Phenotropic Programming?

Clayton Lewis
University of Colorado Boulder
430 UCB, Boulder CO 80309 USA
clayton.lewis@colorado.edu

Abstract. In about 2002 Jaron Lanier began writing and speaking about phenotropic programming, in which programs aren't "simulations of vast tangles of telegraph wires" but instead rest on"the interaction of surfaces", using "pattern classification as the most fundamental binding principle, where the different modules of the computer are essentially looking at each other and recognizing states in each other, rather than adhering to codes in order to perfectly match up with each other." Lanier thereby hopes to combat " brittleness" of today's software, that makes software engineering "Sisyphean". These ambitions are consonant with ideas recently conspicuous at PPIG, as in Basman's (2016) paper, Building Software is Not a Craft. What has become of Lanier's vision in the last 15 years? This paper will review work that has been inspired by Lanier's ideas, and consider how further work may contribute to Lanier's goal of a "convenient [and] cognitively appropriate starting point for human beings who wish ... to go on to build things".

1. Introduction

In two related essays, “The complexity ceiling” (Lanier, 2002) and”Why gordian software has convinced me to believe in the reality of cats and apples” (Lanier, 2003), virtual reality pioneer Jared Lanier presented a critique of software, together with a diagnosis and a proposed remedy, called phenotropic software. The word “phenotropic” derives from the prefix “pheno-”, referring to externals, or surfaces, and “tropic”, meaning interaction. I’ll label these two essays by their publication venues E (for Edge) and B (for Brockman) when quoting from them.

1.1 Critique

Lanier argues that software is unreliable, apart from small programs:

> [I]t's been relatively easy in the history of computer science to make impressive little programs, but hard to make useful large programs. (E)

> Accompanying the parade of quixotic overstatements of theoretical computer power has been a humiliating and unending sequence of disappointments in the performance of real information systems. Computers are the only industrial products that are expected to fail frequently and unpredictably during normal operation. (B)

He accuses computer scientists of idealizing software, creating a gap between what we expect of programs and what they actually deliver:

> Wouldn't it be nicer to have a computer that's almost completely reliable almost all the time, as opposed to one that can be hypothetically perfectly accurate, in some hypothetical ideal world other than our own, but in reality is prone to sudden, unpredictable, and often catastrophic failure in actual use? (E)

1.2 Diagnosis

Lanier attributes the problems to choices made by the earliest thinkers in the field:

> I've had a suspicion for a while that despite the astonishing success of the first generation of computer scientists like Shannon, Turing, von Neumann, and Wiener, somehow they didn't get a few important starting points quite right, and some things in the foundations of computer science are fundamentally askew. (E)

Their early thinking, he argues, was based on the transmission of signals over wires, an idea that does not provide an adequate basis for constructing computational systems as we need them to be:

> If you go back to the original information theorists, everything was about wire communication. ... [This] might not have been the most convenient or cognitively appropriate starting point for human beings who wished to go on to build things. (E)

The limited idea of signals on wires has committed the field to the creation and implementation of protocols, conventions that define the form and meaning of sequences of signals:

> At the dawn of computer science, in the mid—twentieth century, the only available intuition-building experience of information was the sending of pulses down wires. The early versions of information theory, which still dominate the standard curriculum, were concerned with single-point sampling of the world at the end of a wire. Therefore computer architecture as we know it was designed around simulated wires. Source code is a simulation of pulses that can be sent sequentially down a wire—as are passed variables, or messages. (B)

> The way to make pulses on a single wire meaningful is to have a protocol that assigns meaning according to sequence. Most of the first half century of computer science was inspired by such protocols. (B) [emphasis added]

> With temporal protocols, you can have only one of point of information that can be measured in a system at a time. You have to set up a temporal hierarchy in which the bit you measure at a particular time is meaningful based on "when" in a hierarchy of contexts you happen to occupy when you read the bit. You stretch information out in time and have past bits give context to future bits in order to create a coding scheme. (E)

