Lessons from the Science of Nothing At All

Richard P. Gabriel
Distinguished Engineer
Sun Microsystems, Inc.

Where I come from we make things from nothing—from dreams and fantasies. The laws of physics don't apply. Our products weigh zero. We've explored just about every product development approach there is—extreme or otherwise: waterfall, iterative, rapid prototyping, community development, pattern languages, and mobs.

[Pause]

Imagine this: Dionysius has deleted all the software on the planet—what's going to happen? You won't be able to surf the Web, send e-mail, make and run spreadsheets, use word processors, and download music. No more anonymously available dirty pictures, no more mapping services, no more reading newspapers from around the world, no more computer and video games, no more pocket organizers, no more modern warfare, governments will come to a halt. Well, that's not so bad.

Oh, and movies will have cheesier special effects, Playboy centerfolds will seem to have more blemishes, buying airline tickets won't work well, booking hotels will take longer, no one will be going into space, you'll mostly be watching local TV, you won't be getting the New York Times, Wall Street Journal, and The Economist in the mail, some airplanes will fall out of the sky.

Wait, what's that? —This is getting serious. No more telephones, some buildings will stop being livable, most cars will stop running, food won't be delivered efficiently, health care—don't want to think about that. We'd be thrown suddenly back into the 1950s or 1940s. Just about everything we do every day will change or disappear.

Software is a real part of our lives, a real part of the world. If it wasn't real, then why does deleting it break so many things? Well of course it's real: Big companies make lots of money creating it: Microsoft, Oracle, IBM; even smaller ones: Netscape, Borland, Adobe. A few million people earn their livings writing software, an equal number attend to those who create programs: documentation writers, testers, managers, customer support folks.

What's writing software like?

Suppose I showed you two rooms where in one a group of programmers was developing a program to monitor a heart patient and keep him alive until the doctors arrived if something goes wrong and in the other a group of street people was using Microsoft Word to write letters to their parole officers. You would not be able to tell the rooms apart.

Here's the problem and the intriguing possibilities of studying software and how to make it as exemplars for all sorts of making—or if you prefer, product development.

What of lot of you make uses stuff in the physical world. You hammer it together, stick one thing to another, screw this into that. The stuff wears out, you put oil on it to make it run smoother. A guy in overalls can fix it. You deliver it to people in boxes. When you take your product to the post office and they put it on a scale, it weighs something.

That is, you have reality on your side. You make things in 3-dimensional space. Maybe it seems to you that these are limitations that you'd like to unload. Don't. Hang onto those things. In my business, we don't have simplifying constraints like these.

On the other hand, we have some things better. I've never had a few thousand gross of bad IF statements delivered.

Let's look at some statistics. You can hardly pin anyone down on numbers like these, but a typical number you'll see for an average programmer is 50 lines of tested code a day. At delivery, there can be about 30 defects or bugs per 1000 lines of code.

Bugs are probably not quite what you think. Imagine you've designed a car and built it, but someone designed the wrong size tires for it. If the car were software, maybe this would cause the car to not start, or you wouldn't be able to steer it, or maybe the car would explode. Or maybe everything would go fine until the driver tuned to NPR and then the gas tank would spring a leak.

Windows NT has nearly 50 million lines of code in the full build. That would take one person about 2700 years to complete. Oh yeah, and it would have about 1.5 million bugs. Adding people to a project can speed things up, but not linearly. If you add a programmer to a team larger than 50 people, it's not unlikely the productivity of every programmer will drop, and at some point, increasing the number of people on a team will reduce its total productivity.

Bugs are not a good thing. Some of the recent Ariane rocket failures were the result of bugs—some of them downright silly.

[Pause]

The Ariane 501 was the European Space Agency rocket that self-destructed after its engine nozzles all swiveled into an extreme position, causing the rocket to veer abruptly, which caused the rocket to self-destruct. The bug had to do with some alignment software on the navigational computer that normally runs right after they put the rocket on the launchpad, but in this case it kept running well into the launch during which time it computes nothing meaningful. This software eventually encountered a bug since no one was aligning anything. The navigational computer sends navigational data to a command computer, which directs the rockets. The navigational computer put a textual error message on the data bus to the command computer and then shut itself down. The command computer took the textual error message as extreme navigational data and swerved the nozzles. Then it noticed the navigational computer was off and switched to the other one, which had also shut down for the very same reason. The stumped command computer turned to its backup, which was also stumped, and after noticing the extreme aerodynamic stress on the rocket, one of the two decided to self-destruct.

