WINDOWS SEAT

Reusability is a Habit

Gen Kiyooka

Welcome. A monthly column about Windows and OLE, now, what would that be? The programming magazines on newsstands today are packed with articles explaining the coding how-tos of various aspects of the ever-expanding Windows universe. So perhaps then, a different approach is in order. Having had the good fortune of a liberal arts education—an education that invariably prepared me well for what it seems I love, striving to design tools and experiences in which the usability and elegance run deep—I recall one particularly engaging sociology professor who challenged his students to "take their experience and make something of it."

And yes, I imagine, that is what I am doing. So, rather than attempt to simply disseminate morsels of WIN API knowledge, I believe my goal here is to bring you as a visitor into my headspace: one, that as you may see, is largely concealed by a wig of natty dreadlocks. And dreadlocks, you see, are a hairstyle of habit.

Reusability, I have ascertained, is nothing more than a habit. You see, it happened to me. It began with an idea—a notion of eliminating redundant effort. Just over two years ago, I pledged to myself that reuse would be my highest software development ideal (at least from the perspective of "internal elegance"). Usability and delight would take a similar position as primary concerns in the software's outward expression. I was starting from scratch, of course, writing on a clean slate while headed out into the intensely competitive realm of commercial application software.

My goal is to bring you as a visitor into my headspace: one largely concealed by a wig of natty dreadlocks. And dreadlocks, you see, are a hairstyle of habit.

And what a choice that turned out to be! I recently had an opportunity to take a break from a hectic schedule and reflect upon a 20-month stretch of intensive development. I read some philosophical tomes, including Napoleon Hill's *Think and Grow Rich* and Robert M. Pirsig's *Zen and the Art of Motorcycle Maintenance*. What struck me as I reflected, read, and sipped espresso, was that the truths of successful software development were quite simple. We are what we habitu-

ally do. The habit of creating reusable code, the habit of object-oriented design, the habit of encapsulation, the habit of robust code, the habit of checking every return value, and so it goes.

The gulf that stands between theory and successful practice is not whether I understand, not whether I am able, not in which language I use, not in whose methodology I worship, not in the speed of the box I own, not in the FLOPMARKS of my loops, but rather whether I endeavor to habitually perform and practice those techniques that can be readily identified and associated with the art of successful software and of human enterprises in general.

For me, reuse will remain my highest concern until I no longer consider my own practices inefficient and primitive. My soul chafes at the thought of spending hours, days, or weeks implementing a particular solution whose design, structure, and form I can establish in my imagination within a few minutes. It was this fundamental dissatisfaction that drove me to form and discover new reusable structures to reduce the effort of my Windows programming by an order of magnitude. Where once hundreds of lines of code were needed, a single line could do the job. Sounds a little preposterous, doesn't it? It isn't. From my perspective, I've still got a long way to go. If you set forth a goal, and refuse to stop until you achieve that goal, it will come to be. This applies even to the goal of eliminating 95% of the coding normally associated with Windows dialog boxes. It's not a silver bullet, it's a habit.

How exactly does one form such a habit of reusability? To tell the truth, I haven't got any absolutes for you here, but I do have my own experience. And this column is about sharing that experience with you.

The Zen of Reuse

Start with a problem that challenges your ability as a developer. Start with a large, complex problem with hundreds of thousands of lines of legacy code to be maintained. Experience firsthand how bad it can be. Experience complexity. Develop a burning need to master complexity. Once you've established this burning need, take on another project in addition to the maintenance of legacy code. Choose Microsoft Windows for your new project. Choose a pure object-oriented programming language as the tool for your folly. Choose C as a tool for performance-critical sections of your project. You follow so far? It could happen to you; it happened to me.

Now, let's say you expend a great deal of effort trying to duplicate the pure object-oriented programming experience in the parts of your project that are to be written in C. The first thing you notice about the pure object-oriented programming environment is that your productivity is enhanced because you are not forever trying to decide where to put new functionality. The richness of the class framework suggests simply by the rigor of its organization where new classes are to go. Let's just say that you decided to emulate this affordance (Donald Norman's terminology) in your standard C environment.

You begin by establishing some libraries of abstract data types, even though there is nothing yet to place in them. Whenever new code is designed and written, if some functionality could be identified as a method of an object, a class would be added to the library, even if the single method is all that is initially associated with the class. Over time, second, third, fourth, tenth, and more methods would begin to accrete to the seed classes, and, as if by magic, a framework would grow. Because these reusable classes are part of a library, they could be easily reused in any number of projects, and, again, when included in these other projects, more methods would accrete to the framework. And from this, a fundamental pattern or habit of reusability would emerge.

