The Cuban Software Revolution: 2016–2025

David M. West
transcendence corporation
profwest@fastmail.fm

Abstract

Presented as a work of fiction, this essay is about software development—how it has come to be what it is and what it might have been. The concept and metaphor of culture is used to frame the discussion. What might have been is presented as an approach named "Living System Design" and its practice in a fictional Cuba of the near future.

Categories and Subject Descriptors D.2.0 Software Engineering, General, D 2.2 Design Techniques, D2.10 Design

General Terms Management, Design, Economics Human Factors

Keywords living systems, agile, software engineering, software crisis, design

–Havana, Cuba / 2026

1. My Story

Writer: It was dark and cool inside the Bodequita.

The darkness, because not a single ray from the blinding Caribbean sun made it past the heavy curtain separating the bar from the entry vestibule and the street.

The coolness, ephemeral and mostly illusory—vestiges of early morning cold emanating from the walls and floor curled in response to the lazy twirling of the fans on the ceiling. Faint breezes, cool only in contrast to the broiling temperature outside caressed my face and arms.

Standing behind the curtain, I let my eyes adjust to the light thrown by a few incandescent bulbs on faux candles on the table and a bit of neon behind the bar.

Onward!'15, October 25–30, 2015, Pittsburgh, PA, USA
ACM. 978-1-4503-3688-8/15/10...$15.00
http://dx.doi.org/10.1145/2814228.2814247

When I got this interview I was told the bartender would know where he could be found, so I stepped up and asked *"Professor West, por favor."* A quick nod directed me to the wooden open riser stairs leading to a loft and eventually a second floor.

The bar was not half full.

Before the embargo was imposed, the Bodequita was a must-see tourist mecca—one of Hemingway's two favorite bars, they say. *My mojito in the Bodeguita del Medio and my daiquiri in the Floridita*, Hemingway's words, in his own hand, framed behind the bar.

Most post-embargo tourists have only a cursory knowledge of Hemingway, few have read any of his work, and next to none show any nostalgic interest in the master and his habits in his adopted home.

Only one of the tables on the mezzanine balcony was occupied. An old man sporting the long flowing hair popular in the 1960s, quite grey now, framed a face—its most distinguished element were the eyes.

"Professor West?"

West: Just Dave will do. Sit down. Mojito?

Writer: The bartender was already behind me with two fresh drinks on a tray. He set one down in front of each of us. Judging from the empty glasses at the side of the table, Dave would have gotten both were I not there.

"Thank you for seeing me."

West: No problem, although I confess I am not sure why you want to talk with me. You mentioned an article, The New Yorker or the Atlantic?

Writer: *"I'm hoping so. A story about you, your ideas, your life, an explanation for why so many in the phenomenally successful Cuban software industry credit you with that success."*

West: People have said that, yes; but aside from a few students who adopted my ideas, I am not sure how I am supposed to have had such influence.

Several years ago I was interviewed for stories in several business magazines, Fast Company and Bloomberg BusinessWeek come to mind, and I told them that my Cuban friends heard the exact same words and ideas as all of my American

students and colleagues—so any difference must be found in the Cubans, not in me.

Those writers recapped my ideas pretty well, but there was no outpouring of requests for my wisdom or assistance from the US or Europe!

Writer: *"Well they had different objectives. I'm looking not only for the ideas or your life stories, but the context—where the ideas came from, why were you the one that expressed them, that sort of thing."*

West: OK. Where do you want to start?

Writer: *"Two things: why did you bring your work to Cuba? And then, how did you get involved in software development?"*

Dave pointed to the wall beyond the end of the table.

West: See there? You have to look closely because it is kind of small—there wasn't much room even way back then.

Writer: Barely visible in the jumble of signatures, graffiti-like sayings in multiple languages, and crude caricatures was *Dave West 1994.*

West: Wrote that on my first visit here. Brought a felt tip pen especially for that purpose. Ever since Hemingway made this place famous, people have been writing on the walls. Papa's signature, of course, had lots of white space around it and was protected. All the rest of us had to make do with whatever space we could find.

I was here with colleagues from the University of St. Thomas in St. Paul Minnesota. I developed an object-oriented curriculum for the Masters in Software Design and Development program there and established the Object Lab in partnership with local businesses.

I held four seminars at the Polytechnic and two at the University of Havana. At the University I spoke about the philosophical presuppositions of computer science and software engineering.

Writer: *"What kind of subjects did you talk about?"*

West: Two examples. One, was tracing ideas about decomposition and composability—reuse, essentially—from Plato who, in his work Phaedrus, talked about the art of taking things apart not in the manner of a bad carver, and putting them together again, to Christopher Alexander in his book *Notes on the Synthesis of Form*, who spoke of clusters of forces with dense local connections and loose distant ones, to metaphors and the behavior-based decomposition of objects.

Another was on method and the kind of scientific management approaches adopted in computing combined with the ideas of Feyerabend and Gadamer and others. Gadamer especially. He wrote about the limitations of method, or, rather, the limitations of the perspective that any method imposes on an individual seeking knowledge.

It was clear to Gadamer that once a perspective is assumed—say a particular style of research in the natural sciences—then the kinds of questions that will be asked are predetermined, as well as the form the answers will take. Biases, prejudices, opinions—whatever one wishes to call them—come into play whenever a certain methodological perspective is assumed and with them, an inability to see the influences of that perspective—that's the problem with methodology. Gadamer believed that the human sciences of the nineteenth century were limited by their attempts to ground their studies in a method that mimicked the natural sciences. I made the same argument vis-à-vis computer science and software engineering.

Writer: The mojitos kept arriving just as Dave finished the previous one. The bartender knew his habits and desires. My ability to keep up was tested, and ultimately failed. At one point I subtly—I thought—made sure that one of my drinks was placed in the neutral zone dividing his and my sides of the table. Sure enough he consumed that one as well. But there was no affect on his speech or behavior—his capacity was next to astounding.

West: …Alcohol has never had much effect on me—other than making me hot and sweaty as my body turns the alcohol into sugar…

Writer: …he said when he noticed me staring at the empty glasses. At one point a pretty young woman joined us at the table and he introduced her as Estrelita, "one of my best students and favorite people."

West: The workshops at the Polytechnic were on object-oriented programming, at least officially. As I told the students, the unique ideas around objects had little if anything to do with programming. Instead it was all about design—the decomposition criteria you used to identify modules (objects), and how you distributed functionality across those modules.

If done correctly—using behavior as your object discovery criteria—you came up with designs "outside the realm of the structured design culture," as Grady Booch wrote in his 1991 book, *Object Oriented Design*.

The important thing was not what I said, but the circumstances facing the students at the Polytechnic. They were using old x86 PCs. The entire country had a single T1 link to the Internet. Of the fifty or so machines in any given lab, maybe thirty were functional, the rest in a state of repair or serving up replacement parts.

Simple, efficient, compact, reusable software was a necessity. They believed—and demonstrated—that the ideas I shared with them could produce software with exactly those qualities.

I was simply a conduit for ideas, ideas I believed in; but it was they who took those ideas and made something of them.

I had brought with me about twenty copies of Digitalk's Smalltalk/V. It didn't run on the machines available, but the students took my gift, researched how to write a Smalltalk VM, and wrote one that would operate on the hardware they had. They did a lot of work with Smalltalk in the next couple of years but, again because of limitations imposed by the embargo, were never able to make it a foundation for their work.

Writer: Estrelita, with a touch to Dave's arm, signaled it was near time to leave.

West: One more story, and one for the road, then please walk with us to my home. We can continue there with some friends and former students over dinner.

In 1968 I matriculated at Macalester College in St. Paul, Minnesota. Only 19 but I had a wife and a daughter due to be born in October, so I needed a job. One came my way —computer operator / reconciliation clerk at Northwest National Bank.