You might think that the software running on the navigational computer wasn't mature or well-tested. In fact, it was used by the Ariane 4 rockets for many years without any problems at all, and not a single line of code was changed when it was put on Ariane 5 rockets.

Software for space rockets is written much more carefully than commercial software. Here's how the software for the space shuttle is written.

SEI Level 5 is the highest rating for a software methodology assigned by the CMU Software Engineering Institute. Lockheed Martin Corporation employs a SEI Level 5 software group that puts together the shuttle software.The software is not very big—only 420,000 lines of code. It was developed over a 20-year period by a group of 260 people. These numbers imply that each line of code has had 25 person-hours of attention, and that each person has been responsible for a little over 1600 lines of code—25 pages. $700,000,000 was spent to develop this code, or about $1700 per line.

Further, the software had been completely specified in pseudo-code agreed to beforehand. The task of a programmer in this regime is to translate pseudocode to actual code. But, all this work ***does*** buy NASA only 1 error in each of the last three releases of the code. Because we cannot afford to have deaths in the space program, the cost and effort are worth it. The cost amounts to making each subroutine a career-long research project. Twenty years, 8 hours a day, studying and eliminating errors from 1600 lines of code.

This sort of slow, deliberate process of creating software is good at producing bug-free code. Variants of this approach have been used since the beginning of programming. One popular variant created in the 1970s is called the Waterfall Model which breaks up the process into 5 stages: The first stage is understanding the concept, doing analysis, and defining the requirements; the second stage is design; the third stage is implementation and coding; the fourth stage is integration of the parts and testing; and the fifth stage is operation and maintenance.

The Waterfall Model was a response to earlier methodologies that quite essentially started with someone shouting, "hey, let's make a text editor," and proceeded with people running toward the keyboards. Deep thinkers realistically believed that using the Waterfall Model, the quality of software could be improved and the process of creating software could become much more predictable.

Shallow thinkers quickly noticed that this reality thing was going to be a problem. Recall, software creation doesn't get the benefit of reality and 3-dimensional space. What does this mean? When you're designing a real product, you have centuries of experience with the world and real artifacts to base your designs on. Designing a new hand tool?—60 centuries of experience, has to fit human hands, has to accomodate human motion. Does it cut? Done that? Does it fasten? Done that. Does it make holes? Done that. Designing a new telephone? Two or three generations of experience there. New door? Has to open and close, has to be big enough to let what's to move through, move through. New stove—people have been cooking for hundreds of centuries.

But, consider the first people to design and build a text editor. Before that, there was never a text editor. Changes to a manuscript were always made by retyping or retypesetting. How would people want to make textual changes? How would people want to navigate? Searching?—no one ever heard of that before. Systematic changes? Huh? By the way, there were no display terminals, so how do you even look at the manuscript?

So what do you think would happen if you tried to design and implement a text editor for the first time? You'd maybe talk to some folks and decide what it was going to do. Then you'd design and build the software—keeping in mind that the slightest, most picayune mistake and maybe nothing works at all. Designing and implementing it so it works maybe would take a couple of people 6 months. Once it sort of worked, you'd let people use it, and once they had seen what can be done, then they would start saying, "oh, now I get it, well, how about if you add this and that to it and change this so it works like this."

Then the developers would go back and make another few months of changes, and the process would repeat. Why? No one has ever seen a text editor before or maybe only the previous versions, and so they can't imagine how they would really use it and what the experience would be like until they had one. Then they would start to use their imaginations to guess what it could be like, but they are only guesses. The people who are burdened with coming up with the requirements for the first text editor **cannot** do it because first, they have never even heard about a text editor before, and second, they have no idea what is possible or impossible for software to do. Usually they require only modest and conservative things because they think what they can imagine is too hard or impossible, and once they see that something they thought would be impossible can be done, they start expanding their imaginations. So they iterate and try and try things.

