Anything You Can Do, I Can Do Meta

Space tourist and billionaire programmer Charles Simonyi designed Microsoft Office. Now he wants to reprogram software.

by **Scott Rosenberg** January 1, 2007

On April 9, at a remote launchpad on the plains of Kazakhstan, a ground controller will finish his countdown; a Soyuz rocket will fire; and Charles Simonyi–Microsoft's former chief architect, the tutelary genius behind its most famous applications, the inventor of the method of writing code that the company's programmers have used for 25 years, and now the proponent of an ambitious project to reprogram software–will begin his ascent into space.

Charles Simonyi president and CEO of Intentional Software.

Snug in a Russian space suit, feeling four Gs pressing him down intc
liner, the 58-year-old billionaire will become the fifth space tourist tc
Space Station. The journey, which will cost Simonyi around $20 mill
becoming a "nerd in space" (to borrow one name he chose for the w
extraterrestrial adventure: www.nerdinspace.com). It will also give h

Soviet mainframe powered by vacuum tubes, then engineered his own escape to the West. In the 1970s, at Xerox's legendary Palo Alto Research Center (PARC), as part of the team that invented personal computing, Simonyi wrote the first modern application: a word processor that banished the complex codes then used to tag text and displayed a document as it would look on paper. Whether in his Stanford University doctoral dissertation on a "meta--programming" approach to boosting programmer productivity, his career at Microsoft organizing legions of software developers and teaching them how to structure their code, or his planned voyage into Earth orbit this spring, moving beyond established ways of doing things has always been Simonyi's method. Now he is plotting what he hopes will be his most vaulting meta-move of all. Simonyi believes he can solve a host of stubborn problems that have always plagued computers by offering everyone who uses them, and the coders who program them, a higher-order view of software.

Bill Gates calls Simonyi "one of the great programmers of all time." Indeed, Simonyi is arguably the *most* successful coder in the world, measured in terms of financial reward and the number of people who use his creations. (Other celebrated programmer-billionaires, such as Larry Ellison and Bill Gates himself, made their money and names founding and managing technology ventures.) Simonyi could easily choose to spend the rest of his life endowing philanthropic ventures, flying planes, or cruising in his yacht. Instead, he says, he is "programming probably harder than ever before." He is obsessed with a project that he has pursued for a decade and a half, and that four years ago carried him right out of Microsoft's doors. He is proud of his profession. But he is also haunted by the thought of what programmers must contend with each time they sit down to code. He asks, Why is it so hard to create good software?

Multimedia

Additional Photos of Simonyi

Napoleonic Code

"Our civilization runs on software," says Bjarne Stroustrup, the inventor of the C++ programming language *(see " The Problem with Programming ")* . But the software itself doesn't run very well. Everywhere you look, software is over budget, behind schedule, insecure, unreliable, and hard to use. Anytime an organization attempts to introduce a new system, or upgrade an old one, it takes a colossal risk; today, large information-technology projects are technological tar pits that immobilize institutions. Studies regularly report that two-thirds of such projects encounter major delays, significant cost overruns, or both. The U.S. government has found it nearly impossible to introduce or upgrade large-scale software systems: decade-long efforts at the Federal Aviation Administration and the FBI have collapsed in chaos. Businesses have fared no better. To give a single example, McDonald's executives dreamed of a Web-based management system they called Innovate that would track the real-time flow of burgers, fries, and chicken nuggets in every one of their restaurants around the world. By the on a barracks killed 28 U.S. soldiers.

The past half-century of computing has seen wonderful progress. Programmers have abandoned punch cards and teletypes. They have given us a computer on every desktop, tools for work, toys for play, and a network that links homes and businesses to form a teeming global pool of information and entertainment. This progress has been fueled by the exponential curve of Moore's Law, Intel founder Gordon Moore's prediction that microchips' power would double (or their cost would halve) every one to two years. But even as Moore's Law has made each year's new computers faster and cheaper, the flexibility and utility of our computer systems have been limited by the slower, uneven evolution of software. One formulation of this problem is known as Wirth's Law, after programming expert Niklaus Wirth: "Software gets slower faster than hardware gets faster."

Simonyi shares much of the common dissatisfaction with software. "Software as we know it is the bottleneck on the digital horn of plenty," he says. "It takes up tremendous resources in talent and time. It's disappointing and hard to change. It blocks innovation in many organizations."

Simonyi's ambition is to unstop that software bottleneck–characteristically, by going meta. He's developed an approach he calls intentional programming (or, more recently, intentional software), which he hopes will overturn programming. If Simonyi has his way, programmers will stop trying to manage their clients' needs. Instead, for every problem they're asked to tackle–whether inventory tracking or missile guidance–they will create generic tools that the computer users themselves can modify to guide the software's future evolution.

On a gray afternoon last October, I sat down with Simonyi in Bellevue, WA, in front of two adjacent screens in his office at Intentional Software, the company that he founded after he left Microsoft in 2002 to develop and commercialize his big idea. Simonyi was racing me through a presentation he was preparing for an upcoming conference; he used Microsoft Office PowerPoint slides to outline his vision for the proposed great leap forward in programming. He was in the middle of moving one slide around when the application just stopped responding.

In the corner of the left-hand screen, a goggle-eyed paper clip popped up: the widely reviled "Office Assistant" that Microsoft introduced in 1997. Simonyi tried to ignore the cartoon aide's antic fidgeting, but he was stymied. "Nothing is working," he sighed. "That's because Clippy is giving me some help."