Nine of us worked in the computer room. Three keypunch operators, two computer operators, two programmers, a manager, and a computer technician. In fact, the technician was an engineer, a full-time employee of the Burroughs corporation—our computer was a Burroughs B100, with a console, a line printer, four reel-to-reel tape drives and a reader-sorter—and the technician was charged with keeping the hardware running. (It says something that the bank had to pay Burroughs $50–$60K a year to keep one of their electrical engineers on-site forty hours a week.) One of the programmers and I filled the role of "reconciliation clerk"—making sure that proof tape totals matched computer captured amounts for checks and deposits—an hour or so each evening.

I was the first person hired that was not a professional banker. The keypunch operators had been proof operators, the manager had been an operations manager, and the other operator was an assistant operations manager. The two programmers had been tellers. Average banking experience across the team was 10+ years. A requirement for keeping my job was to become certified in bank operations and banking principles.

Within six months, I was promoted to programmer, having learned both assembler and COBOL programming on my own. We also hired a third programmer, a young woman graduate of the COBOL programming class offered by the local vocational-technical college. We wrote two types of program: one similar to device drivers (in assembler) and the second, banking applications (in COBOL). COBOL, as a language, was exactly what the name suggests, a Common Business Oriented Language. Whatever its computer science merits as a programming language, COBOL's greatest value was the ease of expressing business concepts, processes, procedures, and information constructs.

The intent of higher order languages was to free the programmer from detailed attention to the machine and the complications of its operation. COBOL, like FORTRAN, was created for domain experts not computing experts.

The programming staff was, to all intents and purposes, bankers writing software. We knew banking and our particular expertise was our ability to "speak bank to the computers," i.e., to express banking problems and solutions in COBOL accurately enough that the computer did the right thing.

We did a lot of programming—in three years we collectively wrote close to 200 applications, mostly as individuals, sometimes in pairs.

The average program was 2–5 inches in length, referring to the size of the deck of Hollerith cards that was the physical embodiment of our programs. Code was written in pencil on paper. Keypunchers created punch cards from the paper sheets creating a deck of cards—one line of code per card.

Job control was, quite literally, the physical placement of various programming decks in sequence inside of three-foot long metal trays. Data, except for the magnetic resonance encoded (MICR) checks and deposit slips, was also on cards and would follow the program processing the data in those same metal trays.

Debugging occurred in three stages: 1) the correction of typos; 2) discovery and correction of syntax or logic errors based on compiler output—paper printouts of cryptic and often indirect error messages; and 3) core dumps. If a runtime error occurred the contents of core memory (all 64K of it) was printed in hexadecimal on fanfold paper 128 columns by 60 lines in a stack of paper, 300–400 centimeters tall.

The programming cycle consisted of coding on paper, keypunching, compiling, festering, and debugging. One cycle per workday was good luck.

But we were incredibly productive. It would take one programmer 2–6 weeks to create an application, test it, and put it into production. Often we could document a return-on-investment of more than 1000%. For example, I wrote a program that increased the efficiency of the reconciliation process that increased the number of checks we cleared each day by 20% thereby increasing the interest earned on those deposits by more than $1,000 per day—five days a week, 50 weeks per year: $250,000 per year. The program took me about three weeks to write.

Our experience at the bank didn't seem atypical. In a lot of ways the late 60s were a true golden age of application software development. But it was not to last. In August of 1968, a mere month before I joined the bank, NATO sponsored the first "Conference on Software Engineering" with the intent of finding solutions to the "software crisis."

Although for me those late 60s and also the early 70s were a golden age for software development, my nostalgia for them is not a motivation for what I will tell you over the next few days. That experience did, in a deep and pervasive manner, establish the perspective that affected the future development of my ideas.

It is true that we were naïve—uninformed by theory or deep understanding of what we were doing—and that we were simply automating well understood procedures in a well understood domain; but we were "doing the right thing" by focusing on understanding and improving an existing system and not attempting to create an artificial replacement system.

If development had continued in this "unselfconscious" (à la Christopher Alexander) manner we would have arrived at essentially the approach that I am advocating, that I taught in Cuba, thirty years ago.

Instead the problem was redefined in terms of creating artificial systems. My cynicism and skepticism toward the numerous development fads and revolutions I was to see

over the next sixty years is grounded in the conviction that all of those "advances" were focused on the wrong problem.

Writer: The transition out of the bar was far less dramatic than my ingress; the sun was just below the horizon, the breeze had shifted, coming off the ocean, just a hint of the cooling evening ahead.

I had assumed that Estrelita had joined us to help an alcohol-impaired old man make it home safe. Instead she held his hand, hugged his arm, and rested her head on his shoulder, while he gestured at buildings in old central Havana and pointed out landmarks.

West: A lot of these buildings had scaffolding and falsework when I came here in 1994. Castro had made restoration of the Cuban patrimony a priority. Alas, he had money to start but not finish the work. When the embargo was removed, the economy blossomed. More and more projects were started and it seemed every building in town was under renovation. Although money was plentiful, labor became scarce as workers focused on economic sectors that were growing faster than construction.

Writer: *"Yes, tourism grew by several hundred percent to account for about 10% of the overall economy. Agriculture tripled to account for about 12%, and medical services also tripled to account for about 15%.*

"What no one expected was the massive rise in software-supported systems and computer applications that grew from nothing to account for just over 40% of GDP.

"People here say that it was your ideas that made it possible for Cuban developers to dominate not only the domestic software market but all of South and Central America.

"Forgive me for being blunt, but outside of Cuba no one knows who you are, or anything about your ideas.

"What are those ideas? Where did they come from? Why do the only seem to work here?"

He sighed. Estrelita looked up at him with a warm, amused smile. She turned her attention to me with a look that said, "get ready for this." Obviously gathering his thoughts, Dave walked silently for a few blocks until we arrived at his home.

He lived in a three-story, circa 1880s, colonial façade building on the Malecon. A four-lane street—now busier than ever with both modern and classic (think 1950s) cars—separated his home from the ocean.

West: It is politically correct for me to live here only because the first two floors are classified as a school—a studio, really. Cuba is still Communist and re-purposing these buildings away from multiple family dwellings back to single ownership is frowned upon—both by the government and the people.

Writer: We climbed the stairs to the third floor and a waiting dinner on an open patio overlooking the ocean; he continued his story.

West: I was raised in Utah as a Mormon—this is relevant, hang on just a bit—and ages 12 to 18, I received about twelve hours a week of religious instruction. This is not unusual except for the fact that the laypeople providing that instruction were exceptional— all were church historians or church theologians.

Instead of the typical Sunday School children's lessons, my classmates and I got a healthy dose of metaphysics, Gnostic philosophy (this is why the Catholic Church in particular dislikes Mormonism), and morality based on the premise that we all were potential gods, and it was our responsibility to make that happen.

My interests and focus those years was science. I read everything written by popular science authors like Hoyle, Clarke, and Gamow, plus hundreds (I am not bragging) of monographs on physics and mathematics. In high school I fast-tracked my math education with a two-hour, four subject junior year (Trigonometry, College Algebra, Statistics, and Probability) so that I could take Calculus I and II my senior year.

Then, in my first semester of college—freshman year—everything changed. Physics I (mechanics) turned out to be deadly dull—so did the entire major curriculum. I abandoned it. Pure accident had led me to take a freshman seminar on the Bhagavad-Gita and Indian Philosophy.

By the end of the semester I had changed my major to Asian Philosophy—eventually focusing on the metaphysics of Buddhism, Taoism, and their offspring, Zen (which turned out to resonate perfectly with the Mormon theology I had learned as a teenager), and abandoned all but a curious, "what's new" interest in science and physics.

Science, math, and eventually computer science seemed, to me, to be too limited, to be focused only on easy questions—that is, questions that had demonstrable and provable answers. They were too simple!

John von Neumann (according to the recollections of John Alt), at the first ACM conference in 1947, captured an important aspect of what I was feeling:

> *von Neumann mentioned the "new programming method" for ENIAC and explained that its seemingly small vocabulary was in fact ample: that future computers, then in the design stage, would get along on a dozen instruction types, and this was known to be adequate for expressing all of mathematics….Von Neumann went on to say that one need not be surprised at this small number, since about 1,000 words were known to be adequate for most situations of real life, and mathematics was only a small part of life, and a very simple part at that. This caused some hilarity in the audience, which provoked von Neumann to say: "If people do not believe that mathematics is simple, it is only because they do not realize how complicated life is."*

I was interested in life, in the complicated—the complex, actually—and in the mystical, in altered states of consciousness, and always, in the human. I never shared the fascination with the computer, the artificial, the scientific and mathemati-

cal, even though I spent fifty years as a professional software developer and was a professor of computer science, MIS, and software engineering in various schools at various universities.

