"So You've Decided To Develop Your Own Software!?"

Lessons Learned from A Decade of Broken Software Projects

Jim Henry and Chris Herot

Sag Harbor Group

February 2002

"Computers will never replace business people, because business people don't know what they are doing."

-- Prof. Carl Kaysen,
Institute for Advanced Studies, Princeton, early 1960s

"Know the problem before choosing the solution…."

-- From the UNIX SysAdmin aphorism collection

Introduction

So you've decided to develop your own software! Terrific! Sag Harbor Group team members have been down this path quite a few times in the last decade. We are happy to help you design and manage the project so that your odds of success can be greater, help you explore alternative development plans and sources of technology, and generally share your pain. But before you embark on this road you should pay careful attention to the following hard-won lessons.

Just as in the case of buying a company,[1] the very first favor you can do for yourself is to think "a hundred times again," as the Chinese say. According to the extensive literature that has accumulated on software development, and our own experience,[2] this is another one of modern capitalism's great trap doors -- right up there with corporate mergers, lending money to developing countries, political corruption, permitting CEOs to design their own compensation packages, and relying for objective valuations on "independent" auditors or Wall Street analysts whose firms are also seeking lucrative consulting or I-banking assignments.

The fact is that almost *eighty percent* of software projects take at least *twice* as long as their original timelines, and a solid 25-30 percent of them never see the finish line at all. Obviously this is one big average that lumps together tiny projects tackling incremental problems within well-defined boundaries and massive "boil the ocean"/ "change the very nature of our being" projects on the frontiers of civilization.

In general, however, the task of "coding" is still very much a primitive art. In particular, the task of managing large teams of people who are trying to turn out high-quality "systems" software is like herding cats on a treadmill.

Nevertheless, as usual, some things can be said. As dismal as the track record of many large-scale software projects has been, the even more disturbing fact is that many of these disasters were man-made and entirely preventable. The following are some guidelines that might help you avoid repeating some of these disasters.

[1] See the SHG White Paper, "So You've Decided to Buy a Company!" (February 2002), on our Web site at www.sagharbor.com.

[2] SHG senior consultants Eileen McGinnis, Chris Herot, and Andres Carvallo have all had significant hands-on experience in software development management at Sun Microsystems, IBM/Lotus, Microsoft, ATT Labs, and HP, and Jim Henry was VP Strategy for Lotus Development during the launch of Lotus Notes™ and a senior manager at Geosphere, an AT&T effort to develop a robust Internet services platform in the mid-1990s, "just slightly ahead of its time."

SHG's Top Ten Rules of the Road – Software Development

1. Field of Dreams -- Just Because It CAN Be Done, Doesn't Mean That There's a Market for It or That Competitors Won't Get There First

This rule is first for a very good reason – if we had to choose just one thing for software developers and their managers and funders to get serious about, this would be it. This is because, empirically, violating it is responsible for *more than two thirds of all failed, delayed, or cancelled software projects..*

That is not to say that we don't sympathize with all the heroic developers out there who have devoted years of their lives to internal visions, without so much as a "what'dya think?" from real live customers.

Non-Linear Exceptions. Nor do we deny that there are many important exceptions to the "customers know best" rule. These include most of the world's greatest software innovations. No focus group on earth could have anticipated innovations like Unix, browser-based Internet access, Lotus Notes,™ graphical user interfaces, or PC spreadsheets. We are reminded of the (apocryphal?) tale told about the Boston consulting firm ADL, which was asked in 1956 to estimate the potential US market for Xerox™ machines by Xerox's inventor, Chester F. Carlson. It responded by counting up the number of carbon copies consumed by US typists every year, and replied -- "Approximately six machines."

Industry Maturation. So it is not easy to estimate the potential demand for truly transcendent innovations. But these are rare -- indeed, perhaps increasingly so. We live in an era when it is increasingly difficult for fresh software to be *sui generis* – there is simply too much other software that it has to be compatible with. There is a huge base of "legacy" applications and systems, standards, established security procedures, and ingrained customer habits that have to be taken into account.