I was puzzled. "You mean you haven't turned Clippy off?" Long ago, I'd hunted through Office's menus and checked whichever box was required to throttle the annoying anthropomorph once and for all.

"I don't know how," Simonyi admitted, with a little laugh that seemed to say, *Yes, I know, isn't it ironic?*

It was. Simonyi spent years leading the applications teams at Microsoft, the developers of Word and Excel, whose products are used every day by tens of millions of people. He is widely

In 2004, Simonyi proposed his own law: "Anything that can be done could be done 'meta.'" In his younger days–when he'd grandiosely named a project "Simonyi's Infinitely Glorious Network"–he would probably have been more arrogant: "Anything you can do, I can do meta!" But like many prodigies who have done well and aged well, Simonyi has learned to cut his cockiness with touches of humility and grace. A decade ago, he described himself as "a shaggy-looking guy with a foreign accent." He favors black turtlenecks and double-breasted blazers. With his upright posture and square face, a shock of dark hair combed forward over his forehead, he is often said to resemble a larger-boned Napoleon.

Intentional software is a grand scheme in a field where grand schemes have seldom worked. Every previous innovation introduced as a complete solution to software's woes has ended up providing no more than modest, incremental improvements. But Simonyi brims with the confidence of a self-made immigrant who's always had a firm grip on his own bootstraps. In a photo that hangs over his desk, he is standing in the White House beneath a portrait of Ronald Reagan. His broad grin mirrors the president's. The caption reads "The Two Optimists."

The offices of Simonyi's new company occupy a suite in a sleek glass skyscraper, and if you lean into the window and look down you can see the roof of the squat, nondescript white building that housed his first office at Microsoft, back in 1981. (It's a bank now.) Since then, Microsoft has grown beyond all recognition. The software industry has transformed the world. So why would Simonyi set out to rewrite all its rules? The problem is so big it seems part of the settled order of things. Simonyi's proposed solution could take decades to complete, and his critics are intensely skeptical. No one is asking him to leave behind the known routines of programming and strike off for a new world. But such migrations have paid off for him in the past.

The Machine's Language

Simonyi was born in Budapest in 1948. The son of a physics professor, he fell in love at 15 with his first computer–a mammoth Russian Ural II in Hungary's Central Statistical Office. By the 1960s, the Ural, which received its instructions through cash-register-style keys and had a roomful of vacuum tubes to perform calculations, would already have been a relic anywhere else in the world. But Hungary's Communist leaders were trying to use the Soviet castoff to optimize rail and trucking schedules. The Ural wasn't up to the task: there was no way to input real-time data on shipments. "It was completely hopeless," Simonyi recalls. "It could have been done very easily by supply and demand. Unfortunately, that was politically incorrect."

But Simonyi didn't care. "I loved that computer," he says, "even though it was useless." As a child he had built an Erector Set car with a four-speed transmission–not so much because he wanted to play with it as simply to understand how it worked. A former student of his father's found Simonyi a job as the Ural's night nurse. Because the machine blew out a tube each time it was turned off and on, the Statistics Office preferred to allow it to run all night. Thus, from dusk to dawn, the mainframe was all Simonyi's; he had a personal computer before such things existed. He learned to program it by writing clever but useless routines to generate "magic squares"–numerical arrays in which the sums of the rows, columns, and diagonals all match.

understand. The method remains largely unchanged today, even if most programmers now use programming tools running on ordinary PCs. But on the Ural, Simonyi learned to program at a more primitive level, laboriously punching in the "opcodes" of machine language, specifying, instruction by instruction, the sequences of memory fetches, additions, memory stores, and jumps that the computer's processor had to follow to execute even the most trivial operation. It was (as Simonyi told author Steve Lohr in the 2001 book *Go To*) "Stone Age programming." Simonyi still remembers the codes. "Twenty-two is JUMP," he says today. "It's burned into my ROM."

Hungary in the 1960s, still flinching from the Soviet suppression of its 1956 revolt, was not a place for an ambitious young man with a taste for problem-solving. At 17, Simonyi landed an internship with a Danish computer company by showing some of its programmers samples of his hand-coded Ural programs. The Hungarian authorities expected Simonyi to return; he'd already won a coveted university spot. Instead, with his father's encouragement, he fled to the United States.

A letter of recommendation from Danish programming expert Peter Naur helped him win entry to the University of California, Berkeley. He paid the bills with a job at Berkeley's computer center, where he caught the attention of a faculty member named Butler Lampson. - Lampson was one of the leaders of the U.S. Defense Advanced Research Projects Agency's Project Genie–an experiment in time-sharing computer systems, in which multiple users sitting at terminals could share a single computer's brain time. When the Project Genie creators started a company, called the Berkeley Computer Corporation (BCC), whose purpose was to build a machine that would commercialize their work, Lampson recruited Simonyi.

At BCC, Simonyi would debug the company's balky prototype through the night, working with system designer Chuck Thacker. One night, Simonyi showed up in a see-through black outfit–"a kind of a hippie thing from one of the shops on Telegraph Avenue," he says. Today, he can't remember exactly why–coming from a party, perhaps? The debugging went especially well that night, and the outfit became a good-luck charm–Simonyi's "debugging suit."

BCC went belly-up after only a few years, but Lampson, Thacker, and much of the BCC team migrated to Xerox PARC. Simonyi–then just "a random Hungarian undergraduate without a green card," as he says now–joined them in 1972, laboring at Xerox while simultaneously pursuing his Stanford doctorate. Bob Taylor, who oversaw PARC's Computer Science Lab during part of that legendary era, says Simonyi's creativity stood out even in the lab's famous crowd: "He just could imagine ways of expressing code and ideas that put him off the charts."