This choice meant I was soon left in the dust, lacking the ability to comprehend, use, or contribute to the theories of programming, and computing theory. This meant I did not publish in the right journals or get invited to the right conferences. I was a nobody.

Writer: *"Did that bother you?"*

West: Of course! I had things to say that I thought were important, that mattered, that could help. But I did not have the reputation to get them listened to. But, in time, the resentment went away as I saw how people, like Terry Winograd, who had the rep, got tenure at Stanford, and yet, when his ideas drifted from "pure" computer science, they were mostly ignored. It wasn't personal, just part of the culture.

But much of what I lacked in the area of computer science and programming technology, I possessed in the area of software development—not software engineering, just development. I did become a tenured professor—at one point in the world's largest software development masters program (900+ students) at the University of St. Thomas, presented over 50 papers at refereed conferences, and wrote three books. But I never felt part of the in crowd or influential, except in the case of some of my students.

Writer: He rolled a now-empty glass between his palms.

Crisis

West: Culture is key here—observing and understanding culture.

After twenty years in software—from operator to IT director—I returned to university as a student. Between 1985 and 1988 I completed three degrees: MS Computer Science, MA Cultural Anthropology, and PhD Cognitive Anthropology.

I took anthro courses because I never, in that twenty years, had a system I developed fail for technical reasons; always something social, psychological, or political—something cultural—got in the way.

Anthropology taught me how to be an anthropologist, to take advantage of my natural position: on the outside, looking in. I became a participant-observer: in the world of computing, but not of it. Ignorant, unfortunately, of every detail, every nuance understood by an insider, by a member of that culture; but with the advantage of perspective, a vantage point yielding insights.

I'm going to give you what Clifford Geertz calls a "thick description," an ethnography of a very real computing culture. You see, I'm not a historian or scholar any more. I just want to tell you what I saw during my career, and how that contrasted with my own ideas.

Writer: *"You mean insights?"*

West: Actually, I'll tell you stories that illustrate those insights and my ideas.

Writer: There was a pause as dinner concluded and we adjourned to the balcony for coffee and cigars. About ten others, mostly young, about half and half male and female, had joined us for dinner and stayed to listen to Dave's stories.

We had strong Cuban coffee (mostly for the guests, I think) followed by glasses of Coca-Cola made with real Cuban sugar with lime wedges (mostly for Dave). Fine Cuban cigars—Cohibas and Bolivars—for almost everyone.

West: Never smoked until I came to Cuba, and have only smoked Cuban cigars when I visited and occasionally since I moved here.

I want to tell you about the emergence of a culture—a group of people sharing values, worldview, technology, and behaviors. Like any culture, it is not homogenous. Outliers and dissidents exist. Culture is often invisible to those who share it, but its power over their actions and their rationalizations for those actions are very real and very powerful. It may seem as if I am stereotyping people in that culture but, except for details, that stereotype is in fact quite accurate.

The roots of this new culture is Western Rationalism, the Age of Reason, Descartes and Leibniz et al., and the circa 1900 physics of Mach and Einstein.

With few exceptions, this culture imposes, often invisibly and unconsciously, its worldview, its values, its traditions on everyone whose work is even remotely associated with computing and computers. In fact, obtaining any professional applied computing title—programmer, systems analyst, user experience designer, etc.—depends on successful navigation and enculturation in the academic computer science culture.

Part of this culture emerged with the first academic programs in the area of computational science. The part I am most focused on is the nonacademic profession, the applied developers who work in corporations, governments, and organizations of all types. This is the largest segment of the culture—as much as 75% of all those in computing work—and came into existence, essentially, beginning in 1968 with the proclamation of a "Software Crisis."

Mariel, please show our guest that clipping on the wall over there.

Writer: Mariel handed me a framed page from the November 8, 1966, issue of Business Week. The headline read: "Software gap—a growing crisis for computers." Under that the words, "Shortage of programmers—and the fruits of their solitary art—is stunting growth of computer use and costing the industry hard cash."

West: Note the second paragraph where it states that programming is a new human intellectual art, not a mechanical or electronic skill. And lower down where it describes the software crisis as a lack of programmers sufficient to meet demand. The crisis was the growing divergence between demand (every business, government, and army was desperate for software) and supply (knowledgeable and capable software developers). In 1968 the large majority of programmers were re-purposed domain experts; e.g. scientists, military officers (think Admiral Grace Hopper), or business experts.

Permit me a brief aside—some of my many critics complain when I say this about Grace Hopper. Admiral Hopper was selected for the Mark I and Univac projects because of her PhD in math and expertise in computer languages and compilers. Her selection to head the COBOL effort, however, was grounded in her domain expertise in Pentagon procurement and management policies and procedures.

Back to the main point: the pool of domain experts as potential developers was essentially dry—no more were available without negatively affecting the domain, e.g., the business and its normal operations.

Quality and waste were not real issues, yet. Although much of the software in the late 60s and early 70s was, by today's standards, inefficient, murky, and poorly conceived; but even the worst software was usable, cost very little to produce, and was easily replaced if it was intolerable. The reason for variety in quality was individual differences among developers.

Given that the crisis was a lack of capable programmers, why was the answer "software engineering"?

That 1968 NATO Conference on Software Engineering: those invited to the conference were scientists, mathematicians, engineers, and generals plus academics from the few extant computer science university programs—those academics were almost exclusively mathematicians and electrical engineers by background and training.

The Business Week article described programming as a "human intellectual art." Why wasn't the NATO conference named "Software Artistry?"

I was not privy to the discussions leading up to the NATO conference, nor did I have access to academic faculty in computer science. However, I did read the business press and later reports and papers from the NATO conference and others. There was debate of art versus science but science prevailed and art lost.

Why? First, the volume of programmers needed was huge. No one had experience with the mass production of artists. Since the end of World War II, however, the entire higher educational system was retooled to mass-produce scientists and engineers. Sputnik put even more energy and money in that endeavor.

Second, there was the cost efficiency of one professor teaching a subject like math—with its discrete, right and wrong answers—to 500 students in one lecture hall; as compared with the arts faculty with a studio restricted to ten or so students at any given time.

But the real reason that art lost, I believe, is because of a basic misconception of what was involved in writing a program.

Reflecting on the work of my colleagues in 1968 it was clear that programming had two distinct aspects: deciding what the program should do, and how the program should do it. Doing the first part well requires a deep understanding of the problem and the domain along with insight, gained from experience. This is art or at minimum, craft.

If, however, you assume that the "what" of a program can be articulated in nothing more than a set of imperative requirements, then programming requires nothing more than knowledge how to provide instructions about how the computer is supposed to satisfy those requirements.

Writer: *"Requirements?"*

West: A detailed, discrete, accurate, complete, and unambiguous description of the conditions and capabilities needed to achieve a desired objective. The programmer simply decides what algorithms, control statements, data structures, and error handling to employ to meet those conditions and provide those capabilities. That is, satisfy requirements.

Generalized, this view of programming became what Herbert Simon called "the sciences of the artificial." Simon's views did not establish the mindset of this emerging computing culture; he simply provided a concise and concrete articulation of an already dominant worldview. Computer scientists, especially academics, were obsessed with the precise and clean world of mathematics, occasionally expressed in the form of electronic circuits, and the physics of transistors and magnets. Herb Simon argued, in *Sciences of the Artificial*, that both the computer and the world were deterministic machines operating in accordance with mathematically precise principles—sets and function types.

If you could (Simon and physicists like Mach stated that this was, in principle, true) specify the current state of such a machine and a desired state, you could define with mathematical precision the exact set of transforms necessary to turn requirements (specification of desired state) into an executable set of instructions necessary to ensure the machine would, in fact, be in that desired state.

