Extracted from:

No Fluff, Just Stuff Anthology

The 2006 Edition

This PDF file contains pages extracted from No Fluff, Just Stuff Anthology, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragmaticprogrammer.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

Copyright © 2005 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Chapter 15

Buried Treasure

by Glenn Vanderburg

Glenn Vanderburg has been a programmer through only the second half of the history of computing, but he's interested in the first 30 years, too. For years he has dreamed of teaching a course on the topic. That's just one reason he's delighted that people are discovering the practical value of knowing our history. Glenn is a consultant who lives in Plano, Texas, with his wife, Deborah, and their sons, James and Daniel.

Glenn talks about some recent tool and book favorites starting on page 215.

Over the past three years, many of my talks for the No Fluff, Just Stuff symposium series have shared a common theme. It was partly conscious, but mostly it came naturally, as a reflection of where I think our field is going.

I think our field is going backward.

And it's not a minute too soon. For years, we've been fighting our way forward, step by harried step, but for the most part it has been down the wrong path. The grass looked greener here—or at least, better manicured—but traps are lurking here, some of them very well concealed. We keep falling into them, but we keep fighting on. "We must be more careful!" we say, calling over our shoulders to our companions as we walk toward the next pit.

But some in the programming field have started to remember another place, one we passed on the way. It was a little unkempt and overgrown, to be sure, and maybe there were just as many dangers—but somehow the place, overall, was less dangerous. Plus, people who ventured in there keep telling us about the riches to be found in that place—wonderful treasures buried just below the surface.

The reasons *why* these older ways turn out to be better are subtle and occasionally complex, and I don't claim to understand them all. Whatever the reasons, the signs of what's happening are clear. Let's look at those first and then try to make sense of the whys and wherefores.

15.1 The Signs

The signs that we're returning to older stomping grounds are everywhere. Those of us programmers who know the history of our field spotted them early (although I certainly wasn't the first). Now they're so prominent, and growing so quickly, that many people have spotted the trend. The signs I've noticed tend to fall into a few distinct areas: the way we go about designing and building systems, the kinds of programming languages and techniques we employ, and the way languages and platforms are implemented.

Design

The way programmers and teams of programmers *design* software is changing. After decades of increasing investment in tools and disciplines to support an analytical approach to software design, our field

CLICK HERE to purchase this book now.

is running headlong toward a more empirical approach based on iteration, trial and error, and rapid feedback. There is widespread acknowledgment that the task of software design is simply too complex to tackle with a purely analytical approach. Programming will always involve a lot of careful thought, of course, but we must also be guided by feedback, checking our assumptions against the hard realities of real systems and running code.

The modern approaches to design aren't precisely the same as the older approaches from the 1960s and 1970s, but they share many of the same characteristics. A prime example is the emphasis on iterative development. Long before it became fashionable to try to design a program completely before beginning programming, the common practice was to build a simple, working system and gradually enhance it. Stories are even told of Marvin Minsky at MIT taking this practice to an extreme, beginning development by starting to debug an empty program. The modern equivalent of that, of course, is test-driven development. Guiding our development with automated tests is relatively new, but developing in small increments, evolving the design as we go, has a long history.

Another sign: today we are beginning once again to emphasize code over pictures in the design process. Don't get me wrong—we'll always draw pictures of our systems from time to time; that's something programmers have always done. But as the centerpiece of the design process, UML and other graphical notations have clearly failed. After having tried for years to improve software design by focusing on graphical models before we start writing code, programmers have learned something crucial. Code—good clean code, at least—is a more expressive notation for the details of software than boxes and lines.

As a computer science student in the 1980s, I read papers by Jon Bentley and others from Bell Labs extolling the virtues of domain-specific languages (DSLs). The best way to build many kinds of systems, they said, was to design simple, focused, special-purpose programming languages for the applications' domains, implement those languages, and build the systems using languages tailored to the tasks at hand. Language development tools such as yacc and lex were introduced as tools to facilitate developing such languages. And that group had remarkable success practicing what they preached, building groundbreaking tools such as pic, grap, make, sed, awk, and, of course, yacc and lex. All of those tools are still in use, in some form or other, decades later.

CLICK HERE to purchase this book now.