The Importance of Understanding Customer Requirements. All this implies that, right up front, it is often invaluable – and never a complete waste of time -- to allow for a hefty dose of "customer inputs" on new project requirements. This ought to occur well before a proposed project gets too far down the road – long before coders and their managers acquire vested interests in defending what they're built or *promised* to build.

Experienced software engineers understand that by far the most important step in project management is this initial customer requirements analysis. Indeed, it is far better to design simple solutions that meet basic user requirements, put those in the field as quickly as possible, and get feedback from actual users than to spend months in a customer-free environment, searching for the "next big thing."

The Indirect Costs of Change. Another route to failure is to expect customers to make too many *indirect* behavioral changes to accommodate a new product. For example, in the late 1980s IBM spent millions trying to get its customers to upgrade from DOS to OS/2. It was, in effect, requiring them to buy an entire new suite of applications software – several of which didn't exist -- just to accommodate IBM's attempt to repossess the OS market. It also assumed that consumers had access to a complete user infrastructure, like compatible printer drivers, that were not available.

Microsoft, in contrast, offered its PC users a less costly upgrade path, with early versions of Windows that supported all the old DOS applications. Along the way, it required software developers who wanted Windows certification to support NT, its new OS. This meant that once NT applications became available, Microsoft could migrate users more smoothly and abandon the older "Win-DOZE" versions.

Who are The "Customers," Anyway? The relevant "customers," for purposes of such requirements analysis, are quite diverse. They include internal business units that are expected to pay for a project and consume what it produces; channel partners who have to sell it; technology vendors that need to produce complementary products; systems integrators who have to install it; and support groups that have to train its users and answer their questions. Ideally, all these diverse audiences should be consulted before design goals are set in stone.

Toward Routine, Incremental Progress. Even with clear customer requirements in place, we should not be too sanguine about achieving such goals, let alone world-shaking innovation, or about leap-frogging our competitors in a sustainable way. These are not the halcyon days of the global software industry's infancy. In the wake of the "Internet bust," many observers have detected a pronounced slowdown in the rate of innovation, a shift away from the "step-function" introduction of whole new categories like network communications, groupware, and desktop productivity applications that characterized the 1970s and 1980s. We have entered a "post-Internet era" dominated by prosaic, incremental refinement of existing applications and services, focused on "small change" issues like increased reliability and reduced cost, plus extensions to new platforms like wireless and broadband, rather than fundamental change.

In other words, the game is no longer so much about realizing bold new visions, as it is about methodically working out all the kinks in visions that for the most part are by now rather middle-aged.

Software's Mode of Production. Indeed, the basic social organization of software engineering – at least among "closed source companies" that are seeking profits by producing (or privatizing) and selling secret code – has long since shifted from reliance on gifted individuals who work alone or in small teams, motivated by the sheer satisfaction of writing good code. It is now a much more industrial, routinized, Beni-Hana affair, with large projects that demand scores of coders, debuggers, and testers working in continuous rotations to refine GUIs, swat bugs, and make test cases.

Really Good Code is Boring. All this is not very inspiring to the industry's most creative minds – some of whom now wander from startup to startup in a Quixote-like quest for the days of yore. But it may be very good news for software consumers. There seems to be a direct correlation between the degree of boredom involved in writing modern code, and the degree to which that code ends up in easy-to-use, valuable applications for real customers. This is one one reason why so much "open source" code is fragile and hard to use, even if more novel.

Alternative Sources of Development. Nor are software developers any longer such rare birds, working under cottage industry conditions. Their chosen profession is now heavily populated, and many work for giant firms. As of 2000 there were more than 1.1 million software programmers in the US alone, including 375,000 application developers and 265,000 systems developers. India, the next in line, has more than 600,000 developers, available for "offshore" contract work at a fraction of the cost of Western programmers. Russia, Israel, Ireland, Brazil, Korea, Finland, and South Africa also have vibrant software engineering communities, and, indeed, many developing countries with low labor costs and educated labor forces are pushing software engineering services as an export – a new digital form of Third World plantation, with the "sugar" and "tea" exported by way of the Internet, rather than by ship or rail.