It was a heady time. The team of visionary engineers was creating a series of innovations that would shape the next quarter-century of the PC era: the graphical user interface, networking (Ethernet), the laser printer, object-oriented programming (Smalltalk), portable computing (the Dynabook), and more. These breakthroughs all converged on a prototypical personal computer called the Alto.

The Alto was an amazing invention, but it wasn't clear what you could do with it until Simonyi get!" The name (reduced to the acronym Wysiwyg and pronounced *wizzywig*) stuck. Suddenly, Bravo had users: relatives and friends of PARC researchers began asking to use it to print school newsletters and format academic papers. Lampson's wife printed her thesis using the system, and when it was time for Simonyi to print his, he did the same.

Levels of Abstraction

Wysiwyg is an example of a layer of abstraction–a higher-level tool that allows computer users to ignore some lower-level complexity. Programmers use abstractions all the time. The text code written in a programming language is an abstraction of the machine code that a computer actually understands. A Web domain name is an abstraction of a server's numerical Internet Protocol address.

But most of the layers of abstraction in computer systems are less visible and more arcane than Wysiwyg. Ever since programmers stopped memorizing the opcodes that Simonyi used in his youth, they have been layering new abstractions upon older abstractions. Every generation of programmers uses its era's programming languages and tools to build the programs of the next generation. Layers of abstraction have accumulated like geological strata. Messages are constantly racing up from the binary bedrock of your machine and back down again, making it possible for a mouse-click to accomplish its function. Your mouse-click triggers some code in the operating system, which sends a message to the word processing program, which instructs the operating system to save your file to a hard drive. But that apparently simple process is possible only because of many, many layers of abstraction.

The history of software is the history of these layers, each of them lifting programmers farther from the binary, leaving them better able to coax computers into performing useful tasks. Steadily, programmers gained more power. But they were also tackling ever more ambitious problems. Programs ballooned in size, and programmers started getting lost in tangles of what they called "spaghetti code," which proved impossible to unravel and repair. Thus, large software projects became epics of frustration and delay. Program managers faced business problems like, How do you realistically schedule a project? How do you improve individual productivity? How do you coördinate complex work across a large team? Each of these questions proved surprisingly difficult to answer.

The difficulty of coördinating a team's work inspired software engineering's most famous dictum, known as Brooks's Law: "Adding manpower to a late software project makes it later." Frederick P. Brooks Jr. reached this gloomy conclusion after leading IBM's troubled effort to write software for its 360 mainframes in the 1960s. In his 1975 book, *The Mythical Man-Month* , Brooks observed that work proceeds slower on bigger teams because of "coördination costs"– the time programmers lose keeping one another apprised of their work.

This was the backdrop for Simonyi's 1977 dissertation, "Meta-Programming: A Software Production Method." Simonyi proposed a new approach to "optimizing productivity," in which one lead programmer, or "meta-programmer," designed a product and defined all its terms, then handed off a blueprint to "technicians," worker-bee programmers who would do the

Bravo itself was project B.

As the 1970s wore on, Simonyi grew impatient with Xerox's inability to turn PARC's pioneering research into successful products. One day a friend showed him VisiCalc, the new spreadsheet program for the Apple II. It thrilled Simonyi. Here was another application, like Bravo, that could change people's lives, but unlike Bravo, it ran on a mass-market computer that people could afford to buy. PARC's work, he realized, was never going to see the light of day. He asked his former PARC colleague Bob Metcalfe, who'd left the lab in 1979 to start 3Com, to recommend prospective bosses in the fledgling PC industry. At the head of the list was Bill Gates.

In 1981, Simonyi moved to Seattle to start the new-applications group at Microsoft, which until then had sold programming languages and operating systems. He was 33, but that made him a grown-up among Microsoft's striplings (Gates was then 26 years old, Steve Ballmer 25).

Through all the years that Simonyi oversaw the products that eventually coalesced into the "program suite" known as Microsoft Office, he continued to seek new efficiencies in new kinds of programming abstractions. Most notably, he schooled generations of Microsoft programmers in the discipline of keeping track of the myriad variable names used in big programs. In computer programming, variables represent information that can change as a program runs. For example, an online store's shopping-cart program will have variables that represent the number of items of each type to be purchased, each item's price, and the shipping costs and taxes. Using those variables, a programmer can write a simple line of code that multiplies quantity by price, adds shipping and taxes, and calculates the total cost–which becomes the value of yet another variable.

A large program can have thousands of different variables that a programming team must keep straight. Naming them carefully becomes crucial. Today, most code features variable names designed to convey meaning to the programmers who will read it–names like NumberOfItems or ShoppingCartTotal. In Simonyi's naming scheme, which he'd invented for his own use years before, every variable name comes with a prefix that tells you useful information about it, like its type (integer, say, or decimal fraction, or string of letters). Some systems limit the length of variable names to eight characters; Simonyi simply left out the vowels.

The resulting code was dense and hard to read. Simonyi's system came to be known as Hungarian notation, both in homage to its creator's birthplace and because it made programs "look like they were written in some inscrutable foreign language," according to programming pioneer Andy Hertzfeld. Hungarian is widely cursed by its detractors. Canadian Java expert Roedy Green has jokingly called it "the tactical nuclear weapon of source code obfuscation techniques." Mozilla programmer Alec Flett wrote this parody:

prepBut nI vrbLike adjHungarian! qWhat's artThe adjBig nProblem?