That style of development never really took off outside Bell Labs. Now, though, it's seeing a sudden resurgence. One of the most dramatic overnight success stories in software development is the Rails web framework, and much of Rails' strength comes from its inclusion of several distinct, small domain-specific languages focused on various aspects of web application development. Two related tools that have also garnered their share of attention, Capistrano (née SwitchTower) and Rake, are also based on those concepts. The implementation techniques are different from what the Bell Labs gang wrote about (and I'll talk about the new techniques next) but the concepts are the same. The idea of domain-specific languages seems to be one whose time has come.

Programming Techniques

I also see big changes in the programming techniques we use to build our software. This isn't entirely unrelated to the previous section; these techniques have strong effects on our design, and vice versa.

The most obvious change in this category is the move toward dynamically typed languages. Static languages of various stripes have dominated the software development for decades, from the loosely typed C and C++ to the stronger type systems of Pascal, Java, and C#. Most programmers have been taught that strong, static typing and compile-time analysis provide the only way to build robust, reliable systems.

That idea seemed to make sense, but it ignored the many solid systems built using dynamic languages. Additionally, during my ten years as a Java programmer I saw firsthand that strong typing is not a panacea; in fact, truly robust Java-based applications are rather rare.

Today, many developers have realized that a static type system is a two-edged sword. It does have some benefits, but it also has some costs. The advent of unit testing, more than anything else, has served to weaken static typing's appeal. Ruby, Python, and even JavaScript are growing more and more popular as developers discover the productivity advantages of dynamic typing.

For various reasons, many of these dynamically typed languages are dynamic in other ways as well. Your code can change (or augment) the way built-in facilities work, for example. This sounds similar to how aspect-oriented programming systems work, but the idea isn't new; in fact, aspect-oriented programming is a direct attempt to adapt older

CLICK HERE to purchase this book now.

dynamic language techniques to static languages, pioneered by some of the same people who built those dynamic language facilities.

Dynamic languages also blur the distinction between compile-time and run-time; in such languages, new code can easily be added to the system while it's running. Combined with other dynamic characteristics, this gives rise to a technique called *metaprogramming*, which is essentially extending your programming language from within. The practice of metaprogramming is a big part of the reason that domain-specific languages are making a comeback, because compared to building a stand-alone interpreter or compiler for a language, it's much, much easier to define domain-specific constructs in a language that supports metaprogramming. The new wave of DSLs gaining popularity in the Ruby community are built within Ruby itself as libraries.

The trend toward dynamically typed languages is both widespread and strong. Less obvious, though, is a resurgence of interest in functional programming and functional languages. Just in the past two years, two compelling applications have appeared that are written in Haskell: PUGS (an exploratory, prototype implementation of Perl 6) and Darcs (a powerful, decentralized revision control system). Other interesting systems have been written in Objective CAML (including MTASC, a free, blazingly fast ActionScript compiler). Those systems have prompted many programmers to learn those languages just so they can contribute to the projects, and the newcomers have been struck by the power and efficiency of functional languages.

Plus, interesting functional languages continue to appear. XQuery, the XML query language, is a functional language. This year's No Fluff, Just Stuff symposia will feature a talk from Ted Neward about Scala, a terrific functional language designed to work compatibly on both the JVM and the CLR. In fact, Ruby, Python, and JavaScript have strong functional characteristics and are often used in a functional style.

But it's not just a revival of old concepts in new languages; the old languages themselves are seeing a resurgence. A surprising number of people are discovering (or rediscovering) Lisp, due in part to the popular essays of Paul Graham. Also, the use of Smalltalk is growing again, sparked by some impressive systems such as Croquet and the brilliant Seaside web framework.

CLICK HERE to purchase this book now.

Language Implementations and Infrastructure

I remember vividly the reaction of many programmers when Java was released: “It’s interpreted! It’s garbage-collected! All array references are bounds-checked. You can’t use languages like that; they’re too slow!”

That was the common wisdom among most programmers for about three decades. To be efficient, languages had to be compiled, and programmers had to manage memory themselves.

It’s true that many Java-based systems perform poorly, and Java to this day has a reputation for sluggishness. And, for that matter, early implementations of Java really were excruciatingly slow. But that was mostly due to immature implementations that used pure interpretation of bytecodes and naive garbage collection strategies. In modern Java-based systems, though, the slowness is due not to those characteristics of the language implementation but to the libraries, frameworks, and platforms that have been built on top of Java. Java’s garbage collector performs extremely well, and many Java systems spend much less time managing memory than do equivalent C and C++ programs. As far as interpretation goes, the just-in-time compilers (*JITs*) and dynamic optimization technologies employed by most Java implementations produce very fast machine code at run-time.