Three Key Questions. All this implies that the proponents of any new software development effort should be asked to explain quite clearly (1) "Why customers will want it when we deliver it?"; (2) "Why our solution will be better than what our competitors will offer?"; (3) "Why we should build it ourselves, rather than outsourcing its development to starving developers abroad who may cost one-fourth as much?"

2. Superior Technology is Nice, Insufficient, and Often Unnecessary

Microsoft is the poster child for this proposition. Time and again, it has succeeded beyond its wildest dreams on the basis of "feature-bug" software that many technical experts regard as mediocre, but which hit the market at just the right time – neither too early nor too late. This ability to deliver products that are just barely "good enough," but work with the other MS products, shows that persistance and good marketing can trump sheer technical excellence or "first mover advantage" any day of the week.

With certain rare exceptions, Microsoft has proved this time and again over its thirty-year history. It has mastered the art of delivering products that provide the acceptable minimum of functionality and quality. It is an industry cliché that version 1.0 of a Microsoft product almost never works very well. But by being satisfied with less-than-perfect software, the company gets invaluable feedback from its guinea-pig customers. This helps it work out (some of the) bugs in the next release. Even when it enters markets late with inferior products, through relentless efforts it has often been able to take command of markets while its competitors are still busy reaching for perfection.

Sometimes this has been possible because the choice of platform and features is more important than elegant implementation. By concentrating on the admittedly imperfect "Win-DOZE" operating system, for example, Microsoft avoided the "matrix of pain" that so many other software companies have struggled with. This refers to the need to invest in maintaining applications across multiple platforms. There is also the platform paralysis of having to wait patiently for the next version of an operating system whose development is under someone else's' control.

Microsoft is just one example of the "not too bad" school of software development. AOL is another -- technology gurus cringe when they see the billions in market cap that have been generated by its simple-minded Internet overlay. Others argue that 25 million regular AOL subscribers and 70 million AOL Instant Messaging users can't be wrong.

3. It Is Possible to Get *Too Far* Ahead/ Go It Alone *Too Long*

The other side of "good enough" are the many examples of truly innovative software companies that have lost out to more prosaic rivals, by trying too hard to be innovative. One good example is Lotus, which in 1985 was more or less the same size as Microsoft. Indeed, that year there were unsuccessful merger talks between Lotus' Mitch Kapor and Microsoft's Bill Gates!

In the early 1980s Lotus had of course pioneered the PC spreadsheet, an application that actually deserved most of the credit for helping IBM's PC takeoff in 1981, as well as for generating millions of dollars in sales for young Microsoft's DOS operating system, which shipped inside every PC.

But Lotus' leaders were never quite satisfied with simply leading a "PC spreadsheet" company, or even leading the dominant "desktop applications" company. That was simply too boring – software was supposed to have meaning. So for more than a decade, from the early 1980s on, Lotus invested heavily in pioneering whole new categories of software applications -- personal information managers, groupware (Lotus Notes™). an oddity for Steve Job's Next machine called a "3D spreadsheet," and a heavy-duty word processing program for legal offices called "Manuscript."

Of course Lotus also faced a fundamental competitive problem. Throughout this period, Microsoft dominated the PC operating system, which became the key playing field for everyone's applications. IBM had originally farmed the DOS operating system out to Gates' tiny firm in 1980, partly because CPM's Gary Kildahl was too busy to take its calls, and partly to avoid being charged with monopolizing the whole PC industry. (Gates has long since has displaced IBM in that dock.) When Gates decided to enter the applications market and compete directly with Lotus, the fact that he was also able to fund this with his high-margin DOS annuity certainly did not help Lotus.

