GAMEARCHITECT.NET
MUSINGS ON GAME ENGINE STRUCTURE AND DESIGN

Home

What It's All About

Who Is This Guy?

The List

Complete Archive

Personal

RSS Feed

People

Search

## Software Is Hard

*By Kyle Wilson*
*Sunday, August 19, 2007*

"Software is hard," reads the quote from Donald Knuth that opens Scott Rosenberg's *Dreaming in Code*. The 400 pages that follow examine why: Why is software in a never-ending state of crisis? Why do most projects end up horribly over-budget or cancelled or both? Why can't we ship code without bugs? Why, everyone asks, can't we build software the same way we build bridges?

The framing story for Rosenberg's investigation is the Open Source Applications Foundation's Chandler project. Chandler begins life in the spring of 2001 as an alternative to Microsoft Exchange:

> [Exchange] was the least unbearable of available options for small-group scheduling. Still, it seemed crazy, far more powerful than the office needed. Costlier, too: You had to dedicate a computer as the server, you had to pay for a Windows license for that server, you had to license the Exchange software itself.... Before you knew it, you were spending thousands of dollars just to keep a handful of calendars in sync.

OSAF is a non-profit venture funded by Mitch Kapor, who made enough money as the founder of Lotus that he can afford to spend a few million dollars out-of-pocket on software that may change the world. Kapor's munificence makes Chandler immune to the budget pressures that force most software applications to ship before they're ready, but it also means that there's no real pressure for Chandler to ever ship at all. "We're not operating with the mythology of the Silicon Valley death march, with the deadline to ship a product and get revenue, where the product quality goes out the window," Mitch Kapor says in *Dreaming in Code*.

As a counterpoint to OSAF, Rosenberg presents 37 Signals, which has made a business out of small web-based applications. Jason Fried, the president and founder of 37 Signals, says, "Constraints are key to building a great product. They're what makes creativity happen. If someone said you have all the money in the world to build whatever you want, it would probably never be released. Give me just a month!"

Chandler's saga is an uncomfortable story for me because it's powerfully reminiscent of development at Cyan back in 1999, when we started on what eventually became Myst Online. Mitch Kapor created Lotus 1-2-3, a killer app that brought him surprise fame and fortune at an early age. Cyan CEO Rand Miller created Myst, a killer game that brought him surprise fame and fortune at an early age. Kapor ended up a lot richer, but they both made enough from their early success to leave software development behind and live comfortably for the rest of their lives. Instead, both men spent millions of dollars from their personal fortunes trying to create something even greater. Rosenberg asks Andy Hertzfeld, an early OSAF volunteer, if Kapor's accomplishments aren't enough:

> "No, he still needs more glory. We all need more glory as designers--to show we can design another great thing. Everybody who has a first success, especially when it's young, wonders: Was it luck, or was it skill? Well, it's a little of both. If you can do another really great one, it shows the world something."

I don't want to disparage Mitch Kapor or Rand Miller. I know Rand is, and I believe Kapor to be, genuinely good, likeable, and smart. But the determination to make something insanely great is a kind of hubris, the flip side of which is a terrible fear of making something merely ordinary. Myst Online started out as a single-player game of modest scope called D'ni In Real Time, or DIRT. By the time I left Cyan, it was planned to be an MMO with a million subscribers, a driving application that would convince casual computer users to sign up for broadband Internet connections. Chandler starts out as a free replacement for Exchange that's simple enough for casual users. But that's not enough. That would be merely ordinary.

The story of Chandler's ensuing development is, for anyone who's spent time developing software, sadly predictable. There comes a point halfway through relating Chandler's saga, where Rosenberg writes, "By now, I know, any software developer reading this volume has likely thrown it across the room in despair, thinking, 'Stop the madness! They're making every mistake in the book!'"