Hertzfeld, writing about an encounter at Apple with some Hungarian code written by a colleague who'd worked with Simonyi at PARC, said the names "looked like they were chosen by Superman's enemy from the 5th dimension, Mr. Mxyzptlk."

By the early 1990s, Microsoft's success had made Simonyi���s fortune. (For several years, *Forbes* has estimated it to be $1 billion.) But he still felt the tug of unfinished business. Software's confusion had made the creation of Office nerve-racking for Microsoft. But now, with computers more powerful than the Alto on every desk and the Internet linking them together, software's crisis was everyone's crisis. Simonyi began to think it was time to go meta again.

"Charles has always tried to build his systems in ways that raise the level of abstraction, so that you can manage the complexity of the system. Because complexity is death," says Chuck Thacker, Simonyi's old colleague from BCC and PARC, who is leading a research project on computer architecture at Microsoft. "And unfortunately, these days, providing the facilities people actually want results in a complex system. We're hanging on with our fingertips right now."

Moving to a position at Microsoft Research, Simonyi began to define the concept of intentional programming, or IP for short. Intentional programming would add an entirely new layer of abstraction to the practice of writing software. It would enable programmers to express their intentions without sinking in the mire of so-called implementation details that always threatened to swallow them. Like the "meta-programmers" of Simonyi's dissertation, passing instructions to worker-bee coders, the intentional programmer would hand off the scut work–but not to a junior colleague. Instead, intentional programming called for a sort of code factory called a "generator," a program that takes in a set of relatively high-level commands and spits out more-detailed working code. The goal wasn't so much to ease the labor of programming as to let programmers clear their brains of trivialities so they could actually be creative.

From his programming initiation as a teenager punching opcodes into the Ural, Simonyi had been climbing the ladder of abstraction. But he felt he wasn't high enough. In many ways programming still felt primitive. Why were programmers still saddled with incompatible programming-language syntaxes? Why was it so hard to extend their preferred languages into new areas? Why did programmers still work with plain text, arranging a small number of characters into linear strings as they had in the punch-card past? Simonyi's Wysiwyg work had liberated office workers to create and edit complex documents. Engineers and designers were using advanced CAD/CAM tools to design and modify blueprints for skyscrapers and airplanes. Why were programmers, the wizards who'd made all this possible, still pecking out their code one character at a time?

His Microsoft Research team got to work, and by March 1995 they had built a working system for constructing programs using the intentional-programming approach. Simonyi said IP had "achieved complete self-sufficiency": that is, "all future work on IP would be done using IP itself." He rewarded his team with T-shirts emblazoned with one of his favorite pictures from childhood: the image of Baron Munchausen lifting himself and his horse out of a bog by tugging at his own hair. Simonyi announced intentional programming to the world in a September 1995 paper titled "The Death of Computer Languages." It was time, as he later put it, "for the cobbler's children to get some shoes."

intentional programming, .Net was finished, and it required a less radical break from existing programming techniques. Simonyi itched to take his idea out of the lab and put it in front of customers, but that was awkward under the circumstances. He explains, “It was impractical, when Microsoft was making tremendous strides with .Net in the near term, to somehow send somebody out from the same organization who says, This is not how you should do things–what if you did things in this other, more disruptive way?”

Simonyi had been a company man for more than 20 years. But in 2002, he left Microsoft and launched an independent company. He walked out with a patent-cross-licensing agreement that let him use the concepts and ideas of his intentional-programming research but did not permit him to take any of his old code with him. He would have to start writing a new code base from scratch.

Under the banner of his new company, Simonyi dropped the word “programming” and rebranded his project as “intentional software.” The basic idea hadn’t changed, but now he began to stress the approach’s value to nonprogrammers. Simonyi’s pitch went something like this: Today, only the programmer is able to have a direct effect on the software. “Subject matter experts” or “domain experts”–the people who actually understand what the software needs to do, whether it is medical record keeping, corporate accounting, or climate modeling–can’t make changes to their tools; they’re forced to “submit a sort of humble request to the programmer.” Intentional Software would sell software development tools not just to programmers but to the domain experts who really knew their fields.

Intentional Software’s strategy borrows from a trend in programming known as “domain-specific languages” or DSLs–little programming dialects tuned to the needs of specific disciplines. Simonyi praises DSLs but says they don’t go far enough. They’re hard to create and therefore costly; you end up needing more than one (for a medical billing system, you’d need at least a medical and a financial language); and they’re incompatible with one another. Intentional Software’s system is like a factory for multiple DSLs that can talk to one another.

Here’s how it might work: Suppose an international bank wanted to develop a new system for managing transactions in multiple currencies. First, the bank’s own domain experts would define the system’s functionality, using their customary terms and symbols and identifying the most important variables (“time” or “value” or “size of transaction”) and the most common procedures (“convert holdings from one currency to another” or “purchase hedge against falling value”). Then the programmers would take that information and build a “domain specific” program generator that embodies that information. A separate software tool would allow the domain experts to experiment with different sets of data and ways to view that data as easily as businesspeople today rearrange their spreadsheets.

The programmer wouldn’t have to be summoned each time some new development in the world of international banking, or any other domain, required a new software feature. The customer wouldn’t feel straitjacketed by a programming language. Everyone would be happy.

Simonyi argues that his approach solves several of software engineering’s most persistent problems. Programmers today, he often says, are “unwitting cryptographers”: they gather accordingly. (For a more complete description of intentional programming, see " Intentional Programming Explained ")

In 2002 Simonyi assembled a new development team; today it includes a dozen programmers, split between Bellevue and Hungary. They began re-creating Simonyi's intentional-programming code from scratch and working with a handful of customers to test their assumptions and get feedback. A year ago, inspired by a new insight into how to present multiple views of heterogeneous types of data, they threw out a lot of their code and began again. "It's creative destruction," Simonyi says. "At Microsoft, it was fairly hard to do that, to throw away everything. But you have to abandon things that are difficult to extend."