Moreover, the transformations could be determined and stated by a machine operating according to mathematical laws of its own. Automatic programming!

At the same time, roughly, the business community contributed a misbegotten concept of its own, "scientific management." Grounded in a misanthropic assessment of the working class as "lazy, dishonest, and prone to mistakes," scientific management believed in the necessity to define in precise detail exactly what each employee should do to accomplish any and every task that came their way and then force the employee to comply with those instructions. Further, tasks would be decomposed to the point that each worker would be responsible for one task repeated over and over again for a full 8-hour shift.

The caricature of programmers in those days was "Twinkie eating, Coke swilling, basement dwelling, socially inept males afraid of girls." Who more might a manager wish to subject to the dictates of scientific management? (Few of my

colleagues in those days would recognize themselves in that caricature. This is an example of being blind to the culture within which one is embedded.)

Writer: The full moon, risen rather spectacularly from the bay, was approaching mid-sky. Neighbors on a nearby roof top patio were enjoying a three piece band by dancing Cuban folk dances. Most of the kids had drifted off. Estrelita, with others, had cleaned up. Estrelita only was still present.

West: Enough for now. You want to continue tomorrow?

Writer: *"Very much!"*

West: I'll send a car for you around 15:00. We can talk more or you can listen to more stories on the beach as we watch the young, beautiful, and rich (the new class of Cuban youth and international tourists) frolic in the surf.

Someone will be waiting for you at the front door to take you to your hotel.

Buenos Noches.

2. The Illogic of Business

Writer: I spent the next morning trying to get a sense of the Cuban people. I was looking for something, some character trait, some cultural bias, some aspect of 'spirit' that might explain why Professor West had met with such success here and not in his homeland.

Riding a "camel," a public transport vehicle based on a flatbed semi truck, I noticed that 1950-era autos still dominated traffic. The expectation had been that all those vintage cars would have been sold to collectors in the US. Dealers outside Cuba also expected to make a killing selling old parts. Neither happened.

During the fifty-year embargo, the Cubans learned self-reliance, determination, and, above all, ingenuity. They also developed a healthy suspicion of and antipathy toward the US. Not toward the people, of course, but toward government, banks, and large corporations—including all those parts hoarders and speculators expecting to make a killing at the expense of the 'naïve Cubans.'

They also had a fifty-year window where they developed healthcare systems—like medical information, medical treatment, and medical education—free from the dominance of the insurance and pharmaceutical companies that shaped the US system.

When it came to software, the Cubans had spent fifty years figuring it out for themselves. They lacked the investment in legacy systems and technology lock-in— think relational databases—that constrained thinking in the US. They did not have the asset base necessary to do much, but what they did develop was inexpensive, effective, simple and lean.

Estrelita was my driver from the hotel to a beachfront restaurant at the resort of Varadero (once the site of the only golf course in Cuba). I took the opportunity to make a discrete 'I thought' inquiry about her relationship with Dave.

"Oh you silly Americans," she laughed, "Dave is old, you know!"

West: Buenas tardes. Pull up a chair and have a daiquiri. We honor Hemingway again, in his choice of afternoon drink, if not his favorite drinking establishment.

Writer: We met on the veranda, an extension of a bar attached to a restaurant, which was attached to a high rise hotel, via a path winding among tropical palms and flowers, animated by birds and turtles. The daiquiris, like the mojitos the evening before, were bottomless.

West: Tomorrow you will see and hear what is being done in Cuba. You will be able to judge for yourself what I may or may not have contributed.

Today, we must return again to the past and how one culture, of which we have already been speaking, evolved and another emerged.

I must remind you again that every culture has variations and degrees. Not every Frenchman can cook gourmet meals, and not every American is a crude lover.

Descriptions of cultures, ethnographies, are essentially true but sometimes false in the particulars. Not false really, more like incomplete or with missing nuances and glossed-over exceptions.

Before 1968, and for a while after, both the world of business and the nascent world of computer science were in thrall to the circa-1900 worldview of physics.

When the "what" of programming was separated from the "how," business was given the task of specifying what was to be done, via a set of specifications or requirements, and programmers were assigned the task of articulating the how. What had been one culture gradually began to schism.

The computing culture we have been discussing focused inward, expanding its theory, honing its worldview, reducing behaviors and rituals to formal methods, etc. Simultaneously it began a kind of virtual conquest—of empire building—by assuming within itself every activity even remotely associated with a computer.

In the late sixties it was actually hard to define the 16–30 courses expected of a CS college major plus another dozen or so courses for each graduate Master's degree and roughly the same number for the PhD program. Technological advances soon made it easier as a variety of specializations—for example, communication protocols, network configuration, packet based communications with associated problems of timing and assembly—expanded the realm of computing.

Adding to this was a kind of academic greed, "if it involves a computer, it is ours!" Today ACM recognizes five discrete subdisciplines (computer science, computer engineering, information systems, information technology, and software engineering) within computer science; three of which have marginal connection to the original scope of computer science.

Only Management Information Systems (MIS) was able to escape and become the purview of a different department—

Business, usually. Software engineering should have escaped too, or really it should have been divided into two areas: the engineering of virtual machines like compilers and device drivers, and the art of application development. But that did not happen.

As a consequence, students no longer had time or opportunity to learn about anything outside of computer science.

By the mid-1970s the typical IT shop contained few domain experts. Essentially, one hundred percent of the employees were professional computer scientists, software engineers, or academically trained programmers. And this was considered a good thing—developers did not need to know anything about the domain in which they worked. In fact, it was necessary because, as professionals, they would seldom be working in the same domain for their entire careers. Like hired guns, they would move to where the needs were, and clean up whatever mess they encountered.

Computer science had the luxury, until recently, of never encountering systems that challenged their innate belief system. They focused on the computer and on computing—nothing but artificial, deterministic systems—and their belief system held firm.

Business, in contrast, faced failure after failure when trying to apply theories based on the deterministic, scientific, worldview. Business learned—kind of—from these failures. They started to look to something other than science to find answers to their problems.

As early as the 1980s the business press was filled with ideas about natural systems, like biotic ecologies.

Tom Peters wrote *Thriving on Chaos* in 1991 and introduced a new idea based on complex systems subject to rapid change. Pelle Ehn, in *Work-Oriented Design of Computer Artifacts* in 1988 asserted the need to consider computer and software artifacts as embedded in complex systems. Ehn specifically addressed the work of Herbert Simon (*Sciences of the Artificial*) and argued it was an inadequate foundation for understanding systems involving people and socio-political systems. Dee Hock in *Birth of the Chaordic Age* argued that firms thrived when operating at the boundary of order and chaos, and presented a detailed case study (VISA International).

Christopher Alexander was a significant source of inspiration for the 1968 NATO conference because his first book, Notes on the Synthesis of Form, held out an approach that would support a program of formal, mathematical design. Alexander, however, turned away from the ideas expressed in Notes, to those found in Timeless Way of Building and Nature of Order.

Although the first adopters of Alexander's ideas were associated with the world of computing—specifically object-oriented approaches—much of the patterns community today is focused on organizational patterns, educational patterns, social change patterns, and even patterns for beauty, earthquake preparedness, social cooking, and dealing with dementia.

Problems stemming from the highly dynamic nature of the enterprise, and the socio-economic context in which it operated, came to dominate business thinking. Change, change management, adaptability, and innovation were the problems and issues at the forefront of business since the 1990s.

Tellingly, business did not look to computing or IT for assistance in dealing with their new problems. In part this was simply because of the adversarial relationship that had come to exist between them, but in part it was a widely shared perception that computer scientists and software engineers with their formalist mindset had little to offer.

Instead they turned to the design professions, particularly graphic, product, and industrial design—thinking these practitioners had the skill and the knowledge necessary to understand and work with complex systems.

Unlike deterministic systems, complex systems are not predictable. It is not possible to measure, or even to know, all relevant variables, and relationships are integrated with context and therefore lack the abstract universality of physics or computation laws.

Designers (architecture and the applied arts like graphic, product, industrial, and interior design) constitute their own culture, with a worldview, concepts, practices, and technology that came into being specifically to address problems that are malformed, ambiguous, deeply context sensitive, and "wicked."