It's true. Here are a few:

- Peer-to-peer architecture. From the beginning, Chandler is conceived as a peer-to-peer application, for little obvious reason except that P2P apps were considered to be a new and nifty thing at the time the project was starting up. This greatly complicates its communication and security requirements.
- Simultaneous cross-platform development. Because it's an open-source project staffed by Linux users and ex-Apple employees, Chandler is developed simultaneously on Windows, Linux and the Mac. As anyone who's done cross-platform development knows, this necessitates three times the build infrastructure and means that at least one of the builds will always be broken due to compiler and API differences. Any for-profit software company would develop first for the 93% of the market that runs Windows and would port to other platforms later if success on the PC showed doing so to be a good investment.
- Choice of language. OSAF chooses to implement Chandler in Python, because members of the original team found that they could rapidly prototype in Python. But experienced Python programmers are harder to find than experienced C++/C#/Java developers, so the company ends up mostly hiring people who'll learn the language on the job. This makes everybody less productive and, worse, means that they write worse code than an experienced Python developer would. Phillip Eby, a Python guru brought onto the Chandler team three years into development writes, "So, the sad thing is that these poor folks worked much, much harder than they needed to, in order to produce much more code than they needed to write, that then performs much more slowly than the equivalent idiomatic Python would."
- Uncontrolled feature creep. Kapor originally conceives of Chandler as a complement to Microsoft Outlook targeted at "'information-intensive' individual users and small companies." But everyone on the Chandler team has a different vision of what that means, and in the absence of pressure to ship, Chandler becomes the union of everyone's ideas. Instead of borrowing from successful similar products like Outlook/Exchange, the Chandler team is determined to invent something entirely new from first principles. They want to support user plug-ins and scripting, built-in encryption, storage of messages in multiple folders and infinitely customizable user views. As the project goes on, the simple replacement for Outlook/Exchange grows more and more complicated.
- No decision is ever final. Time and again, the Chandler team hashes out compromises on complex issues, only to hit reset when someone new joins the project with new ideas or when it turns out that someone wasn't really satisfied with the compromise. They revisit their choice of database layer, whether to use Mozilla's UI primitives or wxWidgets, and the whole design of their user interface. Eventually, the peer-to-peer architecture is abandoned in favor of a client-server architecture. Three years into the project, the team is still debating whether to turn Chandler into a web-based application like Google Docs & Spreadsheets.

Chandler's story, you'll recall, starts in the spring of 2001. As I write this, in the fall of 2007, OSAF is trying to get version 0.7 of Chandler out the door. And yet, OSAF's employees are all smart people, and most of them have a great depth of experience in software development. How did they end up here?

---

I was talking to my father a few weeks ago about my work, and he asked, "When you're working on a game, how do you know when you're done?"

The real answer is economic. Eventually the marginal cost of each additional bug fixed or feature added exceeds the marginal utility--in additional sales--that a publisher will realize when they burn the game to DVD and put it on shelves. The real world, with its contractual obligations and imperfect knowledge, isn't quite so efficient, but that's the gist of it.

"But how do you know when that will be?"

Well... you don't really.