ThoughtWorks, a global IT consultancy, is one early Intentional Software customer. But ThoughtWorks' CEO, Roy Singham, says that many of his colleagues at the company were initially skeptical of Simonyi's new project: "A lot of people look at this and say, 'Brilliant concept–but it's unimplementable.' So we asked some of our best technical brains to go look, and they all came back and said he's on the right track. Yes, it's hard. Yes, it's going to take time–maybe many years. But intellectually, he's got the thing nailed. It's the right problem to solve."

"I've felt some frustration that we haven't got something we can actually use in production yet," says Martin Fowler, chief scientist at ThoughtWorks. "Charles doesn't seem to be in a hell of a hurry to ship. But one thing to bear in mind is that he has shipped things in the past–quite dramatic things, with Office."

The visible fruit of Intentional's work to date is a nifty tool called the Domain Workbench, which stores a program's vital information in an intentional-tree database and then offers you many different "projections" of that information. In a demonstration Intentional gave at two conferences last fall, the Workbench–using a feature called the Kaleidoscope–took a series of code fragments and displayed them in a dizzying variety of formats. It didn't matter how the syntax of the code had been specified; you could view it, and change it, using whatever notation you preferred. You could edit your program as traditional bracketed and indented code, or switch to outline form, or make it look like a schematic electrical-wiring diagram, or choose something called a "railroad diagram," a kind of flowchart notation derived from old-fashioned train maps. Each view is a translation of the underlying tree–which you can also examine and edit.

Intentional Software's work provokes two main lines of criticism. Some theoretically minded skeptics say Simonyi's goal of capturing computer users' intentions is implausible. "How do you represent intent?" asks computer scientist Jaron Lanier. "As soon as we know how the brain stores information, maybe we can represent intent. To me it just seems like a fantasy." Another argument, common among programmers, is more practical. Many programmers love their text-based editors and distrust tools that distance them from raw code. As for graphical programming languages like Visual Basic and the integrated development environments (IDEs) that automate routine programming tasks, they regard them with condescension: such tools, they say, impose their own ways of doing things, constrain creativity, and keep programmers teeming coder forums. In part, that's because so few have seen its software. Intentional's work has proceeded with some secrecy.

When he started Intentional Software, Simonyi partnered with a University of British Columbia professor named Gregor Kiczales. Simonyi admired Kiczales's work on aspect-oriented programming–a way of organizing and modifying code according to "cross-cutting concerns" that resembles intentional programming. Kiczales, another veteran of PARC, has spent his career working on ways to "make the code look like the design." Kiczales saw joining Simonyi as a chance to further that end. But Kiczales trusted open-source development, where Simonyi did not. The Microsoft-style closed-shop approach simply didn't feel "organic" to Kiczales. "I would have done it in Java," he says. "The first release would have been in six months." The disagreement was friendly but irreconcilable, both men say, and before long, Kiczales had left.

For now, sheltered by Simonyi's wealth, Intentional Software has no target date or shipping deadline. But one of its two main customers claims to be close to deploying Intentional tools. Capgemini–a Paris-based international IT services and consulting firm that serves large enterprises and whose CTO, Andy Mulholland, is an acquaintance of Simonyi's–began working with Intentional last March and is considering using Intentional's system for projects in the European pensions business. The field's "very complex rules, intertwined with complex business domain structure," make Simonyi's approach look attractive, says Henk Kolk, Capgemini's financial-services technology officer, who is leading the firm's work with Intentional.

Ground Control

Simonyi's fascination with space has been lifelong. As a 13-year-old, he won a competition to become Hungary's "Junior Astronaut" and traveled to Moscow to meet a cosmonaut. As a new hire at Microsoft in 1981, he convinced cofounder Paul Allen to play hooky from developing the IBM PC's new operating system and fly to Florida to watch the space shuttle's first flight.

Simonyi's coming blastoff offers him a full-circle reunion with the Soviet-era technology that set his life's course. He has been training for months at Russia's Yuri Gagarin Cosmonaut Training Center in Star City, mastering the details of space suits and space toilets, and learning Russian.

The space trip will confirm Simonyi's status as that highly unlikely thing: a celebrity programmer. He has two jets and a pilot's license to fly them. He turns up in the tabloids as the frequent companion of homemaking's high priestess, Martha Stewart. He has built a 233-foot yacht with a wraparound glass-walled deck. He has funded an Oxford professorship for his friend Richard Dawkins, the Darwinian theorist.

None of this, of course, will make any difference in the outcome of Simonyi's quest to alleviate the chronic woes of the software field. "It's not enough to be a great programmer," Simonyi once told Michael Hiltzik, author of a history of PARC. "You have to find a great problem." Intentional might never deliver on its grand promises. But no one can charge Simonyi with a slight earthquake hit it," in the words of *New York Times* writer Patricia Leigh Brown, who marveled at its "hermetically sealed, mathematical precision" and found it "so vast that a visitor can feel like a lonely asteroid rattling around the solar system."