Business found it very easy to align itself with the culture of design.

The business relationship with IT continued to degenerate. First it was just adversarial—who could blame whom when systems inevitably failed—then it escalated to labeling IT a commodity, to the belief that IT was the single biggest barrier to the realization of enterprise goals like agility (adaptability in the face of rapid change), innovation, and sustainability.

Notably, one part of business did not share in this cultural shift. The IT department had yet to escape the clutches of scientific—i.e., antihuman—management and were easy prey for vendors and academicians pushing solutions in the form of formally defined methods and automated tools that embodied and enforced those methods.

Immense effort was put into developing automated tools, computer assisted development tools that would, eventually—it was believed—replace human developers entirely. Automated code from abstract pictorial specifications! Huge frameworks like the failed IBM Repository and the wildly successful—in terms of sales, not results—SAP.

Another aside: to be fair, the motivation for things like the IBM repository was not primarily automated code and elimination of human programmers. One big issue was reuse. Numerous efforts to achieve reuse, at the level of code or modules, failed. Few recognized that the failure stemmed from framing the problem in the context of computers and virtual machines—where there are many ways to do the exact same thing, but each of those ways is deeply embedded in a

context that includes the idiosyncratic "style" of a programmer. Reuse in the real world is a solved problem and could have been a source of inspiration, but was not.

Advocates of these formal approaches were primarily academics or were housed in vendor companies. Those forced to use them were, increasingly, the product of an academic, formalism-oriented education and enculturation process: the perfect conditions to usher in a golden age of software development.

Writer: Pausing briefly, Dave switched drinks from daiquiris to coke with lime wedges. Estrelita emerged from the surf like Ursula Andress, toweled her hair, and slipped into a beach wrap. She joined us at the table.

West: It did not happen. The tools and the methods were not useful for those charged with using them. People do not work that way, and there were so many implicit assumptions and constraints imposed by the tools that they inhibited creative thinking.

Much of what was learned in school turned out to be of little use on the job. In the 90s and 00s, large consulting companies put their recruits, all from top computer science programs around the US, through two-month "boot camps" in practical development before they were considered "billable."

Practitioners found themselves between two conflicting cultures: CS convinced about "how things should be done" and business equally convinced about "how things really are done".

3. Au Contraire

West: Practitioners are paid by organizations that see themselves as complex, dynamic, essentially organic systems charged with solving the kind of ill-defined, ambiguous and wicked problems that arise from complex systems. At the same time they are governed by an IT management complex that, fundamentally, does not trust or respect them. Making things even harder are all the unconscious, cultural presuppositions picked up in college.

Being capable and smart people, practitioners found solutions to their problems. Sometimes by modifying and extending what they had learned in school. Most structured methods—excepting Dijkstra's structured programming—were defined by practitioners like Larry Constantine, James Martin, Ed Yourdon, Michael Jackson, and Fred Brooks. At other times, practitioners staged pseudo-revolts against the status quo; Objects and Agile are exemplars.

Contrarian views were expressed by those deep within the computing culture, like David Parnas's "The Rational Design Process: how and why to fake it," and Peter Naur's "Programming as Theory Building." More recently, Christiane Floyd and her colleagues deconstructed software development in their book, *Software Development as Reality Construction*.

These innovations never had legs. Despite the fact that they could demonstrate with tons of empirical evidence, their superiority to the "official" CS-derived approaches, they seldom had impact beyond one or two years. None had any enduring effect on the practice. And none gained any credence within academia.

Innovations from the practice tended to follow one of two paths: the truly innovative were discarded quickly—behavior-driven object design and Naur's theory building come readily to mind—or they were revised and assimilated.

Extreme Programming is an obvious example of this latter fate. XP rapidly gained too large a foothold to be flung aside and entered the event horizon of CS/SE. XP was first relegated to the "kiddy table" of the software development feast—deemed suitable only for small, non-critical, unimportant projects. Then each of its practices was redefined in ways that removed any vestige of difference from an existing SE practice (e.g., user story to requirement card). Then XP was encased in the straitjacket of project management (Scrum) and the shackles of production processing (Lean), and redefined as Agile. The nail in the coffin occurred when Agile became "CMM consistent" and "Six Sigma compliant."

If the clichéd man from Mars visited software development shops circa 1970 and again in 2010, the differences would be almost indiscernible. Then they spoke COBOL-ish and today they speak Java-nese: that's it.

Writer: *"Excuse me for interrupting this fascinating and thorough history lecture, but when will you start telling me about your ideas and what is happening in Cuba?"*

West: Patience, we're almost there.

Ideas from the more academic contrarians were noted, but seldom taken seriously. Those from the practice, however, were lost as much to sabotage from the proposers as co-option by the computerists.

Agile is perhaps the best example.

Extreme programming had some great ideas, some genuine insights. "System Metaphor" was one, but no one understood it and so Beck abandoned it, at least as an official practice. "User Stories" were not developed (in fact, I wrote the book chapter that offers the most comprehensive explanation) and little, if anything, was said in their defense as they were co-opted and reduced to verbose requirements.

Most egregious of all was the failure to recognize constraining presuppositions like the assumption that all work should be organized in terms of projects and the assumption that programs would always be so large and complicated that testing was the only way to understand, document, and assure the integrity of programs.

Similarly, the Poppendiecks failed to see that most of Lean makes sense only if one assumes the very production model of development rejected by Naur; and Scrum recapitulates scientific management, even to the point of using the tired metaphor, Scrum "master."

Writer: Sun set, our party had moved to a patio with an open fire at its center. A pig had been roasting there the entire day. At the periphery were pots of beans and rice and vegetables. Our party now numbered close to twenty, sitting

at tables arranged to form a shallow arc, Dave at the center table. I was across from him, my back to the fire pit.

West: Remember, I am like an anthropologist, living and working with the tribe; sharing their work, living their lives, but never truly one of them. I see things about them that they do not see, or do not admit seeing. I have my biases, of course, but do my best to set them aside and report in as objective a manner as possible.

OK, pretend it's 2015: CS can point to trillions of lines of working code defining the infrastructure of the world and everything that has become dependent on that infrastructure. We do not worry much, anymore, about the computational integrity of our machines (computers and peripheral devices). The code behind our communication systems and the Internet is a marvel. Much of it is, at least officially, the product of software engineering.

From this success has come a kind of hubris—if it worked for us here, then it will obviously work for everyone, everywhere. All they need to do is master "computational thinking" and the mathematics and logic that underpin that thinking.

Empirical evidence suggests otherwise. Despite all the arguments against its methodology and quibbles about its definitions and measurements—the *Chaos Report* from 2014 is a very telling indicator of the success of applications.

Over six decades the percentage of systems delivered on-time, on-budget, with complete functionality, and that are still useful 2–5 years after they were conceived remains fairly constant—about 10%–15%.

Anecdotal evidence abounds, but except for the much maligned *Chaos Report*, few scientific studies of failures have been reported. We do not need scientific studies of failure, but we could benefit greatly with ethnographic studies of failure. This is because failure is systemic and not reducible to quantifiable factors and formulaic relationships.

Remember, it's 2015: imagine you're flying across the US on a commercial jetliner, 30,000 feet in the air. Take comfort: the software monitoring your flight, providing instructions to your pilot, and preventing collisions with other aircraft was written and put into service roughly fifteen years before Texas Instruments released the first Speak and Spell toy. The system is called Host and it was written in the 1970s.

At least three attempts—costing between $400 and $800 million dollars each—have been made to replace Host. The first two ended in failure and the money was written off.

The third attempt, called NextGen, is five years late and at least $500 million dollars over budget. In theory it will be deployed sometime this year, 2015. It has less capability and less accuracy than the Google Maps app.

In 2013, the US Government spent $400–$500 million dollars to build a Web site for health care. In 2014 it spent almost the same amount to a second company to fix and maintain that same Web site. The site itself handles, on an annual basis, less than .03% of the number of transactions processed by Amazon.com in 2011.