It is also undeniable that when IBM finally acquired Lotus for $3.5 billion and put it out of its misery in 1994, one key attraction was Lotus Notes™, a direct consequence of Lotus' innovation. Still, one wonders whether or not the history of the software industry might have been different if Lotus had been slightly more focused, or if it – rather than MS – had been selected by IBM to develop OS/2 , or if Steven Jobs had skipped Next and opened up Apple to licensing….Or if Lotus had just been willing to merge with Microsoft in 1985!

Lotus is not the only case of a software company that traded off economic success for innovation. Apple is another, AT&T Labs (which produced Unix, the C language, and many Internet innovations, and still holds the patent for overlapping windows) another, and Sun's Java group may turn out to be a third. In all of these were institutions where, ironically, brilliant developers may have been given far too much influence – perhaps, in turn, because the people who headed these institutions were **not** software developers themselves. Gates, in contrast, being a pretty good developer himself, may have better understood the limitations of his own art, and the intrinsic value of lying in this business.

4. Great Software Takes 10 Years and 6 Releases

There are few examples of successful software applications, operating systems, or systems of any complexity that have not taken at least ten years to achieve stability and reliability. This is partly due to the fact that few development teams spend enough time specifying user requirements, as we noted above. Even if a team does try to get the requirements right, it makes mistakes. So it needs to allow for the product's evolution, incorporate user experience, use rapid prototyping, and iterate until the product stabilizes.

5. Put Someone in Charge Who Has Done This (e.g., Failed) Before

Software is either trivial or it is late. Harvey Golub, the former CEO of American Express who knows a great deal about software development, has tried to make this more precise -- "Golub's Rule" says that software projects always take *at least* 2x as long as the longest time estimates made at the project's origin.

One unfortunate corollary is that "software development engineers are always lying." This is a matter of degree. Many don't know that they are lying. At the other extreme are those like the Lotus engineers who, in 1988, held "ship" parties for products that didn't exist.

Why does this happen? It is partly due to the fact that, even at this late date, after years of RAD and OOPs methods, writing software is still more of an art than a predictable science, especially for large-scale systems. And it is easy for complex projects to get into situations where the system is so interdependent that small improvements in one area actually produce deterioration elsewhere. This is a particular vulnerability on projects where the specifications run for hundreds of pages and no one is able to read everything.

To avoid such black holes, one needs to plan for the inevitable setbacks by using platforms and architectures that permit rapid prototyping. And it is also important to put people in charge who have experience with the complexities of team management, a nose for being lied to, and an honest streak.

6. Open Systems Trump Proprietary Systems – But Then, What's Really "Open"?

By now we all realize that Apple should have licensed its hardware and OS to other computer manufacturers in the mid-1980s, to compete more effectively with the PC. But it is also clear that huge successes have been scored by players like Microsoft that have kept their crown jewels proprietary. What turns out to be an "open standard" is in the eyes of the beholder – for example, in the document viewer space, Adobe has mastered playing both sides of this street with its Postscript platform, publishing specifications publicly but also keeping key details like font formats to itself. With respect to the Java platform, Sun has also been able to combine control over specifications with lots of noise about "openness," supported by its partners in the anti-Microsoft alliance, Oracle, HP, and IBM.

Meanwhile, if anything, the open-source community has suffered quite a few legal defeats lately, notably the upholding of legal protection for software patents, the 1998 Digital Millenium Copyright Act's bans on reverse engineering and decryption tools, the Sonny Bono Copyright Term Extension Act, and the amazingly pro-software industry Uniform Computer Information Transaction Act, recently passed by Virginia and Maryland and on the agenda in many other states. For the proprietary software publishers in the audience, all these post-1998 measures have quite been a windfall – sort of the digital equivalent of the 16th century English enclosure movement.

7. Client- Server Doesn't Scale – Services vs. Products

Whenever one sets out to design a software system that might eventually deliver a new service that handles thousands or even millions of users, it is important to realize from the outset that "client-server simply doesn't scale."

