Premkumar Devanbu
Department of Computer Science,
University of California, Davis
devanbu@cs.ucdavis.edu

Desert Island Column (Journal of Automated Software Engineering)

I cannot imagine *why* I would be on a deserted island in the first place (with no high-speed digital lines, no coffee shops, and no express delivery service? Please!). Most likely some sort of catastrophic systems failure---perhaps a plane crash, or may be a malfunctioning GPS unit, or may be I just jumped out the window because the airline put me in an aisle seat and gave me something dreadful to eat. All because of a software failure, to be sure---pretty much nothing else fails these days, if you believe the news media. Some over-paid, over-worked, geek in Mountain View (or Limerick, or Bangalore) forgot to handle a specific action in a specific state in some protocol, and there I'd be, on this island.

So now we've established this: I'm stuck on this pitiful mound of dirt with time on my hands, all because of a software failure. After I've grown weary of cursing my fate (and those silly programmers) I would do the most sensible, mature, constructive thing possible under the circumstances---I would pull out my laptop, and start re-designing (from scratch, of course) the software running that benighted system whose failure brought me there. This eventuality would certainly drive my choice of reading material. So here is my first desert island book selection: *"Design Patterns: Elements of Reusable Object-Oriented Software"* (also known as the "Gang of Four", or GOF book) [1].

A strange sort of book, this is. Most books, even technical ones, have coherent narratives, with beginnings, middles, and ends. But not this one: no orderly march of chapters here, developing characters, thickening the plot, and moving steadily to the denouement. Each chapter in *this* book is on its own, with a special, unique story to tell. Not your average XML-Java-CORBA-WWW-mumble potboiler, by a long shot. But this book is a perennial best seller, with a stratospheric sales rank! It's also a *steady* seller, well ahead of the vapid buzzword-of-the-week titles that infest the technical sections of bookstores. What underlies this book's lasting appeal, in this impatient age of sound-bites and instant IPOs[1]? I'll tell you what: we software engineers (like all human artisans) crave beauty in our work. Creating beautiful designs is certainly pleasurable, but finding beauty in designs wrought by others delights no less. Writing almost 30 years ago, Weinberg devotes an entire chapter of his classic book [2] to the joys of reading programs. The GOF book offers us lovely illustrations of the software designer's art, and inspires and challenges us to add to this marvelous tradition. But let me move on to an example from the book; it's surely worth O (10^6) of my words.

Consider the following problem. I want to model an Airline passenger reservation. Reservations can be made, canceled, changed, confirmed, seated and retired. Reservations can have several options: vegetarian, or non-; smoker, or non-; frequent flier status; seating preference; traveling with/without infants, and so on. The actual semantics of each reservation operation depends on the applicable options, and must be executed diligently. Trust me, hell hath no fury like a frequent-flying, window-preferring vegetarian offered *bouef a la bourguignonne* on an aisle seat. We'd also like to be able to dynamically change/add options to a reservation. So what's the right design? What are the trade-offs?

[1] Initial Public Offering, as in stocks.

One might work through the following design choices:

- A single, giant, `Reservation` class with methods for each operation, has the siren call of putative simplicity. However, upon a closer look, we can see that methods are likely to grow complex, and difficult to maintain. If we wanted to add new option, we'd have to crawl through all existing methods, making changes. In addition, if reservations used only some of the options, we might be wasting a lot of resources, since we'd have to clone this bloated
- monster *in toto* for each reservation.

- How about several classes, one for each combination of options? This approach quickly looses appeal as we contemplate the combinatorial number of classes that need implementing. It gets downright revolting when we think about the difficulties of adding a new option. However, it does have the appeal of using only as much memory as was needed for a specific combination of options.

- Inheritance appears as a reasonable choice. Beginning with a base Reservation class, with no options, we can get single options as directly derived classes. Option combinations can be obtained by multiple inheritance. Alas, we still must endure a combinatorial number of classes; however, method implementations can simply invoke methods from base classes. Adding a new option, however, would remain tiresome.

- Templates? Interesting choice! Van Hilst and Notkin [3] offer a charming approach to this problem, that neatly sidesteps some of the difficulties noted above; however, like all the above choices, even this approach does not allow easy, dynamic addition of new options to an existing reservation.

So what's the GOF solution? Not so fast, buster! Find it yourself. I will tell you this: it's elegant, simple, fully dynamic, and it[2] will delight the artisan in you. Surely, there are trade-offs, but these are explained clearly and well.

And so it goes on. There are 35 basic design problems discussed in the book, with elegant solutions to them all. The GOF try to find commonalities among the 35, and group them in a logical fashion; in my view, this organizational effort is largely salutary. The book is simply a delightful collection of 35 design gems; each one is broadly applicable and lovely to behold. And that's about all there is to it. No narrative, no mystery, no resolution. Who needs all that, anyway? If I want more narrative in my life, I'll just go give my undergraduates a really, really tough final.