"[Only] Charles would build a 20,000-square-foot home with one bedroom," Simonyi's dissertation advisor and PARC colleague Butler Lampson once remarked. The lone bedroom boasts a cockpit-like control center that lets Simonyi tweak all his systems–heating, entertainment, telephone, lighting, and watering–to his satisfaction. "Like a submarine," he explained to Brown. "They all have to be green before you submerge." There's also a pivoting bed, which Simonyi can use to fine-tune his view–out across the lake; or over to the Seattle skyline, with its warrens of office workers wrestling with their documents and spreadsheets; or up into the starry night sky, where his latest journey will soon take him.

Scott Rosenberg is vice president of special projects at Salon.com. He is the author of Dreaming in Code.

Intentional Programming Explained
Simonyi and company are pioneering a push-button approach to programming.

[Click here for a diagram of Simonyi's planned approach]

Shane Clifford, a developer at Intentional Software, tells this fable.

Once there was a village with four parks, maintained by four competitive neighborhood associations. The first association decided to spruce up its park with a new bench. It solicited proposals from three of the world's leading bench makers. None of the designs won a majority of the neighbors' votes, so the association chose the most popular design. The process was democratic–but in the end, most were unhappy with the new bench.

The second association decided it wanted its own bench, but one that everybody liked. It found a manufacturer that built customized benches from mix-and-match parts. But the wood seat the members liked didn't come in the right length, and the decorative back didn't work with the green legs they liked. So they compromised on parts that did work together. The neighbors were proud of the finished bench, but no one sat on it very often.

The members of the third association saw how much money the first two had spent and decided they could do better. The craftsmen in the group asked everybody for suggestions, and in the end they built a simple, elegant bench that everyone agreed was the nicest in the village. Unfortunately, it wobbled dangerously.

The fourth association wanted a bench, too, but it didn't want to repeat the other groups' mistakes. The neighbors turned to a little-known bench maker who advertised "a new bench-making experience." The bench maker arrived with a flatbed truck loaded with odd-looking machines. He began to ask questions like "What is this bench's most important feature? What's the next-most-important feature? What materials do you like? What's your favorite shape for the bench's feet?"

disgorged a beautiful bench matching the final image on the screen. Everyone was glad to have had a chance to contribute, and many people sat on the bench every day.

To get a bench that makes everyone happy, you must build an automatic bench-making machine; help clients define their precise hopes for their bench; translate those hopes into instructions the bench-making machine understands; and then press the "Make" button. Clients get close control over the outcome, and bench makers, freed from the repetitive and mechanical parts of bench making, get to spend more time using their skills to feed their clients' wishes into the machine.

Substitute software for benches, Clifford is saying, and you'll understand intentional programming–so named because programmers are focused on the way their customers intend a program to work, and not on the clutter of code required to implement those intentions.

Intentional programming is similar in concept to what-you-see-is-what-you-get word processing programs, which Charles Simonyi, Clifford's boss, pioneered. "Wysiwyg" text editors let computer users manipulate a document's appearance on screen without forcing them to master the underlying code. Similarly, intentional programming encourages computer users to express their needs in their own familiar language, then shows them comprehensible views or "projections" of the emerging design before the executable code is assembled. It's not the only programming philosophy that relies on such graphical representations; the Unified Modeling Language (UML), developed in the mid-1990s at Rational Software (now part of IBM), also uses graphical diagrams to represent a program's function, structure, and behavior. But UML diagrams can't be transformed into finished software, which is Simonyi's dream for intentional programming.

Just how does Intentional Software hope to realize that dream? Let's put Simonyi's plan into its own diagram (click here). The software-building process begins, naturally, with the customer: any organization with an information-intensive task that needs automating. Simonyi calls the people at these organizations "domain experts"; they, not the programmers, know what the program should do.

With the programmers' help, the domain experts list all the concepts and definitions the software will need to encompass. All these definitions go into a database that Simonyi calls the "domain schema."

Like the bench maker turning his knobs, the programmers then incorporate the definitions in the domain schema into "domain code"–a high-level representation of the software's functions, expressed in a "domain-specific language," or DSL, that can be tailored to suit the industry in question. But while DSLs can vary, each action the software must carry out is stored in a uniform format, an "intentional tree." Intentional trees have the advantage of being visually simple but logically comprehensive, which means they can be manipulated, revised, and "projected" or reënvisioned at will.

For example, the computation represented by the simple program statement

```
Assign
(

a,
Div
(

b,
Plus
(

c,
1

)

)

)

)
```

Once encoded in tree form, the computation can be projected in many other ways that might be more familiar to domain experts, such as

```
b
return a = ——- ;
c+1
```

As their first concrete task, Simonyi and his colleagues at Intentional Software are working on building a special tool, the Domain Workbench, designed to manage these projections. Both the domain experts and the programmers use the Domain Workbench to edit and reëdit the projections until they look right. After that, the domain code is fed into a "generator"–the equivalent of the bench maker's truckload of machines–that churns out "target code" in a language such as C++ or Java that other computers are able to understand, compile, and run.

Once the target code is generated, it can't be turned back into domain code. In that respect, the generator is like an encryption program that irreversibly transforms plaintext into ciphertext.

However–and this is perhaps intentional programming's biggest advantage–it's easy to scrap old target code and generate improved code from scratch. Simply revise the domain code using the Domain Workbench's Wysiwyg editor and run it through the generator again. In most older approaches, even the slightest change in the original assumptions might require programmers to sift through millions of lines of code, updating every instance of a concept, definition, or computation by hand.

any intentional-programming project.

"Wysiwyg empowered millions of more users to author great-looking documents," Simonyi writes on the company's blog. "It is time to do the same for software users."