Systems based on protocols have to devote resources to representing the protocols themselves, rather than the content with which the system should deal:

> In order to keep track of a protocol you have to devote huge memory and computational resources to representing the protocol rather than the stuff of ultimate interest. This kind of memory use is populated by software artifacts called data-structures, such as stacks, caches, hash tables, links and so on. (E)

Further, the notion of protocol can only represent simple, limited interactions among the parts of a large information system:

> Clearly protocol adherence is not an efficient means of explaining a system that receives a large number of inputs in parallel, and it is also probably an inadequate method of engineering very large systems. (B)

1.3 Remedy

Lanier proposes that protocols as a way of linking components of a system should be replaced by interactions at “surfaces”, structures containing many bits, that can be examined by other components:

> The alternative [to protocols], in which you have a lot of measurements available at one time on a surface, is called pattern classification. In pattern classification a bit is given meaning at least in part by other bits measured at the same time. (E)

The components “connect to each other by recognizing and interpreting each other as patterns rather than as followers of a protocol that is vulnerable to catastrophic failures (E).” Pattern recognition, Lanier argues, places one in a regime of “approximation rather than perfection”:

> With protocols you tend to be drawn into all-or-nothing high wire acts of perfect adherence in at least some aspects of your design. Pattern recognition, in contrast, assumes the constant minor presence of errors and doesn't mind them. My hypothesis is that this trade-off is what primarily leads to the quality I always like to call brittleness in existing computer software, which means that it breaks before it bends. (E)

Interaction by pattern recognition also opens up the possibility of “evolutionary self-improvement” in the interactions within a system. Each component would be trying to predict the states of other components, and receiving feedback on these predictions, enabling it to get better over time. Lanier argues that we should prefer the robustness of statistical interactions to the brittleness of idealized protocol interactions:

> In the domain of multi-point surface sampling you have only a statistical predictability rather than an at least hypothetically perfect planability. I say "hypothetically", because for some reason computer scientists often seem unable to think about real computers as we observe them, rather than the ideal computers we wish we could observe. Evolution has shown us that approximate systems (living things, particularly those with nervous systems) can be coupled to feedback loops that improve their accuracy and reliability. They can become very good indeed. (E)

1.3.1 Example

In (B) Lanier sketches an example of how phenotropic software might work. He envisions two teams of researchers, initially working independently, one of which develops a simulation of a lung, and one a simulation of a heart. The teams then decide to couple their simulations.

In the protocol regime of today, each team would define a protocol, likely an API, that the other team could use for interaction. The API would define what information the other team could access, and what control signals the other team could send.

Lanier describes two problems with this approach. First, he is skeptical that protocols could be defined that would cope with the complexity of the needed interactions between heart and lung. But, even if it were possible to define a workable protocol, “A working protocol would almost certainly reduce the prospects for improving any of the constituent organ simulations it connects.” The ideas about their simulations that the teams had at the time they defined the protocol could never be changed:

> [S]oftware builds up in layers; it would be impossibly complicated and expensive to exhume protocols that have already been relied upon by many users in different ways. Thus we have the phenomenon of “lock-in," in which some software becomes effectively mandatory. ... Beyond lock-in is an even more annoying software characteristic, which I’ve dubbed “sedimentation.” Software sedimentation is a process whereby not only protocols but the ideas imbedded in them become mandatory. ... As soon as the engineering groups have signed on to a protocol, the protocol becomes their master, since the groups would have to change simultaneously in order to revise it, and that would be effectively impossible because of the expense and complexity of the

> task. Whatever ideas about interorgan communication are in vogue at the time of the protocol's invention will be sedimented into place. Thinking will stop.