To reuse is to organize. Without a place to put methods and classes that are candidates for reuse, any such candidates would remain scattered among various projects and source files. To reuse is not to sit down, design the classes for your next three years of work, and implement those classes. To reuse is to plant the seeds of reuse in the work of today and nurture and grow those seeds into forests of trees over time. To reuse is to form a habit.

Spend several years of your life working hard, almost burning out. Make good on the project, reach some important milestones, deploy some of the systems into a corporate environment. Spend some time pondering your work effort. Recall a certain magic surrounding the time you spent working in the pure object-oriented language. Ponder the origin of the magic. Compare it with the results of your experiments in C. Ponder.

The corporate world passes away as you enter the new world of commercial software. A world where you make your own decisions, based on the needs of the marketplace. A world where the software you develop is your only asset and your only liability. You spend about a year developing some products that show commercial potential. As time passes and you sense the cost associated with the ticking of the clock, you begin to ache and long for tools that would shorten your development time.

In a month of breathing space, you revisit the whole pure object-oriented phenomenon in your mind. You build a mental model of software development. In the model, the infinite redux of every programming problem can be stated thus: there are things, and there are collections of things. It becomes obvious why LISP has become a language of choice for AI. In LISP, there are things (atoms), and there are collections of things (lists). Similarly, it becomes apparent why Smalltalk has its share of fanatics: in Smalltalk, everything is an object, in a most uniform sense. In

Spend several years of your life working hard, almost burning out. Recall a certain magic surrounding the time you spent on the project. Ponder the origin of the magic.

Smalltalk's class library, all well-understood entities (classes) have been distilled, organized, and arranged into a hierarchy. Everything that has been understood is already complete. All that remains is to solve new problems. In your model, there are things, and there are things that have been recognized, distilled, and organized into reusable classes. The older a notion is, the more likely there is to be a class that represents it.

Through every era of advancement of software development, a pattern is recognized. We struggle through a mire of details and technical problems to emerge with a model that creates a uniform foundation upon which to stand. An operating system creates a virtual address space with several gigabytes of virtual memory. A class library provides a model for organizing items into kinds of collections: sets, bags, ordered lists, hashed tables and more. These models give us the power to strip away and forget that which would entrench the solution of more ambitious problems in myriad detail.

So, you turn again to face the challenges in your particular development situation. Despite your awareness of appropriate structures and despite the availability of virtual RAM and visual tools, you still find yourself coding, over and over again, repetitive structures that are particular to your problem domain. And let's just say that problem domain is the development of quality commercial Windows applications, and your development involves much work with refined user interface.

To you, the class frameworks offered by library vendors are mere shadows of the elegance of the Smalltalk classes, suggesting that the reusability you desire lies in the distillation of repetitive patterns that fewer people are aware of. A survey of the GUI software community reveals few people focused on reusability, few focused on outstanding user interface and even fewer focused on both. You want it all, and to get it, you realize you're going to have to find the patterns, understand them, and produce reusable classes to conquer them.

The Final Moment

You spend about three months falsifying reports of progress on the next project while you step back, examine where you are, where you've been, and where you want to go, and set forth to design new classes to get there. You're about to break the model-view-controller stranglehold on GUI class abstractions, and there's no precedent for how it's to be done. There's a great deal of thinking involved and not so much coding. There's a lot of slogging at first, and the warm fuzzy feeling of creative feedback seems distant, but you persevere. In the end, you come up with an unusual, almost heretical system of developing Windows applications. It's a cowboy solution that would be frowned upon by any codified system of corporate standards. And it's hard to understand because so few are yet concerned with thinking about the patterns the new classes represent.

With the new model and framework seeded in your environment, you set forth to complete your next project. There's a joy in seeing tasks that once consumed hundreds of lines of code reduced to simple packets of two or three lines. Over the 11 months that follow, your new framework accretes new abstractions, and your productivity continues to improve. Just over a year later, you set down and test yourself. The challenge is to write a code-generating wizard, consisting of a Windows application with five or six dialog boxes that allows you to manufacture DLL modules for a groovy new lifestyle accessory. The project takes a single day, and you've got the results to prove it.

One day you wake up feeling like your productivity really sucks. Sure, what used to take a month can now be done in two days, but it's hardly enough. Standing at this highly productive perspective has given you the ability to see how inefficient your practices really are, and so you begin to ponder. Ponder.

Some habits are hard to break. ■

Gen Kiyooka recently finished developing RoboHelp 2.0 and is currently pondering reuse in San Diego, Calif.

Further Reading

Selected readings about habits and growing things, chosen by the author.

Richard Gabriel, "Critic at Large," in the *Journal of Object-Oriented Programming*.

Napoleon Hill, *Think and Grow Rich*. (Fawcett-Crest, 1960)

Robert M. Pirsig, *Zen and the Art of Motorcycle Maintenance* . (Morrow, 1974)