And I lurked in the background. My own ideas were taking shape. They are grounded in the metaphysics and ethics I learned as a youth in Utah (all that Mormon stuff) amplified by the metaphysics and epistemology of Buddhist philosophy. I see wholes. I apprehend—not analyze—reality. I think I see all the "natural joins"—the areas of loose coupling—among system elements and the dense clusters of resolving forces that Alexander captured in his diagrams and later called "patterns."

I am convinced of the communicative power of stories—and only stories. I believe that metaphor, not logic, is the fundamental property of thought.

My faith is in the unique abilities of human beings and their magnificent minds. The idea of an artificial mind is an absurdity in my worldview. By extension, I believe in the ability of human beings to work together collaboratively and cooperatively.

This ability requires humans willing to make the commitment to be polymaths. If you want to be a god, as I was taught, then you must come to know everything about everything—and through experience, not just via books. The term "modern polymath" was just coming into vogue in business literature when I came to Cuba. It means that everyone, especially everyone on a development team, must be a broken comb—with at least two deep areas of specialization (the ends of the comb) and multiple areas of specialized ability in differing degrees (the broken teeth), all connected by the thick backbone of the comb representing an ability to move among specializations and weave together points from each into a coherent whole. This whole can be communicated among all those with similar expertise and experience profiles.

Yeah, 2015. That was the year I snapped.

Writer: Dave's voice had been growing in volume and timbre the last few minutes. He gazed about the table as if challenging those around him to challenge back. There was no challenge, no obvious disagreement, just nodding heads. His voice returned to normal as he closed out the evening's tales.

West: You see, I am still angry, or maybe mad. I am certain my colleagues in the US would see me as such. Some of my former students might not—I did have some influence.

My ideas are incompatible with software development as it is advocated by the computing culture and mostly with how it is practiced.

During the decade and a half that I argued on behalf of object ideas, an increasing amount of that time focused on why everyone was doing everything wrong. The objects of UML had nothing to do with objects as envisioned by Alan Kay and his contemporaries. Object decomposition and design (based on understanding and distributing behavior) was essential; OO programming (based on UML-defined data structures and algorithmic methods) was irrelevant! Similarly with Agile: telling people that they completely misunderstood Agile as a method instead of a culture and stories as computer requirements instead of domain narrative was not popular.

Mine was a contrary interpretation of contrarian ideas.

But, tomorrow you will see for yourself. We meet at my home; we work with others in the studio.

4. Living System Design

Writer: I arrived at Dave's Malecon home promptly at 8:00 AM. He and Estrelita were on a small balcony on the third floor. He motioned me up. Passing the first two floors, I witnessed an abundance of activity, most of it the "olas," unpacking, and friendly updating that takes place among any group arriving for work or school. A few groups were already standing at whiteboards or gathered around computer terminals—this was a software development facility.

Estrelita met me at the door to the third-floor apartment. She was professionally dressed and handed me a cup of coffee while gesturing the way to the patio.

West: Buenos Dias! Buenos Dias. Come, sit, and enjoy the coffee. I wish to tell you one more thing, give you a metaphor that you can use to frame what you see today.

Consider a human being. It's a complex system, densely interconnected, highly dynamic, and constantly changing, evolving. Unlike the twinkling Little Star, a typical human being is often less than optimal, always it can use some improvements, some corrections. This is your task—make a better human being.

For those in the computing culture this is a daunting task. For them, a human is nothing more than a vastly complicated machine and their task is taken to be the creation of an artificial human; something built from electronic circuits and mechanical (perhaps in some instances including discrete bits of biological) stuff operating according to laws—of physics, math, and logic.

They are being true to the ideas of Herbert Simon who in his *Sciences of the Artificial* asserted that self-organizing complex systems with emergent properties are simply not-yet-understood, deterministic systems.

Years back, a group from the Software Engineering Institute did a study for the US government on Ultra-Large Scale systems. They described such systems as qualitatively different from the systems with which computer science (CS) was familiar and with which CS has had so much success. The initial—almost knee-jerk—reaction of some traditional computer science faculty was to deny this qualitative difference and instead treat ULS as systems of systems, all of which were deterministic (technically, top level systems are only statistically deterministic). Later, this work was assimilated into CS dogma, at first by explaining that the "internet of things" was indeed different from systems of systems, but only because those pesky things interacted with the unpredictable world, and so broke the fiction of determinism.

I should note here that even the most deterministic system has uncertainty and is not, except in principle, totally predictable. There are stray cosmic rays disturbing random bits and creating unexpected consequences; aging and decay, drift. Also, even the most complex of systems can be boringly predictable. It was once said that with the data on cell phones and social media sites you could predict within 2 meters the exact location of a human being 24 hours in advance. This does not make that human a deterministic system, just really boring and habit driven.

Anyway, given the assumption that you can create an artificial human, you go about gathering requirements—hundreds of thousands of discrete requirements. Many have overlapping variables. You have posed yourself a math problem—the simultaneous solution of hundreds of thousand of discrete equations.

However difficult solving those simultaneous equations might be, they are, in principle, solvable; which brings you to design. Other than some kind of bio-memetic metaphor to identify subsystems and architecture, you have little to suggest a way forward. You face a blank sheet of paper and must sketch out a design. Further, it must be a design that you can commit to at the beginning of your effort and by which you will be bound until the culmination of your work.

It takes a lot of hubris to think you can build an artificial human this way—hubris driven by the obvious successes arising from the computing culture. Our world would not be what it is today were it not for the ability of those in the computing culture to build very large, very complicated, deterministic systems.

An alternative approach to the better-human problem is to take the existing human being and intervene. The first step is to understand that system as it is; an understanding that is incomplete, often contradictory, and both uncertain and ambiguous. This sets a tone, like the semi-apocryphal idea that a physician must "first do no harm." You proceed incrementally and with great care.

You use tools, like Christopher Alexander's first notion of patterns (in *Notes on the Synthesis of Form*), or the behavior-driven object identification approach, to isolate subsystems and system elements—where interactions among involved forces are cohesive and lightly coupled with forces outside of that subsystem or element.

Then you make "the smallest possible change that will work" to the system, to paraphrase Ward Cunningham. No change can be irrevocable—if it fails you must be able to reverse course. If it has unexpected and undesirable consequences, you expand your knowledge of the system, and try again. If change renders something you did irrelevant, you simply throw that thing away and replace it. The idea is never to invest too much time and energy into making big alterations to the system, because change is inevitable and you might need to take different directions.

You treat the human as the living system it is, as highly dynamic and complex. Biological systems are not the only living systems. A business enterprise is such a system—business professionals have themselves adopted this view of their world. That is why they use the vocabulary and metaphors

of complex systems, self-organization, agent-based modeling, change (even chaos), learning, culture, and design as they struggle with their concerns and their objectives.

We joke about computer scientists and software engineers being experts at building and modifying "dead" systems. We mean deterministic and mechanical systems. In contrast we talk about our systems as living and use the label of "Living Systems Design" as our umbrella concept.

Come let me show you.

Writer: We went down a floor and a half. It acted as a kind of stage, a place where one could be seen and heard by those everywhere else in the building.

West: Ola my friends; buenas dias. As you know, we have a guest who is very curious about you, what you are doing, and why you are doing it. Please do not stop what you are doing, but forgive us if, as we move among you, we stop and interrupt you for a bit. We have many questions, but you have much knowledge, so please share it with us without reservation.

Writer: As we moved from the dais to the first floor, Dave set the scene and explained a bit more about what we were to see.

West: This is both a school, in a sense, and a development shop. Everyone you see is working, for wages, on work done on behalf of our client companies. When an individual is ready, they leave and immediately start work for one of those companies or others that are eager to hire them. I am pleased to say that an increasing number of those who leave start their own businesses based on ideas begun within these walls.

We have no curriculum, but we have a very broad understanding of what a person should be able to do and how they should be able to communicate and work cooperatively. When it is necessary to learn a bit of knowledge, that bit is provided, taking care to connect it to the work that is being done and to other bits that provide context and lead to deeper understanding.

Writer: Stopping at a group in animated discussion, Dave asked what they were working on. One young man answered: "We are working with the Clinique Français, replacing the information system they brought with them from France. Today we are talking about our approach. Carmen, please explain."

