solution that is clean and elegant, and end with a guide to where you can go from there. Here are a few of those better chapters.

A Regular Expression Matcher by Brian Kernighan

This chapter (the first in the book) is a wonderful presentation on regular expression matcher first written by Ron Pike. The matcher is a simplified version of grep, but even so it is surprising how much can be accomplished with a few lines of well thought out code. The chapter starts with a very quick history of regular expressions and their utilities in the Unix world, presents a stripped-down version that still preserves the most interesting aspects of the problem, and ends with an overview of roads not taken and further extensions.

Subversion's Delta Editor: Interface as Ontology by Karl Fodel

Karl was one of the authors of Subversion's delta editor. (For those not familiar with Subversion, it is one of the more popular source control tools.) The "delta editor" handles a problem whose importance should be clear to any developer: computing the differences between two directory trees in terms of how one tree must be modified to produce the other.

In this essay, Karl presents the editor's design via its API. As he walks you through the requirements, you learn some of the subtleties involved. But then you learn how, given the right perspective, these subtleties lead not to a complex and ugly design, but to one that is clean, simple and elegant. Finally, you learn how well the design has held up through Subversion's evolution. This is the definitive sign of a beautiful design.

Distributed Programming with MapReduce by Jeffrey Dean and Sanjay Ghemawat

Distributed programming is hard and that has helped Google's MapReduce system become such a popular topic of study. This chapter presents a high level view of that system.

Jeffrey and Sanjay make writing about beautiful code seem easy. Just start with a problem statement sure to catch your attention: use thousands of computers to count unique word occurrences in 400 terabytes of text spread over 20 billion documents. Present a naïve solution that anyone could write for counting the unique words in twenty documents on their own machine. Extend the solution step by step to something that solves the problem but is complex and ugly. Introduce MapReduce and it all falls into place: a beautiful solution is obvious. If only it were this easy for the rest of us.

Writing Programs for "The Book" by Brian Hayes

Brian Hayes is well known as the author of the "Computing Science" column for the American Scientist magazine. In this 33rd and final chapter of "Beautiful Code", Brian writes about a seemingly simple problem from computational geometry, deciding when three geometric points are collinear.

The chapter follows the familiar path for the well written essays in this book: motivate the problem, solve it in a way the readers will instantly understand, show where that solution fails, and guide them through the

pitfalls to the ultimate solution. Brian does this all with such practiced skill that along the way the readers will learn both beautiful code and an interesting bit of computational geometry.

In summary

In the end, "Beautiful Code" suffers from serious flaws offsetting its important strengths. It would have been a better book with fewer contributors presenting longer studies under stronger editors enforcing a more coherent theme. This is a book that you, the programmer and designer, will find worth your time. But I would be satisfied to check it out from a library or borrow it from a friend.