By Wade Roush

The Law of Leaky Abstractions
Excerpted from *Dreaming in Code: Two Dozen Programmers, Three Years, 4,732 Bugs, and One Quest for Transcendent Software* , by Scott Rosenberg, to be published by Crown Books in January 2007.

Software, we've seen, is a thing of layers, with each layer translating information and processes for the layers above and below. At the bottom of this stack of layers sits the machine with its pure binary ones and zeros. At the top are the human beings, building and using these layers. Simonyi's Intentional Software, at heart, simply proposes one more layer between the machine and us.

Software's layers are its essence, and they are what drive progress in the field, but they have a persistent weakness. They leak. For instance, users of many versions of Microsoft Windows are wearily familiar with the phenomenon of the Blue Screen of Death. You are working away inside some software application like a Web browser or Microsoft Word, and suddenly, out of nowhere, your screen turns blue and you see some white text on it that reads something like this:

A fatal exception 0E has occurred at
0167:BFF9DFFF.

The current application will be terminated.

Eyeing the screen's monochrome look and the blockish typeface, veteran users may sense that they have been flung backward in computer time. Some may even understand that the message's alarming reference to a "fatal exception" means that the program has encountered a bug it cannot recover from and has crashed, or that the cryptic hexadecimal (base 16) numbers describe the exact location in the computer's memory where the crash took place. None of the information is of any value to most users. The comfortable, familiar interface of the application they were using has vanished; a deeper layer of abstraction–in this case, the Windows "shell" or lower-level control program–has erupted like a slanted layer of bedrock poking up through more recent geological strata and into the sunlight.

(Perplexing as the Blue Screen of Death is, it actually represented a great advance from earlier versions of Windows since it sometimes allows the user to shut down the offending program and continue working. Before the blue screen, the crash of one Windows program almost always took down the entire machine and all its programs.)

In an essay titled "The Law of Leaky Abstractions," Joel Spolsky wrote, "All non-trivial complexity leaks back into their work. In theory, the handy new top layer allows programmers to forget about the mess below it; in practice, the programmer still needs to understand that mess, because eventually he is going to land in it. Spolsky wrote:

Abstractions do not really simplify our lives as much as they were meant to. … The law of leaky abstractions means that whenever somebody comes up with a wizzy new code-generation tool that is supposed to make us all ever-so-efficient, you hear a lot of people saying, "Learn how to do it manually first, then use the wizzy tool to save time." Code-generation tools that pretend to abstract out something, like all abstractions, leak, and the only way to deal with the leaks competently is to learn about how the abstractions work and what they are abstracting. So the abstractions save us time working, but they don't save us time learning. … And all this means that paradoxically, even as we have higher and higher level programming tools with better and better abstractions, becoming a proficient programmer is getting harder and harder.

So even though "the abstractions we've created over the years do allow us to deal with new orders of complexity in software development that we didn't have to deal with ten or fifteen years ago," and even though these tools "let us get a lot of work done incredibly quickly," Spolsky wrote, "suddenly one day we need to figure out a problem where the abstraction leaked, and it takes two weeks."

The Law of Leaky Abstractions explains why so many programmers I've talked to roll their eyes skeptically when they hear descriptions of Intentional Programming or other similar ideas for transcending software's complexity. It's not that they wouldn't welcome taking another step up the abstraction ladder; but they fear that no matter how high they climb on that ladder, they will always have to run up and down it more than they'd like–and the taller it becomes, the longer the trip.

Share Link

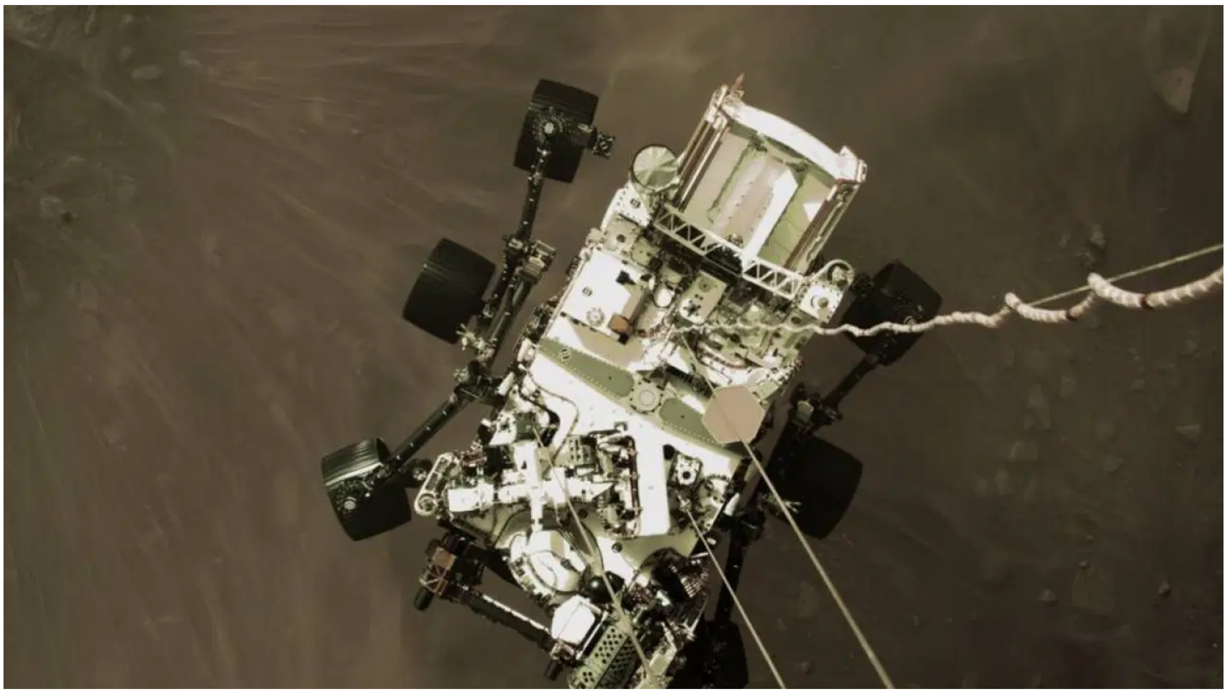
Tagged Intelligent Machines

Author Scott Rosenberg

Space Feb 23

Listen to the first sounds recorded from the surface of Mars

NASA's Perseverance rover delivered footage of its nail-biting landing on Mars, as well as the first ever sounds recorded from the planet's surface.



