WINDOWS SEAT

In Search of Organic Robustness

Gen Kiyooka

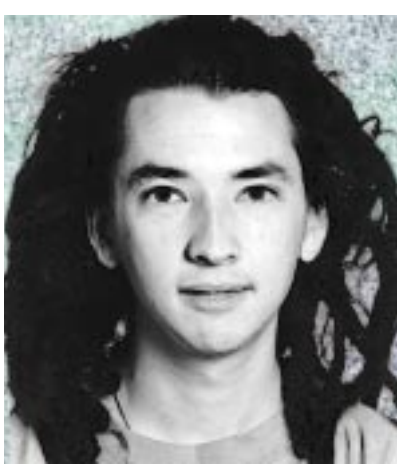
There may be great benefits to be gained by the notion of "growing code"; that is, creating a robust and stable software system by planting appropriate seeds and letting the system grow as an accretion of methods around those seeds. An analogy might be if you dropped some seeds into some big, red flower planters and strung a fishing net alongside the flowerpots, so that the plants would have an appropriate structure to cling to as they grew skyward.

Now, a great many people would suggest that digital computer programs, under a wide variety of operating conditions. Organic systems are created from a tiny, magic strand of DNA, considerably simpler (even in its complexity) than the end product. Thus, we might imagine that organic systems are the products of simple first principles. Likewise, in our software development efforts, we would care to have elegant first principles from which highly beautiful software simply unfolds, as the petals of a flower in response to the sun.

Think of the aggregations of "abstract particles" that compose our universe. We imagine these to exhibit the robustness and elegance of organic systems.

Organic Design

There are many reasons why we as developers might choose the way of nature as the way of our creation. First, we like to imagine natural systems as being more robust than our software counterparts. And robustness is a characteristic we desire from our software—we want software systems that perform admirably being of the discrete system variety, are necessarily excluded from an organic reality of continuous, adaptable, and robust systems. Perhaps the answer lies in the bazillion onion-skin layers of particle physics. At the atomic level, we have envisioned some particles from which we imagine our universe to be composed. And these aggregations of "abstract particles" (a description easily contested, perhaps) we imagine to exhibit the robustness and elegance of organic systems.



It would seem to me that the principal chemistry of organic systems is based largely on a finite set of imagined discrete units. The combinatoric explosion as these discrete units are aggregated is the basis of a robust organic system. Or perhaps it's the myriad possibilities in aggregating the aggregations. So, if you consider our current computer systems to be principally subatomic, perhaps the "chemistry" of digi-organic systems will follow in the next 50 years, and the truly delight-

ful systems themselves shortly thereafter.

But what if organic systems aren't really continuous and robust after all? Continuity may be just an abstract mathematical dogma to be "poofed" in the end by the smear of probabilities that deterministic science has come face to face with in quantum electrodynamics. Perhaps continuous functions exist only in the space of the mind, emulations of which appear in discrete systems comprising many particles exhibiting reasonably continuous behavior. Is a human being (that is, an organic system) robust?

Drop one off halfway between here and Luna and see how robustly it behaves. A gruesome thought, but certainly similar to the programmer who believes his or her software is robust simply by excluding (failing to test or subject the program to) a realm of circumstances—like a certain class of Windows application trying to run under low memory conditions. Can you say "Loss of Data," boys and girls? Likewise, a small amount of arsenic is apparently fatal to our own systems, arsenic being a distant cousin to the uninitialized C pointer stored in an auto-variable on the stack. In fact, why not write a preprocessor macro as a mnemonic device?

```
#define ARSENIC auto
```

Appropriate usage would be:

```
BOOL  OBJECT_SomeMethodUponWhichSoftware\
Depends( POBJECT pSelf )
{
ARSENIC char
*DontMessWithMeBabyICanBringYourDis-
creteSystemToItsKnees;
```

In 99 out of 100 execution paths, the uninitialized pointer happens to receive a convenient value that hides its life-threatening properties. It runs fine until you're in a roomful of corporate executives who are sponsoring (read: funding) your work. Hah. Don't blame anything on Microsoft: Windows at rest tends to remain at rest.