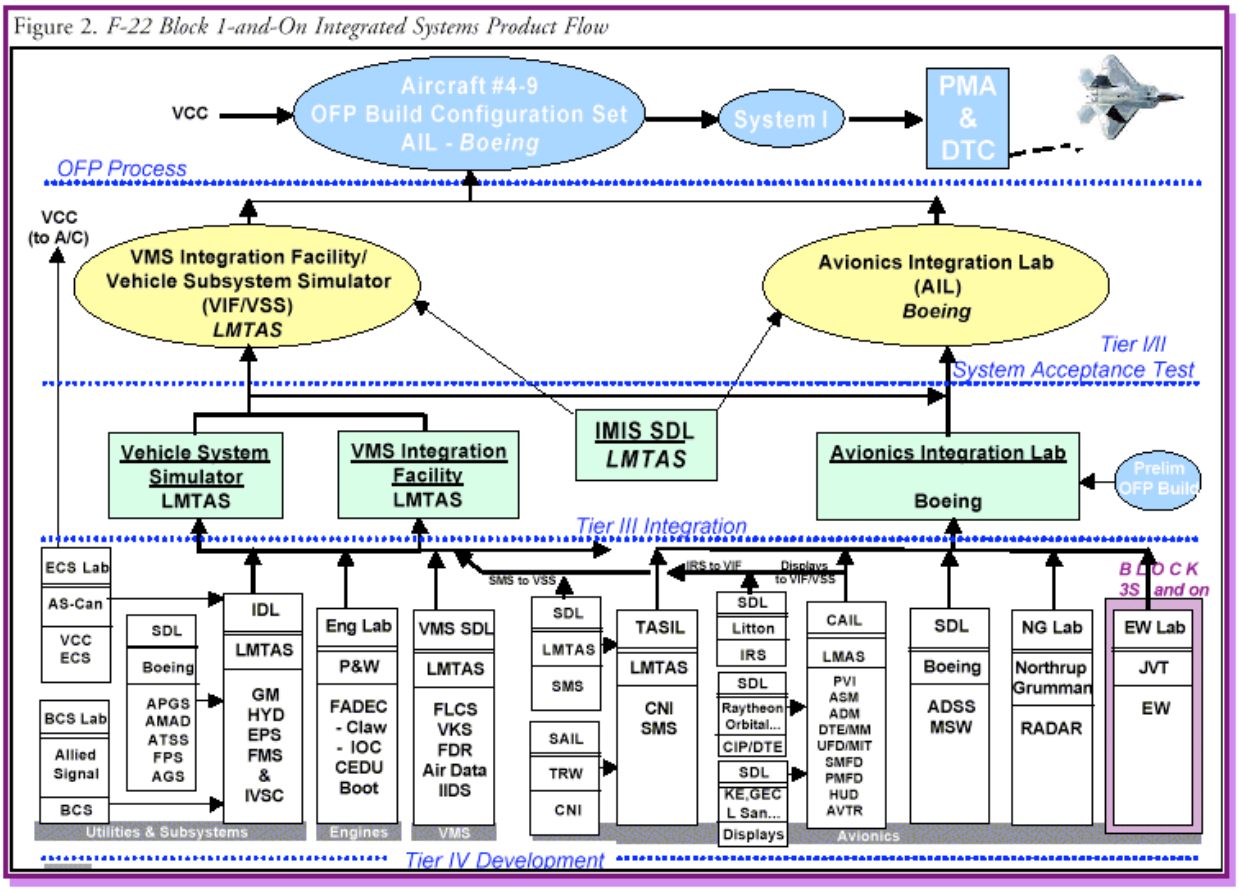
"Can't you count lines of code," Dad asked, "and compare that to how big your last game was when it was done?"

MechAssault 2 was about 500,000 lines of code. Fracture passed that point last August and has been steaming ahead like a freight train since, adding 10,000 to 40,000 lines a month. Games for this console generation are *big*. But we didn't know that when we started writing them. We're learning as we go along.

Speaking of lines of code, my friend Rebecca, who's a Pentagon reporter, has sat through countless briefings about military software projects. She's bemused at the briefer's fondness for citing the number of lines of code in the systems they're developing.

There's not really any good way to measure the complexity or scale of a piece of software, I told her. Lines of code is a lousy metric, but it's the only one we've got. Fracture is 825,000 lines of code right now, and should be over a million by the time we ship. The Feynman report on the Challenger disaster says the shuttle was running 250,000 lines of code. An F-22 fighter jet runs 1.7 million lines of code. What does this tell us about the relative complexity of these programs? Not a whole lot.

Consider a practical example: The first delivery of "Block 1" F-22 software happened in 1999, after eight years of development. It was about 750,000 lines of code. It was developed in partnership by about twenty independent teams, as shown in the helpful diagram below:




Fracture, by comparison, has taken two years to produce as much code. The average number of programmers during that time has been about 15.

I believe that the following statement is an axiom of software development:

*It is impossible, by examining any significant piece of completed code, to determine within a factor of two how many man-hours it took to produce that code.*

And the corollary:

*If you can't tell how long a piece of code would take when you have the finished product available, what chance do you think you have before the first line of code is written?*

Talking about a software development schedule more than a year out is like talking about where we go after we die. Everyone has some idea where we'll end up, but those ideas differ wildly, and there's a lack of solid evidence to support any of them.

If they'd known in advance the costs of all the decisions they made, the Chandler team--like any software development team--certainly would have made some decisions differently. But software engineers don't even have useful metrics for measuring the complexity of code after it's been written, so how can we compare the costs of features that haven't been implemented or alternatives that haven't been decided? Software scheduling is an NP-complete bin packing problem where the sizes of the bins are hidden!