The phenotropic alternative is for each simulated organ to

> pretend... that the other was a real, physical organ being probed by real sensors. Each organ could measure fundamental properties in the other, such as temperature, pressure, and chemical constituents at points in time and space. Each organ would appear to the other as a surface that could be sampled to varying degrees, but no higher-level parameters would pass between them. There would be no protocol other than the low-level one dictated by the nature of possible physical measurements.
>
> For this scheme to work, each team would have to learn to recognize patterns in the other's simulation. The heart would no longer be able to send a message that it had performed a beat. That would have to be inferred by the lung from such things as fluid motion and tissue displacements.

Lanier suggests that this paradigm could be extended to large software systems generally:

> Perhaps in the future there will be an operating system whose components recognize, interpret, and even predict each other. Such a system would be less prone to catastrophic failure.

2. What has happened since 2003?

2.1 The immediate response

The essay (E) was accompanied by commentary, some of which was quite skeptical. The philosopher Daniel Dennett commented,

> There are a few interesting ideas in his ramblings, but it's his job to clean them up and present them in some sort of proper marching order, not ours. Until he does this, there's nothing to reply to. (E)

Other critical commentary assumed Lanier to be talking about parallel computation, and answered what they saw as his arguments for that approach. Lanier responded that this criticism had "misfired", because he was not talking about parallel computation; in fact (he points out) the word "parallel" does not appear in E.

The historian George Dyson suggested that Lanier was wrong to blame our problems on lack of vision among the founders of computer science (see also Note 1):

> [I]t is unfair to attribute to Alan Turing, Norbert Wiener, or John von Neumann (& perhaps Claude Shannon) the limitations of unforgiving protocols and Gordian codes. These pioneers were deeply interested in probabilistic architectures and the development of techniques similar to what Lanier calls phenotropic codes. The fact that one particular computational subspecies became so successful is our problem (if it's a problem) not theirs.
> ...
> The pioneers of digital computing did not see everything as digitally as some of their followers do today. "Besides," argued von Neumann in a long letter to Norbert Wiener, 29 November 1946 (discussing the human nervous system and a proposed program to attempt to emulate such a system one cell at a time), "the system is not even purely digital (i.e. neural): It is intimately connected to a very complex analog (i.e. humoral or hormonal) system, and almost every

feedback loop goes through both sectors, if not through the 'outside' world (i.e. the world outside the epidermis or within the digestive system) as well." (E)

2.2 Later publications

Lanier has mentioned phenotropic software here and there since 2003, on his Web site (http://www.jaronlanier.com/), in some interview material in Rosenberg (2007), and, very briefly, in You Are Not a Gadget (Lanier, 2010). Google Scholar returned 104 hits for “phenotropic” on June 14, 2018, but most of these refer to unrelated uses of the term in pharmacology. Of those papers that refer to Lanier’s idea, some contain only brief mentions or summaries of Lanier’s work (Feiner et al., 2004; Ommeln, n.d.; Juba, 2011; Hissam et al., 2016).

A cluster of papers by Frénot, Hu, Privat, and colleagues applies the concept of phenotropic interaction to communication within a network of sensors and effectors (Hu, et al., 2011; Hu, et al., 2012; Privat, 2012; Hu, 2014). Such communication can be based on sensing physical quantities rather than on interpreting protocols, in some situations, as Lanier suggested. For example, an appliance like an oven can be identified by an energy management system by its pattern of energy consumption, with no participation of the oven in any protocol. The authors show how this approach can lead to greater flexibility in setting up and extending a network of devices. Some later work builds on these ideas (Jerde, 2017). Gabriel (2006) and Gabriel and Goldman (2006) include phenotropics as a mechanism for looser, more forgiving couplings between system components, as Lanier proposed.

Fleissner and Baniassad (2008) propose harmony-oriented architecture, inspired in part by the idea of phenotropic software. Systems in this framework are composed of snippets of code that communicate by diffusion of information within a spatial layout. Fleissner and Baniassad (2009 described a dialect of Smalltalk that works in this way. Martin (2011) describes a prototype implementation using Javascript. While this work goes some way to loosen the coupling between system components (some aspects of coupling can be changed by moving components without rewriting them) information exchanged between components has to be tagged, contrary to the phenotropic idea. The ideas of interaction by recognition, and the improvement of recognition and prediction over time, are not yet included.