For products in the real world, you have already had the decades, centuries, and millennia to go through this iterative process, and we software guys have had 50 years total.

[Pause]

Let's go back to that Waterfall Model. Here's its first step: Understand the concept, do analysis, and define the requirements. If you can do that, then the model will produce a program that satisfies the requirements. The Waterfall Model depends on having the right requirements, and initial requirements are usually just a guess.

A number of software processes in use assume that the real requirements can be fully and accurately specified **beforehand**. Why do they think that when it is clear that when you're building something no one has ever seen or had experience with before, you are at best guessing every step of the way?

In response to the Waterfall Model, to other methods were proposed: the Iterative Method and Rapid Prototyping, for example.

Rapid prototyping uses a special prototyping language to make quickly a prototype of a system. With this prototype, potential users can try out the software, and the real requirements can be figured out. Then a more traditional method like the Waterfall Model can be used to implement the requirements using a real programming language.

In the old days—15 years ago or so—prototyping languages were distinguished from real programming languages by not requiring as much description of the program as is needed for real programming languages. The reason real programming languages require a lot of description is so that a compiler can produce very efficient programs using highly efficient data representations—a compiler is a program that translates computer programs that people write into code that computers can execute. A way to look at the difference between prototyping and real programming languages is that the prototyping language spares the programmer from typing in a lot of extra stuff, which means the

computer needs to do more work while the real programming language makes the programmer do the extra stuff so a computer doesn't have to do as much. The result is that with a prototyping language, you can write code a lot faster than you can with a real programming language, and with a real programming language it takes a lot longer to program, but the code runs fast. In the old days when computers were slow, it was worthwhile to make people work hard in service of computers. In the old days, Java would probably be called a prototyping language.

The Iterative Method employs ***use cases*** to define what the software will do. A use case is a scenario of usage and what should happen—that is, a person does this, that, and the other thing, and this is what the software does in response. The total specification of what an interactive program should do can be made by describing all the possible use cases.

The iterative method proceeds by gathering use cases, then implementing enough of the system to implement a small batch of the use cases, trying the system out, then adding new use cases or modifying existing ones, and then doing more implementation, etc. The idea is to intersperse understanding the nature of the software being written with its implementation.

With both rapid prototyping and the iterative method, the goal is to speed up understanding what software should do and how people can best interact with it—because with software we are building things that have no physicality and which do not adhere well to physical metaphors. In many cases we are creating things which have never been created before, and we have no idea how people will use them, what they will find hard, what they will find confusing, and the process of writing software is so arduous, time-consuming, and money-intensive, that we have not yet been able to try enough experiments to know the answers to those questions.

Here is one way to look at it: If you place almost anyone from a Western culture in front of any desk that they have never seen before, he or she will be able to look through the drawers, find pens and pencils, and otherwise do their work there. The primary computer metaphor is the desktop, developed by Xerox PARC in the 1970s and "borrowed" by Apple and then Microsoft. But it isn't a real desktop—it's a metaphor, and not such a good one at that. Stuff appears on a vertical screen and not a horizontal desk. The sorts of things that appear there are not books, pencils, and papers, but documents and applications—or nicknames for applications. You don't reach for these things with your hands, but maneuver a small arrow using a round brick on the desk or by twirling a ball mounted on a holder; you don't grasp them with your fingers but operate them by pushing buttons on the brick or ball-holder.

Here, the designers have tried to use a physical metaphor to make understanding the software easier, but operating within a metaphor is not the same as operating in the world. Moving a mouse and clicking buttons to get things done is not natural to people nor are there the centuries of experience with machines similar to computers. Today we take for granted the relatively new idea of a steering wheel on a car. But such steering wheels have been around for about a hundred years, and the use of a "steering wheel" for controlling the direction of ships has been around since the 18th century, where it was found to produce better control with less effort because it used a block and tackle for mechanical advantage.