After a half-century of software scheduling, after counting function points and lines of code and switching from waterfall development to spiral to agile methods, the most effective scientific tools we have for estimating software development time are:

- The Ninety-Ninety Rule : "The first 90% of the code accounts for the first 90% of the development time. The remaining 10% of the code accounts for the other 90% of the development time."
- Hofstadter's Law: "It always takes longer than you expect, even when you take into account Hofstadter's law."

---

In theory, I suppose, the complexity of a well-structured program should be O(n), where n is the number of lines of code, if each line of code is tightly coupled only to the line that precedes it and the line that follows it. A poorly-structured program would be $O(n^2)$, with any line of code possibly calling into any other line of code in the program. If n is on the order of a million, the gap between O(n) and $O(n^2)$ is staggering. So why do we count lines of code at all? Because it gives us the comforting illusion that we can measure that which is essentially unmeasurable.

When it's done, Fracture will be merely the *length* of the Encyclopedia Britannica. But it will be a lot more *complex*. Encyclopedias are well-designed software. They're loosely-coupled--entries tend to not make calls into other entries--and strongly cohesive--an entry about one subject tends to hold together. But there's a lot of detail in a modern game that pushes game complexity further away from being O(n). The developer is God of a virtual world, and it's hard work being God. Not a sparrow falls unless someone writes the code to make it happen, and that code is likely to touch the physics system, the sparrow AI, the effects systems (if you want the sparrow to go splat)--and that's not even getting into the dependencies on art and design!

---

> In a 2004 talk at the Pentagon's annual software development conference in Salt Lake City, Jon Ogg, director of engineering and technical management at the Air Force Materiel Command, laid out what he called the "software divergence dilemma" that the military faces today. In the past fifty years, the amount of code in a typical military system has increased a hundredfold.... Meanwhile, in that same span of time, the average productivity of programmers has only doubled.

Let's look back at the F-22. Why did it take dozens of software engineers working on the F-22 avionics code four times as long to produce 750,000 lines of code as it's taken the Fracture team? Is the average programmer at Day 1 Studios a hundred times as productive as the average programmer at Boeing? Of course not. The F-22's avionics suite is life-critical software. It has strong interdependencies on changing hardware. Development is saddled with all sorts of CMM-5 overhead. The hardware it runs on is nearly twenty years old now, which increases iteration times (the F-22 runs on a VAX/VMS system like the college mainframe when I started at Davidson, back in 1990).

The problems are different.

But the nature of software is that the problems are *always* different. You never have to solve the exact problem that someone's solved before, because if software already existed that solved your need, you wouldn't have to write it. Writing software is expensive. Copying software is cheap.

Scott Rosenberg coins this as Rosenberg's Law: *Software is easy to make, except when you want it to do something new*. The corollary is, *The only software that's worth making is software that does something new*.

---

Every software engineer has a low opinion of the way we develop software. Even the term "software engineering," Rosenberg writes, is a statement of hope, not fact. He quotes the 1968 NATO Software Engineering Conference that coined the term: "We undoubtedly produce software by backward techniques." "We build systems like the Wright brothers built airplanes--build the whole thing, push it off the cliff, let it crash, and start over again." Certainly statements that could still be made forty years later.

Here's Alan Kay, the father of Smalltalk: "Most software today is very much like an Egyptian pyramid with millions of bricks piled on top of each other, with no structural integrity, but just done by brute force and thousands of slaves."

And yet, in a typically optimistic programmer triumph of hope over experience, everyone believes that someone, somewhere, is building software the *right* way, and that we too could make software development predictable and well-behaved if only we possessed sufficient will and discipline. Rosenberg mentions a scare book called *The Software Conspiracy* centered on the premise that we produce buggy software because of "self-indulgent programmers, shortsighted companies, and a public too willing to tolerate shoddiness and bugs." Game developers, particularly, believe that our software failures are punishement for our own sins. "They'd never let us get away with practices like this in the real world," people say.

The story of Chandler's development argues otherwise.

I believe, rather, that game development is the hardest kind of software development there is. Game programmers routinely deal with more uncertain requirements and shorter schedules than any "real world" project, and they're expected to ship a compelling original product without patching or iteration. Joel Spolsky writes in "Five Worlds",

"Games have the same quality requirements as embedded software and an incredible financial imperative to get it right the first time. Shrinkwrap developers have the luxury of knowing that if 1.0 doesn't meet people's needs and doesn't sell, maybe 2.0 will."