Today we seem to have shed those earlier qualms about Java’s style of language implementation. Oh, there will always be situations where C is the most appropriate technology, but for most of the systems we build, VM-based or interpreted languages are fast enough, and features such as automatic memory management and array bounds checking really do help us build more robust systems—they’re much more helpful, in my opinion, than static typing.

For most systems, you get much more performance benefit from good architecture than you do from fast code. That’s a big part of the reason that typical Rails applications are at least as fast as their J2EE counterparts, even though Java typically benchmarks as about ten times faster than Ruby.

15.2 Why Now?

So far I’ve avoided a crucial question: if these older ways of doing things are so great, why didn’t they succeed at first? Lisp and Smalltalk had

CLICK HERE to purchase this book now.

their moments, as did bottom-up and iterative development, and the market chose a different direction. Why? And what has changed now to make the time right?

First, it's important to realize that there are more ways to fail than there are to succeed, and the problems weren't necessarily inherent to the technologies. Here are just a few ideas about what went wrong the first time and why things are different now.

The kinds of design techniques and processes that are returning to prominence were originally used by individuals and very small teams and began to show real weaknesses on more ambitious projects with larger teams. It was perfectly natural to try to inject more "discipline" into things with the use of phases, careful analysis and planning, inspections, and so on. But there are other forms of discipline besides top-down control, and we've learned from painful experience that software development is just too complicated a task to really benefit from central planning. Economies around the world, successful businesses, and even military organizations are pushing power and responsibility down toward the people in the trenches. The software development industry has learned the same lessons. Rigid control hasn't helped us avoid mistakes, so the industry is returning to basic skills, communication, and cooperation, supported this time by powerful tools and improved team practices.

Dynamic languages can be implemented very efficiently, but it's not *easy* to do so. Early implementations of dynamic languages were rather slow and required a lot of resources. It was much easier to build a C compiler that generated fast code than to build, say, a Smalltalk VM that performed similarly well. But implementation techniques have continued to advance, and the performance gap has shrunk dramatically. Not every dynamic or functional language has a state-of-the-art implementation, but we know from examples like Common Lisp, Squeak Smalltalk, and Haskell that it is possible for such languages to be blazingly fast.

As language implementations have been getting faster, our cost models have been changing. The first time around, slow CPUs and expensive memory meant that computing resources were not to be wasted, and dynamic languages looked like the wrong trade-off. Now, though, the balance has shifted. Sure, we still can't afford to be completely heedless of CPU and memory utilization, but fast machines and cheap memory mean that the sensible trade-off today is very different. Productivity is

CLICK HERE to purchase this book now.

much more valuable than it used to be in software development, and languages that save *our* time at the expense of some extra CPU cycles make a lot of sense.

As mentioned previously, we've begun using better development practices that help a lot. When projects don't use version control and don't have a disciplined approach to testing, the safety net offered by static typing seems to be quite valuable. We've learned, though, that we have to build our own safety nets that cover all aspects of the project, not just data types.

I could keep extending this list of reasons why things happened the way they did. The full list includes reasons such as primitive tools, fractured communities, weak development practices, incompatible competing dialects, expensive implementations, the lack of any free versions that developers could play with, and more.

Ultimately, though, we never really gave these tools and techniques a fair chance the first time. A world that hadn't yet really grasped the concept and power of "emergence" fled from iterative development as soon as it began showing flaws, not considering that the problem was a lack of supporting tools and practices rather than the technique itself. As far as languages are concerned, Lisp and Smalltalk were always on the fringes of the software field. COBOL, Fortran, C, and BASIC occupied the center. Occasionally we would adopt some of the ideas, such as object orientation, but we would try to fit them into the world we were used to, rather than taking them on their own terms. As a result, we missed some important subtleties, like (for example) the fact that object orientation doesn't exist in isolation but benefits greatly from other language characteristics such as blocks, dynamic typing, and automatic memory management.

So it's wrong to say "we tried that once and it failed." We're not going back to what *we* tried once; we're going back to what others had success with. The industry at large tried to go a different way, and at long last we've begun to realize that no matter how many new tools we throw at our problems, software development still isn't getting any easier. Maybe it's time to rethink the whole way we've been going. The people who really embraced Lisp and Smalltalk early on don't think those languages failed (except in terms of gaining broad acceptance). On the contrary, most of them that I know are either still finding ways to work with those technologies or else yearning for a return to the good old days.

CLICK HERE to purchase this book now.