3. Connections to ideas of Basman and colleagues.

In a series of papers at PPIG and elsewhere, Antranig Basman and colleagues have presented ideas that align in some ways with those of Lanier, as critique and diagnosis (Basman et al. 2015; Basman, 2016; Basman et al., 2016; Basman, 2017; Clark and Basman., 2017; Basman et al., 2018).

3.1 Critique

In “Building Software is Not [Yet] a Craft”, Basman (2016) notes that in the “Computer Science of the Present, a product may spontaneously disintegrate without warning, suddenly becoming wholly unusable.” Compare Lanier (as quoted earlier) on software being “prone to sudden, unpredictable, and often catastrophic failure in actual use (E).” In the same essay, Basman also cites the “the potential for the complexity of artefacts to exceed our ability to manage or comprehend them;” compare Lanier (B), “Since the complexity of software is currently limited by the ability of human engineers to explicitly analyze and manage it, we can be said to have already reached the complexity ceiling of software as we know it. “

In “If What We Made Were Real”, Basman (2017) calls for “software which is real, in that it behaves with the same continuity and consistency as real materials — real trees and real mountains expose a consistent and coherent set of linked aspects, affordances and appearances as we move from place to place, scale to scale and sense to sense.” Compare Lanier’s description (quoted earlier) of the heart and lung simulations that relate as if each “was a real, physical organ being probed by real sensors. Each

organ could measure fundamental properties in the other, such as temperature, pressure, and chemical constituents at points in time and space. (B)”

3.2 Diagnosis

In one respect Basman’s and Lanier’s diagnoses are virtually identical, though arrived at independently. Lanier (E):

> The reason we're stuck on temporal protocols is probably that information systems do meet our expectations when they are small. They only start to degrade as they grow. So everyone's learning experience is with protocol-centric information systems that function properly and meet their design ideals. This was especially true of the second generation of computer scientists, who for the first time could start to write more pithy programs, even though those programs were still small enough not to cause trouble. Ivan Sutherland, the father of computer graphics, wrote a program in the mid 1960s called "Sketchpad" all by himself as a student. In it he demonstrated the first graphics, continuous interactivity, visual programming, and on and on. Most computer scientists regard Sketchpad as the most influential program ever written. Every sensitive younger computer scientist mourns the passing of the days when such a thing was possible. By the 1970s, Seymour Papert had even small children creating little programs with graphical outputs in his computer language "LOGO". The operative word is "little." The moment programs grow beyond smallness, their brittleness becomes the most prominent feature, and software engineering becomes Sisyphean.

Compare Basman (2017):

> We got into this mess through 60 years of consistently drawing the wrong people into our field, and continuing to entrench its vices rather than reform them. In the “Garden of Eden” phase of Computer Science when such inspired products as McCarthy’s Lisp and Sutherland’s Sketchpad were plentiful as tabby cats, it was easy to imagine that maturing to solve more ambitious problems for a wider class of people was just a step away. In a world that has given us Java, Haskell and Ruby, success seems further away than it ever has been.

3.3 Remedy

As we’ve seen, the key elements of Lanier’s remedy are interaction based on surface interactions, rather than protocols, and self-improving recognition as the basis of communication. Basman and colleagues emphasize different themes, especially support for open, ongoing authoring of software systems (Basman et al., 2015; Basman et al., 2016). But there are nevertheless some points of agreement.

Lanier’s idea of system components “pretending” to be physical objects, and interacting by measurement rather than by protocol, is related to Clark and Basman’s (emphasis on externalized state transfer between system components. Traditional software components ordinarily conceal as much of their state as possible; a normal role for APIs is to expose as little state as possible. Clark and Basman argue that components should instead expose as much state as possible. Physical objects, or software components “pretending” to be physical objects, do the same. Lanier’s cautionary story about the fate of the heart-lung simulation, “locked in” and “sedimented” once protocols are in place, can also motivate Clark and Basman’s (2017) open approach.