And the steering wheel is a physical device—not a picture of one—and it's operated by placing one's hands on it—not by pointing an arrow at it and pushing a little button twice.

[Pause]

The two most successful software applications have been the spreadsheet and the text editor. The spreadsheet, developed by Dan Bricklin and Bob Franskton in 1979, mimicked the tabular notation of bookkeeping and married it with the functionality of popular business calculators of the time. The result was something both familiar and new. Given the familiar, the new was easier to learn than if it were all new. The familiar parts were common to people who did business financials—the layout on the screen was not a metaphor, but a duplicate of what accountants would use on paper. What was new was the command-style programming of the spreadsheet.

Nevertheless, the spreadsheet was something never seen before. A chart indicating the 64 greatest events in accounting and business history contains VisiCalc.

The text editor is nothing more than the combination of a piece of paper in a typewriter with other commands and functionality that before could only be performed by retyping or retypesetting. Again, the software combined the familiar with the new: typing, moving around on the page using simple commands like <space>, <tab>, and <return> along with new functionality like search and replace, text justification, font control, and line wrapping. Nevertheless, habits die hard. Some fiction writers are used to revising their stories and novels by literally retyping their manuscripts and regaining the feeling and rhythms of typing the words. Such writers continue to do so: Printing out the manuscript and retyping it rather than fooling with small changes, rather than looking for ones with deeper significance.

The first recognizable text editors were developed about 30 years ago. Since then we have gone through 3 generations and we're on the 4th—Microsoft Word being somewhere between the 3rd and 4th generations, not having become competent at typesetting. Yet people do not fully utilize the functionality available, or they get confused by the user interface. There are icons and tool interfaces all over the screen, and it's hard to tell what is controlled by what. Except for the typewriter part of the interface, nothing about what confronts a writer is familiar and is not part of the physical world. The physical controls of a computer control the virtual world of the software, and that virtual world need not obey any laws of physics or common sense. People are not used to making up paragraph templates or styles, they cannot make cross-references or indexes. Nevertheless, they can operate complex physical machines like lathes and trucks, cook using sophisticated stoves and ovens, ride bicycles with 24 gears, fly airplanes, and sail boats

These, though, are the simple cases. Most business software has no physical equivalents, and even metaphors are hard to come by. The people who pay for the software sometimes aren't the ones who will use it, and the people who use it will not be well-trained—they will simply have to "pick it up," so to speak. Because there is no requirement for the innards of software to make physical or common sense, and because the connections between the cartoonish controls on a computer screen don't have to have a physical or logical connection to what happens inside, people—whether trained or not—can develop incorrect mental models of what's going on inside the software and how their actions control things.

Designers of physical devices have an expert collaborator: the physical world. Because the physical world can, in theory, make it possible for someone to figure out the innards of the device and how they work, the designer can take advantage of his or her helper to make things sensible to a person

using it. The designer of software has no one to help simplify things. The complexity of the model embodied in the software—which sometimes doesn't correspond to any real-world situation or any "model" of the problem being solved—is inscrutable, and leads directly to difficulty using the software. Keep in mind that the software not only has to model the objects and operations involved in the problem but also has to model objects and operations for presenting itself to the user, and for interacting with the user. The physical world of inanimate objects does not care about interacting with us, and leaves that messy job to the laws of physics. For software developers, it's left up to the laws of the imagination.

[Pause]

As software developers we face two grave difficulties: One is that we are building things that have never been seen on the planet before, and the other is that the tools we use to build it with are so brittle that it takes an enormous effort to get it done, and the smallest slip-up can cause none of it to work.

We learned these lessons in the 1970s, but things take time to percolate to the industry. In the mid-1980s, Harlan Mills at IBM introduced the ***Cleanroom Process***. This process considers the program you are trying to write as an expression of a mathematical function. Here's how it works: First, you specify all the components of what you want to write and how they interact. Each component is assigned to a person or team which performs stepwise refinement using structures called "box structures." Through reviews and verification, all the components are shown to be correct. Then by testing statistically determined uses, an operational view of the software including its performance and correctness can be drawn. This testing is done by a separate group, and no code is executed by the development team prior to the independent testing.