But maybe Chandler's just not a good real world example. Maybe Chandler just didn't have enough oversight. So consider the FBI's Virtual Case File system. In 2000, the FBI hired SAIC--purportedly a CMM Level 5, ISO 9000-compliant shop exercising the best software development practices available--to implement a far-reaching upgrade to its existing IT and recordkeeping infrastructure. Instead, according to Harry Goldstein's "Who Killed the Virtual Case File" in IEEE Spectrum, after five years SAIC "delivered 700,000 lines of code so bug-ridden and functionally off target that this past April, the bureau had to scrap the US $170 million project, including $105 million worth of unusable code." That's five years and over a $100 million, driven by theoretically perfect software engineering practice and constant government oversight, to produce less code than we've written for Fracture--and it ended up being code that didn't do anything anyone wanted. Why? Goldstein cites "poorly defined and slowly evolving design requirements; overly ambitious schedules; and the lack of a plan to guide hardware purchases, network deployments, and software development for the bureau."

Or, as Fred Brooks writes in "No Silver Bullet": "The hardest single part of building a software system is deciding precisely what to build."

---

By the time Rosenberg leaves OSAF at the end of 2005, many of Chandler's early developers have drifted away. One leaves to become a woman. Another quits to start his own company. Another goes back to school. Others, who have enough wealth from the Internet boom days that they don't really need a salary anymore, just move on to other open-source projects.

The ones who remain are the quiet professionals who are in it for the long haul. They're willing to cut features and to scale back grandiose dreams to make them attainable plans. And they're willing to push back milestones when the product isn't ready, because they know that finding the right equilibrium between cuts and slippages is the only way to ship a product worth having.

In short, they're the best hope that Chandler has of someday emerging from open-source eternal beta hell precisely because they recognize that software is hard.

---

> Weinberg's Second Law: "If builders built houses the way programmers built programs, the first woodpecker to come along would destroy civilization."

Why is this so? What makes the virtual world so much more brittle and unpredictable than the physical world? Why can't we build software the way we build bridges?

Part of the problem, clearly, is that every time we build software we're creating something fundamentally new. There are, Wikipedia tells me, six main types of bridges. Once you know how to build a bridge of a particular type, you're just iterating on a theme. The principles behind a suspension bridge, for example, are well understood--if not by me, then at least by the people who build them! But like Rosenberg's Law says, once Epic's written Unreal, they never have to create a first-person shooter engine again. Instead, they just need to keep adding new features. Day 1 may need to write a first-person shooter engine, but only because we've never done that before. This is why programmers always want to rewrite everything. Most shipping software is the equivalent of a novelist's first draft.

It's true, too, that construction is less predictable than many would like to believe. I've been a homeowner suffering from "buggy" construction (faulty window frame installation, rotten walls). I've watched the resolution of the bug stretch to twice the contractor's original estimate. Rosenberg cites the San Francisco/Oakland Bay Bridge as another construction project that's running well over scheduled time and budget.

The difference is that the overruns on a physical construction project are bounded. You never get to the point where you have to hammer in a nail and discover that the nail will take an estimated six months of research and development, with a high level of uncertainty. But software is fractal in complexity. If you're doing top-down design, you produce a specification that stops at some level of granularity. And you always risk discovering, come implementation time, that the module or class that was the lowest level of your specification hides untold worlds of complexity that will take as much development effort as you'd budgeted for the rest of the project combined. The only way to avoid that is to have your design go all the way down to specifying individual lines of code, in which case you aren't designing at all, you're just programming.

Fred Brooks said it twenty years ago in "No Silver Bullet" better than I can today: "The complexity of software is an essential property, not an accidental one. Hence, descriptions of a software entity that abstract away its complexity often abstract away its essence."

Software construction is the most complex endeavor ever undertaken by mankind. It makes building things like cathedrals and space shuttles look like child's play, and it strains our little monkey brains to the utmost. If we're ever going to make building software any easier, we're going to have to build a machine that's smarter than we are. At the moment, even weakly superhuman AI is looking a long way off. And when we get it, the smart money says that it'll be late and over budget.

*Any opinions expressed herein are in no way representative of those of my employers.*

---

Home