In “An Anatomy of Interaction”, Basman et al. (2018) present a vision of a system in which system state mirrors the state of some portion of “the world of interest”. They go on:

> opening up the design process requires this mirroring to have an open structure. Each community of interest will have di erent concerns that lead them to select di erent sources of state, and mirror them with di erent representations. We require that both the description of the state to be materialised, as well as the materialised state itself, are externalised [in a way] that is available throughout the lifetime of the system, not just an early design phase.

This is a description of system components that are "pretending" to be physical objects, in Lanier's terms, and carrying on pretending, and being open to interaction as such.

Physical objects can not only be observed but also be modified. In "If What We Made Were Real", Basman (2017) contrasts the way this can be done, in physical craft activities, with how today's software behaves when worked on:

> Scratch the surface of a physical product such as a chair or a wall, and you find something broadly similar underneath. The physical world is worked on by means of tools that are part of its own idiom — whether we cut a piece of wood into a smaller piece of wood, or make a hole to hold a bracket, we are using the affordances of the world itself to cause change. Contrast this with the nature of a modern piece of software or hardware — scratch the surface and underneath it is an incomprehensible world of blinking lights and mass of wiring that bears no resemblance to the physical form and affordances of the overall object (see Note 2).

Further, Basman (2016) describes how physical materials respond to change in a continuous, rather than a discrete way:

> Each state of the material is closely surrounded by a dense collection of neighbouring states that behave similarly. ... However, today's software materials, traditionally consisting of source code text in an editor buffer, could not be more sparse. The "nearest neighbouring program" to any given one is separated from it by a vast ocean of syntactically invalid or crashing variants (see Note 3).

As quoted earlier, the virtues of "approximation" for Lanier require a related continuity, though not expressed explicitly:

> When you de-emphasize protocols and pay attention to patterns on surfaces, you enter into a world of approximation rather than perfection. ... Pattern recognition ... assumes the constant minor presence of errors and doesn't mind them. (E)

The link to continuity is that "approximation" is only valuable if things that are similar in perception are similar in meaning, so that small errors don't matter.

While we can see some resemblances between Lanier's conceptions and those of Basman and his collaborators, there are differences in approach. As mentioned earlier, Basman and collaborators focus their efforts on supporting more flexible authorship of software, by people. By reducing the barriers to understanding and modifying software (often deliberately erected) they hope to create an ecosystem of software that develops more organically, and fails less catastrophically. Problems can be repaired by moving a little way to a nearby state, without requiring wholesale reworking.

By contrast, Lanier hopes that recognition-based interfaces between system components will improve themselves, as each component seeks to develop a better and better model of the behavior of its neighbors. There's no explicit role for human authors in this picture. Lanier hopes that software engineers working with components coupled in the proposed more robust way will find that the systems they build will turn out to be more robust.

One way to probe the prospects for this hope is to explore the idea of "pretending" to be a physical object. If Lanier's heart simulator actually were a heart, it would exhibit the continuous response to intervention that physical objects have. Looking behind its surface, the structure that it presents to the lung simulor, one would find real physical substance, as Basman notes. But if the simulator is only "pretending" to be a heart, that is, only displaying the right behavior on its surface, what then?

One possibility is that one would find the "incomprehensible world of blinking lights and mass of wiring" of today's software behind the surface. In that case the support for human tinkering within the "pretending" heart simulator would be no different from what is available now. The benefits of the approach would come from looser coupling outside the simulator, together with the prospect of automatic improvement in this coupling.