If at some point a problem is discovered with the design or specification, the development group goes back to the design phase. In this sense it is an incremental process.

With this process as well as with the Waterfall Model, you had better know what you want to build, because if it turns out you've built the wrong thing, going back and fixing it is going to take a long time given the programming languages and tools we use.

The processes I've talked about up until now generally focus on the predictability of what will be built—that is, if you really know what is wanted and needed down to the last detail, then you can use one of the planning-heavy processes or methodologies to build it effectively and predictably. For some software like the space shuttle software, this is a great thing to do: Plan it out, design it to the last detail, prove everything you can about it, then build it as carefully and deliberately as you can.

But progress and business won't wait for that. Businesses thrive by being agile—adapting to changes rapidly. And if you are building something that has never been built, then you'd better be prepared for surprises—if for no reason than if you happen to come up with something great that wasn't what you were planning, you'd be smart to exploit it rather than throw it away.

The other issue is that the people who are best at building software that has never been built before are usually quite talented, creative, and intelligent. It is not going to work to design out to the last detail something that has never been built before and then ask talented, creative, intelligent people to

build it but **not** react when things go wrong or a better design or approach pops up. They will not work for you if you do this.

Think about how bridges are designed and built. Designers first come up with a detailed plan for the bridge based on geological investigation, centuries of experience building bridges, and careful, precise mathematical analysis, an analysis made possible by the nature of the physical world and science. Then, another company is hired to apply physically skilled construction workers and unskilled laborers to construct the bridge according to a relatively predictable schedule and known costs.

Many would like software to be like that. It cannot be because we don't have the physical world and experience to make things predictable—you need them both. We have neither.

[Pause]

In response to recognizing that what we are doing is unpredictable and requires very smart people, the software community has developed a group of processes or methodologies noted for being lightweight and adaptable. Such methodologies are sometimes called ***agile*** methodologies.

The best known of these methodologies is called ***Extreme Programming***. It exemplifies the agile methodologies in all but one particular way, which is that it does not emphasize adapting the process itself to the team and circumstances as much as other agile methodologies do.

Extreme programming expounds practicing to an extreme activities that other methodologies promote.

Code reviews by others are considered an important part of ensuring quality early in the implementation of a program. Therefore, Extreme Programming requires that developers work in pairs, sitting at the same computer all the time with one then the other controlling the keyboard and mouse so that the code is being continually reviewed, not simply periodically.

Testing is important to quality, and usually it is performed toward the end of a project when it is almost too late. Therefore, Extreme Programming requires that tests be written at the same time as the code or before, and that all written code must pass all its tests all the time. This way, testing is continuous.

Design is important to any piece of software. Therefore, Extreme Programming requires that design be part of everyone's daily business.

Simplicity of code and design is good. Therefore, Extreme Programming requires that the code in the current system be the simplest code that could possibly support the current functionality.

Architecture is the fundamental stuff of a maintainable and changeable system. Therefore, Extreme Programming requires that everyone with a stake in the software should continually work at defining and refining the architecture.

Short iterations and frequent integrations are good because they enable the team to see where the software is going, to see how users respond to it, and to adapt to newly learned requirements. Therefore, Extreme Programming requires that iterations be very short—seconds, minutes, and hours

rather than weeks, months, and years—and that integration tests be performed several times a day rather than every few weeks or months.

Other agile methodologies build self-review into the responsibilities of each team, so that not only are the plans and designs for the software subject to adaptation, but so is the process itself.

Extreme Programming results in quick turnaround for small additions and changes, so adaptation to what's learned can be immediate, and such a process keeps talented developers challenged and alert.

For predictable software—software that's been written dozens of times before or which is the result of some sort of detailed mathematical analysis—you wouldn't use a methodology like this. But for something never built before? Or heard of before? Or never heard of by you and your team?

[Pause]

I believe creating a completely new product is a hard concept to understand for someone who has not either developed software or been deeply involved with some creative activity. It seems that making software must be like designing and building a house, or a car, or a highway, or a TV, or a stove. Let me try to provide a scenario that might make it clear.