15.3 More Past in Our Future

I predict that we'll see the increasingly wide adoption of dynamic languages, metaprogramming, and agile design and development practices over the next few years. In spite of many naysayers, momentum seems to be building in this direction.

I don't think it will stop with Ruby, Python, or any of the other new old languages that are gaining popularity. Although those languages borrow extensively from their progenitors, they stop short in some other ways. I love programming in Ruby, but occasionally I find myself needing some of the features of Smalltalk or Lisp that Ruby doesn't have—true macros, for instance, or the ability to easily pass multiple blocks to a single method (with appropriate cues as to their distinct roles). And don't get the idea that I'm an old Smalltalk or Lisp programmer! I come from a C, C++, and Java background. But I've recently begun to understand some of the subtle strengths of languages that I used to think were weird.

I'm not predicting a utopia, of course. These are trade-offs, and we'll give up some features to gain others. I can hear my skeptical friends asking now, "Sure, all that stuff is powerful, but is that the kind of power you want to give to the weakest programmers on your team?"

I bought into that argument for a while and argued that you should use truly powerful languages only with sharp, experienced teams. But then I started to notice something about the Java projects I was involved in: weak teams and weak programmers will go to great lengths to do the wrong thing. Time and again I've seen system designs that were not only inappropriate but also *much more difficult to build* than better designs would have been. I've just shaken my head in amazement—not at the inappropriate designs per se because good design is difficult but at the effort and tenacity it took to proceed with those designs in the face of the obstacles the teams had to overcome to build them.

What I've concluded is that you can't keep a weak team out of trouble by limiting the power of their tools. The way forward is not figuring out how to achieve acceptable results with weak teams; rather, it's understanding how to build strong teams and how to train programmers to be part of such teams. One place to start is with more emphasis on history. Our field is just barely 60 years old; there's no excuse for allowing programming students to remain ignorant of such recent history. Our history is rich with lessons that have been forgotten.

CLICK HERE to purchase this book now.

Here's an example. I'm developing with Rails right now, and Rails incorporates nice support for database migrations: little classes that encapsulate the changes to production databases (including both schema and data changes) required to move from one version or release of an application to another. It's a brilliant feature. But it has some problems, and most of them involve the way migrations mesh with the way we use version control. When we have a particular version of the software checked out, we are working with a set of files that describe the way the system looks at a given point in time. But migrations don't fit that model. There, in one version of your project, is a set of files that describe the whole history of the database schema, not just a point in time. It's like having a little version control system stored *within* your project, and that feels odd.

Typically we use version control to manage versions of program source code, and we use that source to build the system *from scratch* each time. Migrations, on the other hand, operate on persistent data; they don't have the luxury of starting from a clean slate.

In thinking about how to resolve some of these issues and perhaps fix them, I suddenly realized Smalltalk developers have dealt with similar issues for years. Smalltalk programs don't exist in source files on disk that are loaded, parsed, and compiled every time the system is run. Rather, they exist as objects—class objects, method objects, predefined and preconfigured instances, and other things—in a Smalltalk *image*, essentially a dump of Smalltalk's heap that is reloaded from disk and reconstituted just as it was the last time you were using it. In other words, Smalltalk programs exist as persistent objects.

So to learn how to solve my problems with migrations, it might help me to find out how Smalltalk developers do version management of their applications. I don't know the answer yet; that's a part of Smalltalk I'm not familiar with. But I'm going to find out.

There's more buried treasure there.

CLICK HERE to purchase this book now.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by working practitioners. The titles continue the well-known Pragmatic Programmer style, and continue to garner awards and rave reviews. As software development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help programmers and their managers stay on top of their game.

Visit Us Online

No Fluff Just Stuff 2006
`pragmaticprogrammer.com/titles/nfjs06`
Web home for this book, where you can find and submit errata, send us feedback, and find other related resources.

Register for Updates
`pragmaticprogrammer.com/updates`
Be notified when updates and new books become available.

Join the Community
`pragmaticprogrammer.com/community`
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
`pragmaticprogrammer.com/news`
Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragmaticprogrammer.com/titles/nfjs06.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	http://www.pragmaticprogrammer.com/catalog
Customer Service:	`orders@pragmaticprogrammer.com`
Non-English Versions:	`translations@pragmaticprogrammer.com`
Pragmatic Teaching:	`academic@pragmaticprogrammer.com`
Author Proposals:	`proposals@pragmaticprogrammer.com`