Workshop on Designing the Software Systems of the Future,
Software Engineering Institute, Carnegie Mellon University,
Pittsburgh, Pennsylvania, USA, January 12-14, 2018

Future Software Design

Richard P. Gabriel
Software Architecture Group
Hasso Plattner Institute
rpg@dreamsongs.com

Let's start with how to structure nonfiction.

In December 2017 I was reading a new book by John McPhee—"Draft No. 4: On the Writing Process"—in it he describes how he structures his nonfiction essays and books; and I unexpectedly started to apply his ideas to how people work with software. I was thinking about programming and not design methodology and standard engineering—or perhaps the experience of programming. (I find it hard to word this description because it is too easy to fall into software engineering jargon, which might lead us into a mental rut.)

The two ways are these: thematically and chronologically. Most nonfiction succumbs to chronological treatment: this happened, then this, then that…until the end. Sometimes a chronological piece has flashbacks and flash-forwards in which segments of chronological time are pieced together out of order. For example: me right now telling you that I first encountered McPhee's writing the early 1970s just after I moved to California from Illinois, and from there I started to take my own nonfiction writing more seriously.

In thematic structure, things that go together are grouped together. I could for example put together a section in this little essay about all the things I did throughout my life to become a better writer, and within that section I might group similar approaches and studies without much attention to when they took place. Many technical papers are thematically structured.

Going only this far into what McPhee was explaining regarding writing, I leapt at the idea that in programming, thematic structure is modules. At least from far, far away it can seem like this. One can think of a module as a grouping to which a single team is assigned, and the information that team handles can be divided into the information that is private to their concerns and the information that is exchanged with other modules. Information that can be grouped together so as to be kept from others is information that goes together. A theme.

What then is chronology in programming? It's one thing after another—it's execution; stuff that happens; running status; **A** then **B** then **C**. This is the inexorability of the actual; it's the real thing not a plan. Chronology is fundamentally about the runtime.

Let's not look too hard at these metaphors but instead examine them vigorously in the shallows. Themes seem abstract as do modules. Certainly a software module contains concrete code, and a module as text contains source code, but aside from what could be described as degenerate cases, that code is abstract. At least abstract in the sense that the code will behave in some manner once some of its originally unknown parts are provided—such as the value of the (abstract) fibonacci function is computed once its source code is compiled and a particular concrete argument is supplied. Until then, the source code is simply there to examine and reason about. When it's a plain and simple function we can usually recognize what it is. But even the code for fibonacci can be mysterious. For example, is there a limit on the magnitude of its argument? Looking at the source code, we might not know. Looking at the source code alone, we might not even know what language it's written in. When we decorate the definition with types, we might know a little more about it, but we still might not know argument constraints. And perhaps all a compiler can guarantee is that it is roughly in the form of source code that computes a number, but it cannot tell us that it computes fibonacci.

Consider the code at the top of the next page. You can be forgiven if you can't read Lisp code, but I would guess that even superb Lisp coders can't figure it out at all.

Well, of course it's fibonacci, but it can compute for large arguments—it takes a lot longer to print the exact answer than to compute it.

(I showed you this code to briefly defamiliarize you, to make you feel a little of the stupidity one can feel when encountering unfamiliar source code.)

A compiler can fuss & fume and finally tell us its types are ok (if we label them ok). And perhaps someone writing out derived equations could figure it out after a bit. But otherwise it is abstract. A programmer writing thematic material is rely-

```
(defun f (n)
  (labels ((t7 (t1 t5 t6)
             (let ((t2 (+ (* t1 t1) (* t5 t5)))
                   (t3 (* t5 (+ t1 t6))))
               (values t2 t3 (- t2 t3))))
           (t8 (t5 t6 t4) (declare (ignore t4))
             (values (+ t5 t6) t5 t6))
           (t9 (i)
             (cond ((= i 1) (values 1 0 1))
                   ((oddp i) (multiple-value-call #'t7 (t9 (ceiling i 2))))
                   (t (multiple-value-call #'t8 (t9 (- i 1)))))))
    (cond ((zerop n) 0)
          (t (values (t9 n))))))
```

ing on abstract thinking and tools that suffer to some degree the ignorance abstraction requires. Thematic programming is like writing code on a piece of paper.

Contrast that with the chronological—the execution itself or the runtime. We can quickly see that for arguments up to about 10 the pattern for *f* looks like fibonacci: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55. We get to do science on the chronological code, and only mathematics on the thematic.

Let's look at the experience of a programmer using a *live* programming system. A live programming system is one where you sit in front of the running code and changes you make to the "source" are immediately reflected in the running system. You can pause parts of the system, inspect data, insert breakpoints, and otherwise manipulate the running system. Systems that approximate this include Lisp systems, Smalktalk, Self, and many others.

A live system is chronological because instead of reasoning about what the program might do, the programmer can look at what it has done and what it is doing right now. The experience is concrete.

Let's call these two types of systems *thematic* and *chronological*. They are good names because they have no baggage in the software engineering and computer science worlds.

In a thematic system, bugs that are found and analyses that are produced are thematic and hence abstract. Thematic systems are generally static, like ink on paper. Finding such bugs and providing such analyses are extremely important, and a guarantee that is made thematically is one that can be counted on. Computer science and software engineering rightly strive to find such good thematic tools.

On the other hand, because there are many things that can be wrong with a executing system, the ultimate test is the running code. Support software can fail; hardware can fail by faulty design or manufacture, or by adverse environmental conditions; correct software can be used incorrectly; it can be used in adverse circumstances (too little memory). This is why there is testing and things like testing.