Alternatively, one might find other surfaces behind the outer surface of the simulator. Lanier says little about this. He does propose that there could be "an operating system whose components recognize, interpret, and even predict each other, (B)". This suggests that smaller, as well as larger, software components could be phenotropic. We return to this possibility below.

Another aspect of Lanier's proposal is that phenotropic interfaces are meant to improve themselves, without the human authorship with which Basman and colleagues are concerned, by a process of feedback-guided evolution. Lanier sketches how this could work in E, using the example of a system that creates a graphical model of a person, an avatar, from video, recovering the geometry of the person, and how they are moving in space, relative to the camera. There's also a system that uses this model to produce video that shows the avatar moving. If both systems are successful, the video of the avatar should closely resemble the original video. Lanier proposes that the system that builds the avatar can improve its processing of the input video by using feedback about the video reconstructed from the avatar.

The avatar example contains an encoder-decoder pair: the system that creates an avatar from video, and the system that creates a video from the avatar. Such pairs can indeed tune themselves, as is common in machine learning work (see e.g. Cho et al, 2014). How would this work in other situations?

Lanier applies the idea to the heart-lung simulation:

> Each team would also learn to build a model of the other organ, in order to assist in interpreting measurements. These models might not exist as independent, separable structures hut might be implicit in the chosen signal-processing methods, and would almost certainly be able to retune themselves with use. (B) [emphasis added]

Here there is no preexisting decoder, corresponding to the system that produces video from avatars in the avatar example. Lanier may be suggesting that the connection between the heart and lung simulations is an autoencoder (https://en.wikipedia.org/wiki/Autoencoder); these can indeed tune themselves. The autoencoder would map the information available on the surface of the heart to a lower dimensional representation, with the requirement that the map can be inverted with small loss. By including temporal information the autoencoder might recover a dynamic model of the heart adequate to predict the timing of changes on the heart's surface, that might correspond to beats. Thus, as pointed out earlier, no protocol would be needed to communicate beats from heart model to lung model

It's not clear what work this does, though. The lung model does not need to know about beats, as such. It needs to know the pressure produced by the heart as it pumps, and has to relate this pressure to the pressures induced in the lung. But Lanier already requires that each model can "measure fundamental properties in the other, such as temperature, pressure, and chemical constituents at points in time and

space", as said earlier. Given that these measurements are available, the need for recognition, and thus the value of retuning, isn't clear.

4. The psychology of phenotropic programming

Lanier's root concern is that when two systems try to communicate they should succeed despite failures of detail. The appeal of "recognition" is that it succeeds in the presence of noise. The appeal of retuning is that the ability of recognition to reject noise can improve over time. The appeal of "measurement" is that it suggests communication that is tolerant of errors; quantities can be measured approximately without losing all meaning.

If we take the idea of "surface" to be simply a structure to which it makes sense to apply recognition, which seems broadly consistent with Lanier's thinking, we can see examples of phenotropic communication in systems today, especially at the user interface. Consider looking up a book in a library catalog. I've just verified that I can search for books by "jaron lanier", "lanier, jaron", "lanier jaron", "Jaron Lanier", or "Lanier, Jaron" in my library catalog, and get the same results. If I ask for books by "lanier jared", the catalog asks if I meant "lanier jaron". This is a concrete instance of communication succeeding in the presence of noise. The catalog is not simply searching for what I specified, it is recognizing what I may have meant. What I typed is the "surface" of me that the catalog responds to.

Amazon's surface for my book searches is considerably bigger. If I log in, as I've done just now, it finds books for me that I didn't search for at all, based on aspects of my history of which it is aware. One of the suggestions is actually a book a friend just recommended for me; I have no idea what was on my surface that enabled that.

Retuning is constantly at work in Amazon's recognition system. This isn't autoencoder tuning, but a kind of reinforcement learning: Amazon's recommendations, that is, its recognition of my interests, are reinforced when I act on one of them. The recognition system is tuned by my responses, and those of jillions of other consumers.