Carmen: Our goal is comprehensive change—changing the existing Clinique into a transformed Clinique—as a whole system. We are guided by the metaphor of biological evolution tempered by the fact that culture is the evolutionary device and mechanism that allows for change at other than the glacially slow pace of biology.

We also ground our work in Naur's idea of theory building and the fact that theory exists only in the minds of those who participate in its creation. Because we are transforming the Clinique as a whole, everyone involved must participate in the construction of the theory.

Our theory must be of the Clinique, as a complex system, and not just the information system, which is but one element of Clinique.

We must identify all the elements making up the system, model them, and understand how and why they interact and what contribution each makes to the system as a whole.

Years ago, Professor Dave taught that the best way to decompose a system was by recognizing the behavior, the responsibilities of the objects (elements) that make up that system. We talk of "contribution" but mean what the professor meant as responsibility.

Only after we understand the system will we be in a position to decide whether any given element or subsystem-as-element might be better constructed as an artificial object, a computer application.

West: Excuse the interruption, but when are you going to start writing code? Are you planning to spend lots of upfront time designing and planning?

Carmen: Not at all! We are doing something that was never talked about in software development as we have read about it. We are engaged in understanding the domain—the Clinique as a system—and making sure our understanding is the same as those directly involved in the Clinique.

Our first step is simply to present to our clients the tools and techniques and ideas that will facilitate our theory development. Theory building is continuous. We will identify possible software bits early and often, implement them, and test how they conform to or modify our theory. It is possible that we will write code this afternoon.

Writer: As we moved across the floor, Dave shared an aside.

West: Back in the late 1990s, a group of frustrated Smalltalk programmers, survivors of the infamous "Project Death Marches" documented by Ed Yourdon, and those perplexed with how UML became a de facto standard, staged a revolt.

Kent Beck's *eXtreme Programmin*g was the first noticeable voice; soon joined by a whole choir. All of them asserting, "We know what we are doing, we know how to do it, we are professionals, just let us alone and we will give you what you need!" The Agile movement was born.

Much of what you see in our studio reflects the early ideas of XP: the open space, the self-organized teams, information radiators, the whole team, and stories—all can be seen everywhere you turn. All with an important difference—we do not compromise those ideas, as happened in the US. The same is true with our use of objects.

What that means, of course, is almost everyone here follows my interpretation of all of these practices. Although I am certain of the correctness and the value of what I have taught here, most of those reading your reportage of our conversations will vehemently disagree.

Writer: We arrived at another group.

West: Here we have one of the few so-called "projects" in our studio. We normally do not do projects—isolated and time boxed development work—nor do we send our people to organizations that do not share our antipathy to "the project."

Miguel, can you describe what you are doing?

Miguel: We are working on an email system under the sponsorship of HaCom, our local phone and ISP provider.

We began our work with a return to metaphor and understanding how a guiding metaphor, what Beck called a "System Metaphor," can influence the design of a product.

Most email systems are grounded in the metaphor of a post office—mail comes into a central office and is then distributed to individual box holders. They, in turn, retrieve the mail, open, read or discard it. If they want to reply they send their own message to a central location for distribution.

Terabytes of data streams around the Internet hourly. Which would be OK except that 70%–80% is junk—unwanted, uninvited, and often malicious.

We are basing our system on the metaphor of a rural mailbox—something remote from your home or business—that you visit only when the "flag is up." This metaphor led us to a design that could have existed in the very early days of the Internet but was not pursued because of the latency required for mail retrieval on modems.

You compose mail and store it on your own server, your desktop, or mobile. You can then send a notice of availability to whomever you wish, but this notice is restricted to a short descriptive subject line and a link to the email.

This drastically reduces the flow of bits across the network. Only mail that someone wants to read actually makes the journey from origin to destination.

Right now we are working on how to make it impossible to spoof your server id, location, and ownership. If we succeed we might have a mechanism for eliminating spam. We are hoping to solve some of the technical problems encountered by the IETF in 2003 in order to address spam and other related issues like the spoofing I mentioned, but we are not there yet. Because so few will respond to a notice—far less than the miniscule number that actually respond to soliciting email—it will no longer be economical. Second, everyone will know where the notices are coming from, and even they can be blocked. If there is fraud or other illegality, the authorities have a paper trail and evidence for court.

Writer: We moved to the second floor, a wide balcony with a central open space atrium, skylights above, a view of the mezzanine stage area and the first floor below. Several large flags or draperies with symbols and vivid colors hung from the second floor ceiling as well as the railing separating floors one and two. Noticing where I was looking, Dave said, "team identification and solidarity—also sound dampening."

Almost half of the second floor seemed to be occupied by a single team. As we approached, a spokesperson stood to greet us.

Spokesman: Hello, welcome to team Leopard. Here we work for several companies around town: a medical firm, a retail store, a cigar manufacturer, and an export company.

Our group consists of those you see in this room and an equal number in our client companies—about evenly divided between those with a development focus and a business focus. The larger group is divided into teams, about five people each team, and we work on overlapping iteration cycles so every day at least one team completes a cycle and takes on new work. This means that no unit of work waits more than a day or two before it moves from proposal to development and no more than three weeks from proposal to deployment.

People are constantly moving from team to team and from studio to client site. Sometimes we see movement among client sites as well. All of this is in support of a common shared theory of the system we address and modify.

Our work is story driven. A story is a narrative, rich with context, characters, plot, and props. We tell stories to articulate and share our understanding of a complex system—to communicate our consensual theory of that system.

Our story telling is loosely based on the model provided by Elizabeth Kostova in her novel, *The Historian*. The story of a young girl in search of her mother develops via the telling of discrete stories, at various levels of nesting and hence detail, each story explaining one character, one action, one motive, etc.

When a story becomes sufficiently focused, usually two to three levels deep in the nesting, and we believe it will take no more than five people working three weeks to implement and deploy, then it becomes a unit of work. The vast majority of stories can be implemented and deployed in a day or two, so each team will spend an iteration implementing multiple stories. Here are some examples:

> The way we handle product returns is all wrong. (1) ***We should have the customer take their items directly to the department where they bought them and have a salesperson take care of the return*** using a cell phone or tablet app. First, the salesperson would (2) ***identify that the item came from our store*** and (3) ***ascertain the reason for the return***. Depending on the reason, different actions can be taken: defective (offer (4) ***replacement***, (5) ***credit***, or (6) ***refund***); don't want (offer credit or refund); wrong size (7) ***make an exchange***), (8) etc. The goal should always be to turn the refund action into a sales opportunity.

Stories can be nested in several levels. Here are the stories within the one above, and stories a level below that.

> 1. policy / procedure story, no software required
> 2. identify item—use your knowledge of merchandise, labels and tags, receipt information, or an (a) ***inventory lookup capability***, to confirm the item was sold by our store or another store in our company.

Writer: He listed many stories. It was clear he was enthusiastic about them. He continued.

Spokesman: The stories in our example constitute a thread—a collection of stories sharing the same context and related to

the same objective. Threading does not determine the order in which stories are addressed or implemented. The thread simply provides context. Each actionable story is assigned a priority—a fraud detection story might have a higher priority than every other story and need to be implemented even before the return process is moved to departments and away from the return desk.

Over here, on these screens, you can observe our workflow. Please notice the absence of anything resembling a project. The center screen collects and displays, in real time, all the stories generated by individuals or teams in our client companies. The only organizing device is some color coding to identify the client company and priority, with more urgent stories at the top of the screen.

The other stories track movement, from selection by a team to deployment. Each team has its own screen.

Note that we use card images—faithful to the physical 3x5 advocated by Kent Beck.

The screens on this wall are replicated at several points in each of our client organizations. The purpose is to keep everyone involved and aware of the evolving theory. It also sparks ideas for new stories. The purpose is to continually build a culture of empowerment, change awareness, and design thinking—by making everything transparent and ubiquitous.

We also design our software to be runtime modifiable—like using indexed collections to store descriptive data instead of object variables with getter and setter methods. If the description changes we can increase, decrease, or alter the contents of the collection without reprogramming the object.