My second desert-island selection is Sedgewick's *Algorithms.* I can add little by way of style or substance to Michael Jackson's earlier paean to this indispensable volume; suffice to say, a well-thumbed and beloved copy sits on my shelf. Far cleverer people than most of us have worked out algorithms for almost any problem that we might come across, and Sedgewick presents these clearly with good explanations, useful illustrations, and (more recently) code. I've plundered a lot of code cheerfully from Sedgewick over the years, and you should too. Well, chosen, Michael.

[2] Psst! It's the DECORATOR pattern.

This brings to me my third and fourth desert island books, both on cryptography. My third choice is *Applied Cryptography*, by Bruce Schneier[4]. Let me warn you---it's a book for outsiders, not for experts (One might eventually need to supplement Schneier with the more authoritative *Handbook* [6]). It was Ron Rivest who famously defined cryptography as "Communication in the presence of adversaries". Cryptographers seek to build techniques for authentication, privacy etc., that resist attempts by adversaries to thwart the attainment of these goals. Why should a software geek care about such esoterica? Two reasons. First, because it's tremendous fun: cryptography deals intimately with heroes and villains; this never fails to get one's Jungian juices flowing (I'll tell you the second reason later). Schneier makes much of the drama inherent to his subject. All sorts of odd characters people his pages: besides Alice and Bob (the good guys) there's Eve the eavesdropper, Mallory the malicious adversary, Trent the trusted third party, Walter the warden, not forgetting Peggy the prover and Victor the verifier. The resulting plot mix is gripping: Schneier describes protocols, attacks, counter-attacks, counter-counter-attacks, and so on. But while these plot twists and turns, are engaging and instructive, some questions will dog the more observant reader: how do we know when a protocol is really secure? Can we tell how well a protocol achieves a goal? Must we always just lurch from protocol to unanticipated attack to counter-attack? Schneier will probably leave your appetite whetted for meatier fare. Hang on, here it comes!

My final book is not really a book, but a collection of papers published by Bellare, Rogaway and their colleagues on modern cryptographic techniques. Bellare and Rogaway are lucid and rigorous. They carefully define cryptographic goals, and describe analytic techniques for precisely characterizing the effectiveness of proposed solutions. The central weapon in their arsenal is reduction (in the style of NP-completeness reduction) between adversaries. Bellare, Rogaway and their colleagues teach us how we might achieve security goals with engineering precision, rather than with guesswork. Their papers are beautifully written, and richly rewarding. I won't pretend that this is easy going---but the style of thinking that one learns by studying these papers will pay rich dividends, I believe. Yes, it's mathematics, but it's mathematics with the eternal grand themes: heroes, villains, supporting characters and good and bad endings. Theory, but with a bit of George Lucas thrown in. Check it out: Bellare [5] is a good place to begin your adventures in rigorous modern cryptography.

Which brings me to the second reason why software geeks should care about cryptography: because almost all software engineering these days happens in the presence of adversaries. Consider that software can be developed by adversaries (mobile code), used by adversaries (users trying to violate copy-protection in video games) or even tested by adversaries (reverse-engineering by competitors). So now we have something new and different: "software engineering in the presence of adversaries!" A bigger rush than watching WWF[3] on the telly, any day! In this new era of developing and operating software in the presence of adversaries, we all need to be aware of cryptographic techniques and more importantly, the adversarial reasoning inherent in modern cryptography. Various proposals have recently surfaced in software engineering and programming language conferences that propose approaches to such problems as software copy protection, code obfuscation, and software watermarking. The techniques are intriguing----however in some cases, careful adversarial reasoning suggests attacks that would prevent the desired goals from being attained. I think that this line of work is rife with opportunity, and will pay rich dividends, as more software engineers become adept at adversarial reasoning. I believe that the next several years will bring some rigorous, exciting new results in this area.

[3] World Wrestling Federation.

That’s it. Now can I get off this forgotten spit of dirt, and return to Northern California? I need a Latte.

[1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Grady Booch , *Design Patterns : Elements of Reusable Object-Oriented Software* Addison-Wesley Pub Co; ISBN: 0201633612 ;
[2] Gerald M. Weinberg, *The Psychology of Computer Programming*, Dorset House; ISBN: 0932633420
[3] Michael VanHilst and David Notkin. “Using Role Components to Implement Collaboration-Based Designs”. *OOPSLA-96* (October 1996).
[4] Bruce Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, John Wiley & Sons; ISBN: 0471117099
[5] Mihir Bellare. “Practice-oriented provable security”. In G. Davida E. Okamoto and M. Mambo, editors, *Proceedings of First International Workshop on Information Security (ISW 97)*, volume1396 of LNCS. Springer Verlag, 1997.
[6] Alfred J. Menezes, Paul C. Van Oorschot, Scott A. Vanstone (Editors), *Handbook of Applied Cryptography* CRC Press; ISBN: 0849385237.
[7] Robert Sedgewick, *Algorithms in C++: Fundamentals, Data Structures, Sorting, Searching* Addison-Wesley Pub Co; ISBN: 0201350882