01.
This is the first image taken by NASA's Perseverance Mars rover. Now the hunt for life begins.
Feb 18

02.
NASA's Perseverance rover is about to start searching for life on Mars
Feb 17

03.
The 5 best pla
system—bes
Aug 17

Tech policy 10 hours

Announcing the MIT Technology Review Covid Inequality Fellowships

We're supporting journalism focused on the skewed consequences of covid-19—and how to change them.

You've read 1 of 3 Sign in Subscribe

Opinion 4 days

What we can learn from the Facebook-Australia news debacle

Democracies are right to look for creative ways to direct money from big tech to the news industry.

Biotechnology 4 days

A leaked report shows Pfizer's vaccine is conquering covid-19 in its largest real-world test

The Big Story: Geoengineering Feb 19

A first-of-its-kind geoengineering experiment is about to take its first step

Harvard scientists plan to launch a balloon this summer to test the equipment needed for the first geoengineering experiments in the stratosphere.

01.
Why geoengineering may narrow global economic inequality
Jan 13

02.
The odds of US climate progress just improved a bit
Jan 07

03.
Our midcentu
radical chang
Oct 13

SPONSORED

You've read 1 of 3 Sign in Subscribe

Humans and technology 5 days

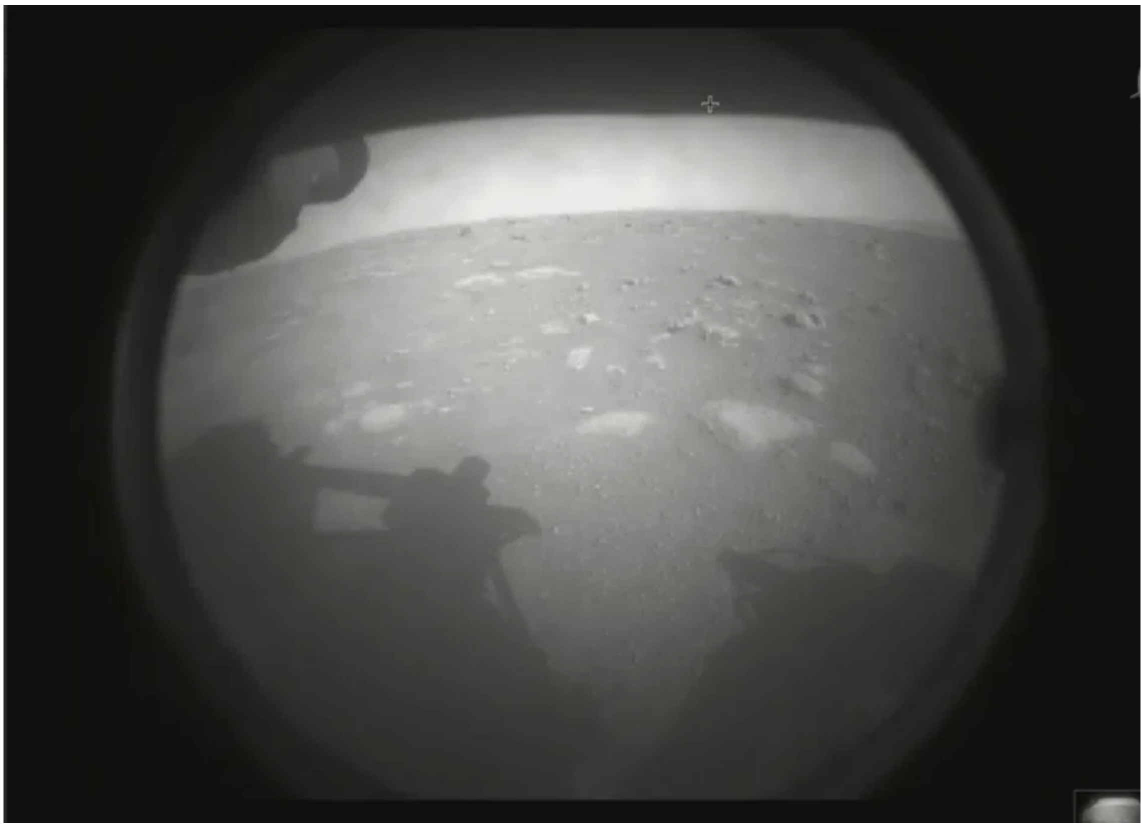
The government failed Texans—so people on the internet stepped in

But activists, working through a record-breaking freeze, say it shouldn't have to be this way.

Space Feb 18

This is the first image taken by NASA's Perseverance Mars rover. Now the hunt for life begins.

After surviving the descent, the rover sent back this picture from the Martian surface.



The first black hole ever discovered is more massive than we thought

shape the future of work.

Take the survey

Load more

You've read 1 of 3 Sign in Subscribe