This means that beyond a certain number of simultaneous users – say, 10,000, as a rule of thumb, or 200,000 total users, based on 5 percent concurrency – the entire system can simply grind to a halt, pending a completely different architecture. If one is designing a platform to serve the masses, therefore, he may want to consider proxy- or peer-to-peer architectures from the get-go. The difference between running a small service and a large one on "client server" is such that, if we were in Vienna, we would say, "We wish we could play it on the piano."

8. Security is a Chimera

It is now fashionable to make loud noises, in these times of terror, about grave threats to "network security," denial of service attacks, unauthorized intrusion, viruses, and so forth. It is not easy to say anything general about this issue that is not also vapid. There is evidence that network security threats – to privacy, authenticity, data integrity, and non-repudiation – are on the increase, and of course we should be concerned – especially if, like many corporations, we have things to hide.

However, one is also struck by the fact that very few large scale cybercrime losses have actually been reported. Perhaps this is because such episodes are always covered up by institutions victimized in order to protect their reputations. More likely, we think, it is because the incidence of truly costly cybercrime events is – so far at least – very low.

And even when such security calamities have occurred, the most serious transgressions turn out to have been by *insiders* who had special access to the enterprise jewels. The solution to prevent such episodes is not better software, more sophisticated firewalls, encryption schemes, variable IP addresses, and the like, but better HR policies and, perhaps, more polygraph tests.

From the software developer's standpoint, the beginning of wisdom here is to understand that most customers find it very hard to value incremental security. Indeed, those systems that get adopted most rapidly are often ones that have been optimized to just get something done, regardless of security issues.

For most enterprise applications, this balance makes sense. On the one hand, absolute security cannot be obtained except by using one-time pads and going anonymously to dead-drops in the middle of the night. On the other, the advantages of speed and flexibility gained by using simple, low-cost, if insecure, methods to communicate often dwarf the risks – it may be better to send a thousand message and lose ten to interlopers than to send only ten and lose none.

So while security is an important concern for many software applications, unless one is producing a product like a virus scanner whose essential competitive position is its security, "good enough" is the appropriate mantra.

9. Outsourcing Software Development – You Pay Peanuts, You Get Monkeys

Earlier we made the point that there are an increasing number of outsourcing alternatives available for software development, including many offshore companies in low-cost countries like India and Russia that now offer contract outsourcing services. The counterweight to this argument is illustrated by the story of the former Tanzanian President, Julius Nyere, who turned up one day in the London office of McKinsey & Co., seeking professional advice on a reform project that his government was planning. He spoke with a senior partner, got a rough estimate of what the project might cost, blanched at the high figure, and left. Six months later, however, he was back. The partner asked what had happened, and he replied, "I found out that if you pay peanuts, you get monkeys."

The same principle applies to software outsourcing. Outsourcing may be fine for run-of-the-mill projects that are not time-critical. But in our experience, for those projects where a company's economic life depends on timely execution, in-sourcing should be the presumption, because it maintains tighter control over development teams and accumulates valuable experience inside the organization.

This doesn't preclude the *combination* of external/ contract and internal/ employee developers, working on joint teams. But the notion that mission-critical projects can be outsourced to the ends of the earth is a risky one. In the management of large-scale software projects, there is much to be gained by the discipline of daily code reviews, shake-down sessions, and face-to-face developer reviews.

This requires close working relationships among team members, which is hard to synthesize across long distances Furthermore, if an enterprise wants to develop a core group of experienced developers with deep expertise in a particular software arena, it undermines the value of its own "human capital" base if it allows that experience to be accumulated outside the company.

10. Great Developers Need Deft Managers

Our final lesson learned is that, despite all the trends toward maturity, incrementalism, and tedium noted above, there are still quite a few truly outstanding individual software developers around, and it is important to learn how to manage them.

We suspect that such folks are neither born nor made. As Gates once said of Ray Ozzie, the key designer of Lotus Notes™ and founder of Groove, "He is God's coder." By one estimate the productivity of top software developers can be at least 10 times that of an average one, in terms of "working code per hour."

So does this mean that we should try to populate our teams with as many of these supernovas as possible? This does not necessarily follow.