Suppose you wanted to design and build a gas stove, but no one had ever used gas to cook with before. In fact, assume that before this, all that had ever been used was a wood stove.

Here are some things you wouldn't know: Will you use the gas flame to heat a piece of metal on which the pots and pans will sit, and that metal will transfer the heat, or will the flame be applied directly to the pots and pans? How will you control the temperature? Will you require that people move their pans from places of intense heat to places of less heat? Will you provide a gas flow controller that will reduce the amount of fuel to the flame? Will you move the flame closer or farther away from the pan or heat transfer metal? Will the heat be a manageable temperature for all kinds of cooking or only for limited use? Will cooks be afraid of using gas? How will you get it into the stove? Will it be stored or continuously supplied? How will you control the temperature of an oven using gas? Will a stove designed this way be affordable? Will the stove explode? Will the gas smell funny? Will the stove need to be inside or outside the house?

There are thousands of questions like this in every software project I'm calling unpredictable, and I'm also claiming that almost all software projects are unpredictable.

[Pause]

Recall that I said that unpredictability is only half the problem. —That the other half had to do with the programming languages and tools we use which make it very hard to write software that wasn't brittle. —And that the brittleness had only a shallow relation to cause and effect.

Writing a correct program requires enormous attention to picayune details written in a language designed for mathematical computation, but symbolic or business reasoning.

Because of that, design, proving correctness, and testing are required. Extreme Programming makes you test continuously. The Cleanroom Process makes you both prove correctness and test.

Programming is very hard work because in many ways, the stuff inside a program is too much like a complicated mathematical function that somehow does what you want, even when mathematics is not part of your game. Computers deal with bits, bytes, and simple logical and arithmetic operations on them. Most of the programming languages we use do not raise the level of abstraction much higher than that, though when designing software we hardly ever think in those terms.

I mentioned prototyping languages before. Those languages did raise the level of abstraction quite a bit. This made creating software a lot easier and quicker, and they were less prone to errors. In many Prototyping languages, errors were assumed to be common and such languages provided easy mechanisms for programmers to decide how to deal with them or for programmers to decide how the programs would decide to deal with them. Today programming languages call errors "exceptions," and deal with them that way.

Languages like these could make a big difference today in how fast we learn and the satisfaction level for consumers. But over 10 years ago the computing world decided to give up on languages like those. They ran way too slowly. They were a little too big. Of course, now that computers are 500 times more powerful and 250 times larger than back then, maybe this wouldn't be as true, but computer science research and computing practice have both forgotten about these languages and approaches and, in some cases, essentially outlawed using or thinking about them.

Well, so we program in the dark ages—what of it? Actually, we've figured out a way around that too.

Because programming languages are too primitive, a large program is too complicated for one person to understand and get right, and there are too many bugs and other errors for one person to find and fix, and it's just too large for one person to get done quickly. How do you solve a problem like this? You have thousands of people work on it. This is called Open Source.

To understand how this works requires a little explanation. When you design, build, and sell a car and it's sitting in someone's garage, its innards and how it works can be—more or less—discovered by opening it up and peering inside. In fact, if something goes wrong with the car, maybe you can fix it yourself, or maybe you can take it to a repair shop where someone will fix it. Or suppose you don't like the shocks or fuel injection system. If you're real good, you can change those things or someone can do it for me. You could even install a nitrous oxide system if you wanted an extra 200 horsepower for short periods at a time.

In fact, just about everything we as consumers can buy have the property that there are plenty of physical things about it that a consumer can fix or customize or pay someone to do for them. Not so with software.

Imagine that you open up your hood and what you see is sand. Open up your TV—sand. Look in the back of your refrigerator—sand. Unscrew any access plate anywhere in your house—and it's more sand. Change or move one grain of it and—ka! boom!—it doesn't work anymore or does something you don't like such as show only the Home Shopping Network, heat up your ice cream, or make your car go only backwards honking the horn. Sounds a little nuts doesn't it?

But that's exactly what software is like. The software in a form that people have a chance to understand is transformed by the software maker into something only a particular set of computers can

understand, and that's what you buy. Even if you were capable of finding and fixing a bug in Microsoft Word, you couldn't do it unless you worked for Microsoft.