Perhaps the notion of robust organic systems is merely an expression of the human ego's belief in self-immortality—a belief that might lead to the blindness that transforms the ecology in which we may be robust into one of Unrecoverable Application Errors. So, if you'll let me dismiss the notion that continuous, robust, organic systems are an artifact of anthropocentric thinking, we can get on with trying to emulate them in our own work.

Toward Organic Robustness

Can we arrive at a consensus of software robustness? I guess we're not in an interactive medium here, so I assume I'll have to do this by myself. Robustness, it would appear, is a different kind of halting problem. Instead of trying to get the computer to halt, the goal is to prevent it from halting at the wrong time. Halting at the wrong time can mean loss of our precious data, data that is the product of work and effort. Halting at the wrong time might also suggest an airplane full of people crashing into the mid-Atlantic, but more about that later.

Another view of robustness is the integrity of data. Data in this case is all the bits and bytes that represent the "state" of execution, digital computer software being nothing more than an aggregation of tiny, finite-state automata. Consider a 16-bit integer whose job it is to represent a kind of fruit from apple to mango with a banana between. We can represent this set of fruit with a number from 0 to 2. That leaves 3 to 65,535 unused. Perchance, at execution time, this 16-bit representation of fruit inadvertently takes on the value of 43. What behavior can we expect of the software system when such a simple assumption does not hold? Unruly at best.

To me, the goal of robustness is twofold. The first is to prevent the program from halting unnecessarily, preventing data loss. The second is to prevent the bits and bytes that make up the execution state from taking on "incorrect" values, those values for which the behavior of the software is not defined. Now, consider a software system consisting solely of a group of objects properly encapsulated by methods solely responsible for changing the states of the objects themselves. If the methods on the object are correct, then we can expect proper behavior. But methods take arguments and arguments can be improperly submitted. Improper arguments may result in undefined behavior by a method. Such behavior may include changing the private state of an object into an undefined state. Objects may also be coerced into an undefined state by a wild pointer write (in languages that allow such barbaric activity).

Now consider this group of objects as nothing more than a Windows application awaiting input from the person sitting at the keyboard. Until the person submits some keystroke, mouse movement, or click to the application, nothing happens. When such an event occurs, imagine what follows. The event enters the system as a message and is then translated into a cascading series of operations on the objects that make up the program. Should the mouse-click result in an "out of range" point argument being submitted to a method, we're into a robustness scenario. How does the software respond?

By my humble estimation, if the software is truly a collection of objects encapsulated by methods, and the methods of the objects "precondition" each and every parameter passed into the method, the software will act robustly. In the event of such a mouse-click scenario, the precondition to the click method will detect the "out-of-range" point value and simply fail the method, which alters the execution path either by throwing an exception or by propagating return values or error codes. The net result is that the event submitted to the software, a mouse-click, does not have any effect whatsoever on the state of the program. The software is robust.

An alternate scenario involves the scarcity of some necessary resource required to translate an external event (that is, a mouse-click) into desired program behavior. In the case of a Windows program, it might be that the mouse-click has resulted in drawing to the screen and, in the course of preparing to draw to the screen, the system reports that no more GDI objects may be created. Again, a

program that behaves robustly simply fails to complete the operation, freeing any other resources that may have been allocated in preparation for carrying out the operation. A message to the person at the keyboard might be appropriate if there is someone sitting at the keyboard. If no one is there, perhaps an entry in a log file would be right. Proper reporting increases the chance that any required changes in environment, necessary for completely successful future operations, will be undertaken by the appropriate persons.

The Naive Parenthood Syndrome

From this Canadian's perspective, the reason that a large portion of the software in the world is completely lacking in any sort of robustness is due to a phenomenon I call the "Naive Parenthood Syndrome." Bear with me here.