There is a whole other dimension to chronological systems, and it's the one I think is related in a surprising way to the future of software design. Chronological systems have a human in the loop. A live system with its programmer on vacation is a static system that happens to be running. A chronological system with its programmer asleep at home is a thematic system that has been compiled, linked, and is executing.

A person sitting in front of a chronological system can improvise—try things out, experiment, make mistakes, and recover from mistakes. There are many tools that help programmers who improvise badly from killing the system. But an important factor is that a chronological system can be adapted on the fly by the programmer sitting in front of it. And such a programmer, if very alert, can be aware of what the system is doing. (And by "programmer," I include several people—a team.)

In some live languages, it is not only possible to examine live structures and data, but to also alter behavior by inserting code, altering code, and manipulating the underlying execution structures. This is called "reflection." Some languages have mechanisms that enable programmers to operate at a meta level on meta-objects that implement the semantics of the language. Hence reflection can be broken into introspection and intervention or intercession. Some refer to the ability to effect changes in the fundamental executing semantics of a system "behavioral reflection." But in all this—so far—this is all under the control of a programmer sitting in front of a live system.

Comparing the experience of acting on a textual representation of a program with acting on the program actually running, we observe that a sort of distance is shortened. When we look at a program like *f* above, we can observe its structure, see how its parts relate, and can imagine it running; but we don't get to see without mental effort what happens when some example values are provided for the various temporaries. We can deduce what the concrete might be or will be, but that is not as immediate, not as close up as seeing actual values. Some programming environments can provide stunning concrete visualizations of the code executing. The distance between the language describing or representing the program and its usage is reduced. If we take this to an extreme, we might say that the language, the program, and the programmer are part of a single unit. Or perhaps a single organism.

When viewed this way, we can take one more step: a system which is able to introspect can be described as self-aware, and a system able to modify itself can be described as conscious. In a live system, that consciousness is supplied by the human sitting there, observing, thinking, and acting. In typical software engineering scenarios, this is part of an exploratory

approach toward understanding the problem and solution domains in order to construct a program that will be used in the traditional way: a static program that happens to be running.

Having made a leap from thematic systems to chronological systems, is there yet another leap that can be made from chronological systems? I think there can be. It is this:

> *We should be figuring out how to create an **artificial consciousness** embedded in our otherwise ordinary systems.*

Not to do or provide artificial intelligence, but to provide the ability to introspect, think, and intervene for the purpose of ensuring the system is doing what we intend it to be doing. Currently we reify such activities in exception handlers, consistency checks, checkpointing, fault tolerance patterns implemented, etc. What this reification does is break down some places where consciousness would help and patch safeguards into the system.

What I am talking about is a generalized consciousness—it can be thought of as a low-resolution model of the running system and the world with which and in which it is (inter) acting. (This formulation is after the work of Thomas Metzinger, a German theoretical philosopher.)

This leap turns into code the human that provides consciousness in chronological systems—the human that creates the advantage chronological systems have over thematic ones. What are some things we could (possibly) do with an artificially conscious system?

The system can ask itself the following questions and take appropriate actions: What am I doing now and why am I doing it? Am I doing things the right way? Does the mistake I just made matter? Do I need to ask for help? Maybe I should plan this next part out by pretending to do it and envisioning what happens. This is taking too long—why? This looks bad, I should stop. I need to do this operation faster; I'll ask my developer to take a look at it. Oh dear, that seems like it could be an attack, I'd better hole up and act cautiously.

What does it take to make a system artificially conscious? Not sure I know, but perhaps working on some of these can help:

Sensors: How can we make sensors for internal stuff like processes, data structures, parallel operations, and the immediate world around the system?

Describable structures: How can we make data structures self-describing or describable as well as understandable and explainable by consciousness code at runtime? Chronological systems already make manifest the types of its data, but this does not go far enough toward capturing intent and purpose along with the particular meaning of the current instances.

Low-res model: What does a low-res model of the system and its immediate surroundings look like? How is it implemented? Does it include a simulation to compare actual outcomes to predicted ones? Can some of these simulations be machine learned? Is the model "inside" the system or "outside"?

Intercession: What kinds of intercessions are possible? How are they accomplished? Does this entail an expanded notion of modularity and information hiding?

Artificial intelligence: To what extent does some degree of AI need to be part of the artificial consciousness of a system?

Execution environment: Does artificial consciousness require a different kind of execution environment—perhaps one not optimized for running (parallel) Fortran?

People: How does an artificial consciousness relate to people—both those inside and those outside the system? Think of that friendly developer who was asked to revise part of the system.

Innovations for programming: Programming for artificial consciousness will not be much like regular programming—I would guess; mechanisms will need to be invented. Is it possible these will spawn new control and data structures for the "regular" part of the system? A long while back a project I was involved with came up with a sort of method invocation that would begin when execution "entered" a specified region of code. This without flags. It seemed to be fun to program using it, and some programs using it were much simpler, shorter, and easier to understand.

The world of software ahead will be full of mysteries and unknowns—and I don't mean simply that we can't predict the future, but that every system will be designed, implemented, executed, and maintained in an environment that is hostile and chaotic. Perhaps at the hands of our own (unavoidable) ineptness. The design and execution of those systems need to take care of themselves. A machine designed to operate adaptively through the sheer cleverness of its making is, I suspect, going to be in for a difficult time of it. Many of the living things around us have brains of one sort or another, or are otherwise autopoietic one way or another. We need to do that sort of thing for software.