Let's go back to the real world example. If the innards of your car were like software, then when it needed to be fixed, the original manufacturer would have to be contacted. They would prepare a new batch of sand, and that batch would replace the sand in your car.

Ask a software company why they want you to not even see what is inside they sell and they will tell you that either it's too complicated for you to understand or that this way they can protect their trade secrets.

Both patent law and copyright law apply to software—this is more protection than the auto manufacturers have—so that can't be a good explanation.

Cars are pretty complicated too, so that can't really be the reason. Maybe the clue is in the idea that for you to get anything changed about the software you need to go back to the software maker.

In the auto industry there is an aftermarket for spare parts, custom kits, instruments and tools for diagnosis and repair, and repair shops. We've got that for a most consumer electronics, white goods, lawn mowers, houses, bicycles, musical instruments, on and on, and even computers. But not for software.

If you buy a book and find a typographic error, you can take out your pencil and fix it. If you find a similar typo in the user interface of a program, only the software maker can fix it. Sounds like they've maneuvered themselves into quite a nice place, don't you think?

Well, Open Source is out to change all that.

In Open Source, all the source code for a system is available for anyone to look at and modify for their own purposes. The source code is what the software makers turn into sand. In an Open Source project, the mechanisms for turning source to sand is freely available.

The source for the project is kept in a special, open place, and changes to the official source are loosely controlled by a group of developers and perhaps some governance process. This group of developers is dedicated to enabling the community to move the software along without hogging control of it. If anyone finds a bug and sends the fix to one of this group, the fix will be put in right away. If anyone makes an improvement, the same is likely to happen. If someone thinks they have a better way to do part of the system and the developers in charge don't agree, that someone can fork off the source and start you own competing project.

Typically, the Open Source project has an active community with mailing lists, a Web site, and e-mail archives containing all the design and implementation changes. Alterations to the software are openly discussed and such discussions can be brought up by anyone. Usually some consensus is arrived at and changes are made.

Companies can take the results and by either adding functionality, providing support, and surrounding it with other software, make money from the software—often as much as closed source companies do.

In an Open Source project, there can be tens of thousands of people who use the code and help fix it, and hundreds or thousands who contribute to its development. In any given day there can be hundreds of e-mail exchanges about the software. The community itself is the support organization, and just about every question is answered by the community or by its archives.

Testing is continuous, integrations and releases happen just about every day. Design and implementation is done in the open on mailing lists. All the design decisions are thereby written down. Architecture, design, implementation, and testing are done in parallel by hundreds or thousands of people, not just by a pair or a small group.

There are two important things this accomplishes. One is that the fact that programming languages are backward doesn't matter too much to the project, because there are lots of people to share the burden with. And the direction of the software is subject to comment by thousands of people, and if there are severe disagreements, a competing project can start up from the same source base. This way, the process of finding out the true requirements for a piece of software never yet built can be rapidly discovered.

[Pause]

What I've been talking about is creativity in a hostile medium—making things that have never been made before using tools not suited for it. But we're doing a pretty good job.

Creativity is a funny thing. It takes lots of practice and lots of whacks to get something right. When the poet James Dickey wrote a poem, he would sometimes do 500 drafts to get the right one. For a National Geographic article containing 22 pictures, the photographer took about 40,000 pictures—over a thousand rolls of film. Before the novelist Larry Brown was published the first time, he had written 100 short stories and three novels, one of which he burned in a 50-gallon drum in his nackyard.

When you design for the physical world, you design **with** it. Nature is your collaborator. Most product designers have the advantage of centuries of experience with artifacts in the physical world, and so they've had 500 drafts or 40,000 attempts in their experience base. Software is fiction, it is imagination. Not many rules apply. We are just 50 years into it and going as strong as we can.

What we've learned so far is that you can build things out of junk, but that finding out what to build and how to best design it takes creativity,—that is, lots of tries and working with the folks who will use the stuff in the end. It's too bad building software isn't like building a bridge. It's more like writing a novel—maybe a science fiction one.