Software creations, like paintings, buildings, and songs, can be likened to the child of the creator. We delight in our creations and wish the best for them, just as parents wish the best for their children. The naive software developer is a naive parent. Rather than confront a world of complexity, the parent creates an insular world for the child where the child can be happy, shielded from any harsh circumstances. Sooner or later, the child becomes an adult, destined to be stranded in downtown Los Angeles with a flat tire, lost wallet, and some close relatives in Toronto.

The software developer, wanting only the best for the "app-ling," gives it a comfortable room with 32MB of RAM, 4GB of hard-disk storage, and the Ultra-MaximumSuperDuper video card with blazing video. Not wanting the poor babe to be subject to less-than-optimal conditions, the naive developer does not test on lesser (or a wide variety of) machines, or run the elephant stress program in the software development kit, or load a plethora of applications to test for interoperability. Only the best for Junior.

Knowing that the child is safe from harm, there's no need to validate all parameters passed to functions from abroad. Knowing that there will always be plenty of memory and resources available, there's no need to teach the kid how to survive on a diet of stale Twinkies. Knowing that a file will never be in use by another application, we can open it with the full knowledge that the file handle will come back ready to do our bidding. The child grows up ill-prepared to deal with life outside the suburbs. To a hobbyist programmer, life is the suburbs, and children software need never leave. To a corporate developer or commercial software developer, destiny dictates otherwise.

The state of affairs in software today is so dismal that we weakly demand our robustness be built into the operating system itself. Ill-behaved applications can be terminated without affecting the rest of the system. Here, take Mommy and Daddy's credit card in case of an emergency. Unfortunately, the analogy breaks down. Human beings are adaptive. Children from the homes of naive parents can use innate creative problem-solving abilities to adapt to new circumstances. Software kids, on the other hand, are doomed to the limited deterministic behavior granted them by their developers.

The principles underlying resilient, robust, organic-like software systems are remarkably simple. And perhaps, if we could build our current systems in a way that would keep them running soundly, we might have a chance at aggregating these systems of discrete objects together into more gregarious organic systems of greater complexity.

Systems We Build

I hope this organic approach doesn't sound too serious. These systems we're talking about need to be put into perspective. We are not gods in this world. This is software, for goodness sake. To ask why air-traffic-control, military, or medical software is taking people's lives by being inadequate is to ask the wrong question. Marshall McLuhan documents the phenomenon in *Understanding Media* (Penguin Books, 1964):

"Those who are concerned with the program 'content' of media and not with the medium proper appear to be in the position of physicians who ignore the 'syndrome of just being sick.' Hans Selye, in tacking a total, inclusive approach to the field of sickness, began what Adolphe Jonas has continued in *Irritation and Counter-Irritation*; namely, a quest for the response to injury as such, or to novel impact of any kind. Today we have anesthetics that enable us to perform the most frightful operations on one another.

"The new media and technologies by which we amplify and extend ourselves constitute huge collective surgery carried out on the social body with complete disregard for antiseptics."

So, if you're wondering how you're going to make the computer-controlled braking system safe for drivers of the car that you're engineering, you might ask whether it is right and proper to encourage a human being in a cloak of sheet metal to go flying down a roadway at high speeds. Humans are apt to die under these conditions, irrespective of the embedded-systems software managing their brakes. Well-maintained analog brakes fail, as does human judgment in times of urgency. I prefer an armchair in the sun on the porch of a house at the nape of the Rocky Mountains. If I have to get in that infernal machine to drive to the grocery store, I can hardly blame an engineer at Ford Motor Company if the vehicle fails in some mysterious way en route; software or no software. ■

Gen prefers an analog Fender Stratocaster plugged into a Mesa Boogie tube amplifier to even a sea of Silicon Graphics Reality Engines rendering missile simulations for the military. You can contact him via e-mail at 7637.43@compuserve.com or through Software Development.

Readings for Robustness

Interview with Alan Kay and Danny Hillis, *Wired*. Vol. 2, no.1, p. 104.

Genius: The Life and Science of Richard Feynman, by James Gleick; Vintage Books, 1992.

Object-Oriented Software Construction, by Bertrand Meyer; Prentice Hall, 1988.

Understanding Media, by Marshall McLuhan; Penguin Books, 1964.