Some things you might notice by their absence are architecture and testing. The system is our architecture. None of our software objects are sufficiently complicated or large enough to warrant much in the way of formal testing. We can implement—write the code that realizes the story— in 100 lines, or less, of pretty simple code. The real trick is design and relying on actionable stories as a unit of work.

Writer: We continued our visits with teams in the studio for most of the day. We joined teams for lunch and I listened to many instances of "Professor West told us…" as people tried to explain what I was seeing and how it worked. Surprisingly, nothing I heard was revolutionary. I had two last questions for Dave when we retired to his apartment at 5:00, when everyone was leaving the building, satisfied with their day's work.

We were enjoying a pre-dinner mojito on his terrace and not the Bodequita, which would have been full circle.

Writer: *"Why does everything I heard today sound so normal? So unsurprising?"*

West: Probably because none of it is new. I am certain you have heard it all before; after all coupling and cohesion have been concepts since the 60s, objects since the 80s, and agile since 2000. All of these ideas, these techniques, have been talked about over and over again.

All I did was take them seriously, try to understand the truth—the mystical true nature behind each concept. Then I sought to link them to other things I knew and other ideas—often far from computer science—that I had encountered. I noticed connections, like the similarity between Alexander's decomposition approach based on cohesion and coupling of resolved forces to yield diagrams (later called "patterns" and the almost indistinguishable form of behavior/responsibility-driven objects.

I read widely. When I moved to Cuba I had over 7,000 books in my library. I had an open mind and refused to adopt any label or any believer's mentality.

In the classroom I was far more interested in having my ideas challenged, especially by those with real experience, than in promulgating doctrine and having it parroted back to me.

I was, and am, an outlier. It did not help that, before Cuba, I had no existence proof of what I believed. Sure, I had successes and was able to get teams to function as Agile, or designs to reflect object advantages—but they were small scale and not widely known. Why should anyone listen to me?

Writer: *"But people in Cuba listened. Why?"*

West: They were a profession starved for ideas, having been deliberately excluded because of the Embargo and, it must be said, the suspicions of a beleaguered Cuban government.

It was probably the only time in my life when being a long-haired, bearded, Harley-riding radical actually stood to my advantage. I was American, yes, but certainly not a representative of the established government. So they listened to me.

When I was here in '94, I was able to plant a few seeds with objects. What I left here was an idea of objects, the responsibility or behavior-driven approach that was already a minority, almost inaudible, voice in the software profession. It worked for them—in part because they had no legacy systems, like relational databases with their antithetical model of reality, to distract them. Nor anyone who followed me to "correct the erroneous impression I had left behind."

When I returned in 2015, there was still a vacuum that I was happy to occupy. It was my good fortune that the University of Havana and the Polytechnic were the most prestigious schools in Cuba, so my ideas spread rapidly. It was at least a year or two before others came to Cuba to contradict me, but by then it was too late.

Writer: *"Will you ever be accepted back home?"*

West: Cuba is my home. But will I be accepted in the US and the wider industrial world? Probably not.

There is a book on the shelf over there on Taoist painting. That style of painting was the result of meditation and deep understanding of the mystery of life. It was based on empirical study and contemplative insights.

Guo Ruoxu (ca. 1041–1098) was the most influential art critic in eleventh century China. Committed to a formal "science" of painting, he dismissed Daoist practitioners:

> *At the beginning of the [present] dynasty there was a Daoist adept, Lu Xizhen, whose every painting of a flower on a wall would attract bees. Aren't such peo-*

ple [who paint a flower on a wall to attract swarming bees] the same who gain merit by bewildering people, and find fame by confusing the practice of art? As for rustics climbing a wall and beautiful women dropping from a parapet, the dazzle of the five colors in the midst of water, and the ascent of twin dragons beyond the mists—all are the products of sorcery, fantastic lies without any relation to the laws of painting. None such, in consequence, is recorded here.

I fear I am but a Taoist software developer without standing. There are many who will dismiss whatever I have said—whatever influence I have had, any successes of the Cuban software movement—as "products of sorcery, fantastic lies, without any relationship to the laws of development." I doubt that I will cross the threshold and be "recorded here."

Writer: With those sad words I took my leave. I returned to the Bodequita, alone this time, drank the obligatory mojito, and before I left, I wrote my name and year on the wall, down near the baseboard where a miniscule space remained.

I will write my article and it will be published. I can envision what the response will be.

Dave once said that the only way his ideas would prevail is if the Cuban software industry were so successful that it escaped the limited realm of Central and South America and became adopted across the world. And even that might not be enough, because then the world would co-opt, dilute, and re-purpose all that he had said. The issue is really one of cultural change, and history suggests that few cultures change willingly but often by conquest. So, if Cuba were so successful that the IT industry in the rest of the world utterly failed to compete, then the prevailing culture would change.

Dave's story—the way he tells it—flies in the face of everything we've known to be true about programming since the early 1970s: the progress of computer science as a science and software engineering as engineering will eventually solve most or all our software-related problems, just give it time.

The way he tells his story reveals biases—the golden age of programming for him was the 1960s, and Cuba sat arrested in time from those 1960s until he arrived in 2015, when his hippie fantasy could resume. Like Christopher Alexander, his motto could be "the grass was always greener yesterday."

His own words reveal his megalomania and conviction that only he is right.

Cuba was a fluke, in the sense that it was in stasis until his arrival and there was a void that he was able to fill. He did not have to change anyone's mind or overcome decades of propaganda with its resulting embedded cultural mind-set. Those circumstances cannot and will not be repeatable elsewhere. From what I have read the early days of both objects and Agile were characterized by amazing success stories but over time the impact of both was minimal.

I won't need to slant my article; it'll be clear to everyone—right or wrong, Dave is simply crazy.

5. Afterword

An interview offers little opportunity to collect detailed footnotes but readers immediately asked for references for the things Dave was asserting, so I am doing my best to provide some additional information.

Early on, Dave mentioned some ideas of Hans George Gadamer. These came from *Truth and Method*, published by Crossroads in New York, 1982.

The quote of John von Neumann's remarks to the 1947 ACM conference is cited in "Archaeology of computers: Reminiscences, 1945–1947," *Communications of the ACM*, volume 15, issue 7, July 1972, special issue: Twenty-fifth anniversary of the Association for Computing Machinery, p. 694.

Clifford Geertz discusses his notion of "thick description"—based on the philosophy of Gilbert Ryle—in his book *The Interpretation of Culture: Selected Essays*. Basic Books, 1973.

The quote about scientific management view of workers as "lazy…" is usually attributed to Frederick Taylor, one of the primary founders of scientific management. I could not find a specific instance of exactly that quote, but it would be easy to infer from his comments about "soldiering"—the notion that workers deliberately mislead management by obfuscating their work and taking more time than necessary, his assertion that only management had the ability to plan and direct work, and that workers need to be microcontrolled with no room for individual initiative if tasks were ever to be accomplished efficiently and profitably.

Three contrarian viewpoints were cited in the text: D. C. Parnas and P. C. Clemens, "Rationale Design Process: how and why to fake it," *IEEE Transactions on Software Engineering*, 12:2. Feb 1986, pp. 251–257; Peter Naur, "Programming as Theory Building," chapter 1.4 in *Computing: A Human Activity*, New York: ACM Press, 1992; and Christiane Floyd, Reinhard Budde, and Heinz Zullighoven, *Software Development as Reality Construction*, Springer-Verlag, 1992.

West claims he wrote the book chapter "User Stories"—in Ian Alexander and Neil Maiden, *User Stories in Agile Development*, London: John Wiley and Sons, 2004. Kent Beck is listed as co-author, but according to Dave, he made one suggestion for the text, adding a paragraph that emphasized the intent to redefine the relationship between user and developer to one of cooperation instead of adversarial. Beck did not even make edits to the text that Dave wrote.

The quote of Guo Ruoxu came from Shihshan Susan Huang, *Picturing the True Form: Daoist Visual Culture in Ancient China*. Cambridge, MA: Harvard University Asia Center (Harvard East Asian Monographs), 2015.