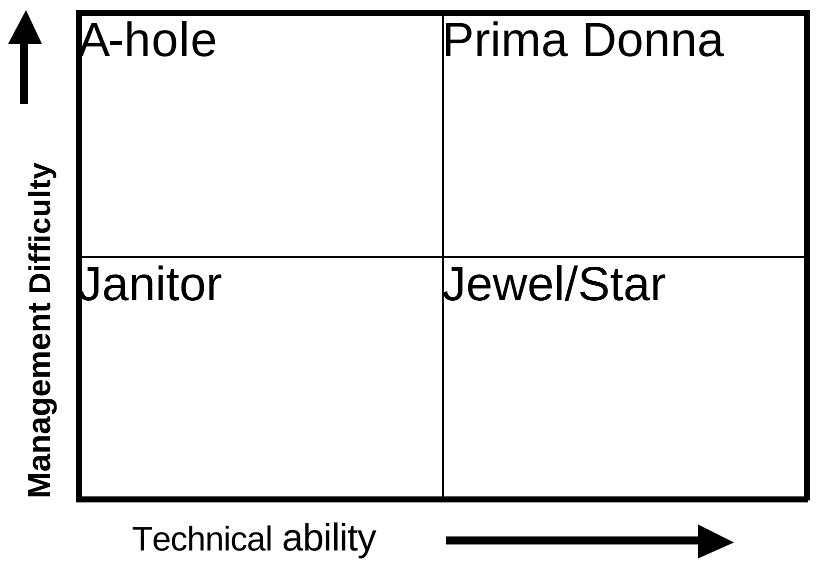
In the first place, such individuals are scarce, in excess demand, and to some extent in hiding.

Second, the stars are rarely the easiest people to get along with, especially when they are forced to work with large teams – the unflappable Ozzie being a distinct exception. Quite the contrary. Many of the most brilliant ones conform to the "Richard Stallman" model, practically living in their offices for weeks at a time, single-mindedly focused on the code at hand.

There is also the propensity, when one goes in search of such folk, to confuse idiosyncrasy and peculiarity with true genius. This is partly because it takes time to figure out that someone's code is less brilliant than their style might indicate. The chart below captures the most important character types that turn up regularly on the software stage.

Of course we are all searching for the "jewels" who are technically astute *and* a real pleasure to work with. Another group is the "A-holes" who can be so difficult to manage that their actual lack of technical ability is disguised. These folk can survive on a project for months just by pinning the blame for slips and bugs on other teammates. Eventually they sink to the bottom, but often not before real damage has been done.

Perhaps the most challenging types to manage are the "prima donnas," who are technically astute, but are also acutely conscious of their native superiority, thank you very much. Whether or not these folk are worth the opportunity costs of all the teammates they alienate is a tough call indeed.



Summary – SHG's Role in Software Development

SHG's team members have managed or assisted with numerous sucessful large-scale software development projects. We also pride ourselves on having told several clients that their development projects either needed complete "rethinks" from the ground up, based on a hefty dose of customer and channel partner input, or that they really ought to consider alternative "technology sourcing" strategies – like sourcing key components of their development plans from outside, or, on the other hand, scrapping expensive outsourcing or licensing arrangements with outside vendors and building things in-house. In the software development arena we have performed successful client assignments in the following arenas:

- ❖ Customer Requirements Analysis and Specification.
- ❖ Technical Due Diligence – helping senior management develop an independent perspective on how internal development efforts are likely to satisfy customers and stack up against competitors, now and in the future.
- ❖ Negotiation Support. Providing on- support for software licensing strategy and , including financial structuring, development of "playbooks" for negotiation conduct, and actual bargaining involvement.

- Technology Sourcing Decisions.

We'd be delighted to work with you on any of these issues, on a standard consulting fee basis or a success-fee basis. Please address all inquiries to Jhenry@sagharbor.com and chris@sagharbor.com In any case, good luck with the coding – and just remember, someone **has already** had these problems before!

****