Can these ideas get any traction within the realm of programming? A few crumbs of this kind of behavior have long been present. The ability to write 1 + 2.5 rather than 1. + 2.5 was not supported in FORTRAN at first. If you intended 1 to be interpreted as a floating point quantity you needed to write a floating point literal, with a decimal point. Happily nearly all compilers now "recognize" that someone who writes 1 + 2.5 wants floating point arithmetic, and insert the needed conversion.

But the surface involved here is miniscule, and corresponding "coercions", or automatically inserted type conversions, are quite rare in other situations. Nearly always a programmer has to do the work of modifying an intention to fit the precise form of an allowed request. This is the kind of protocol fiddling that Lanier wants to eliminate.

For example, suppose one has a function, foo, that takes two coordinates, x and y, as arguments. If one has a point, p, given as a pair of coordinates, in most languages one can't simply provide the point as an argument to foo. Rather, one has to write something like foo(p[0],p[1]). If one's interaction with foo was conducted via a surface, foo might be able to recognize what to do, if just p appears on the surface.

Could foo's ability to respond as desired to a range of surface contents improve over time? It doesn't seem that the autoencoder idea, as an unpacking of Lanier's intention, can work here. But reinforcement learning, of the kind Amazon uses, could be applied. If foo's conjectured recognition can be approved or rejected by the programmer, its ability can be tuned. Such tuning could take place for a whole community of programmers, just as Amazon's tuning uses results from many consumers, or it could be individualized.

For example, foo might learn something reasonable to do if given just a single number (duplicate it? pair it with a zero?).

The surface foo examines could be broadened beyond the arguments specified for it, to include the surrounding code. In the example above, when foo is given a single number, the best response could depend on what else is happening in the code.

This small example stands for a much larger, more consequential population. In many programs the number of characters that correspond to the programmer's overall intentions, those that relate to entities in the outside world, is quite small, relative to the size of the program. That is, a great deal of work goes into internal considerations, including many variations on the theme of this example, which is putting things into structures and taking them out again. These are the "stacks, caches, hash tables, links and so on" whose necessity Lanier attributes to protocol-centric computing. What might be called the payload ratio problem is that too much software doesn't express the purpose of code, but just the implementation of it in a specific setting (see Note 4.)

Would software made up of components like foo have the merits of physicality, that Lanier and Basman and his colleagues call for? The answer may depend on the stability of the resulting recognition behaviors. What's wanted is a regime in which similar programs do similar things, yes, but also what they do is predictable. Part of the appeal of measurement in Lanier's thinking is that the meaning of a gram or a second, unlike the meaning of a pointer, does not change over time. Software structures anchored in these stable ideas can themselves be stable. Can phenotropic software structures that tune their behavior over time be stable, too?

The perspective of the payload ratio problem suggests that they might. Because programmers' intentions are often grounded in entities in the world, a facility that squeezes out other stuff from the code, as phenotropic structures may do, may leave a residue that is more stable, and easier to understand. This could be true despite the fact that the implementation of these structures, relying as it would on complex and even dynamic recognition behavior, would be very complicated, and indeed opaque. But this situation is actually quite natural, and familiar; it's what we experience with our fellow humans, and we are famously more robust than the software systems of today.

Notes

Note 1. In his commentary on E, Dyson suggests that the brittleness and unreliability of software, of which Lanier complains, may actually be virtues:

> I'm not immersed in the world of modern software to the same extent as Jaron Lanier, so it may just be innocence that leads me to take a more optimistic view. If multi-megabyte codes always worked reliably, then I'd be worried that software evolution might stagnate and grind to a halt. Because they so often don't work (and fail, for practical purposes, unpredictably, and in the absence of hardware faults) I'm encouraged in my conviction that real evolution (not just within individual codes, but much more importantly, at the surfaces and interfaces between them) will continue to move ahead. (E)

This perspective builds on ideas Dyson presents in his book, Darwin among the Machines (1997), about the evolution of computational structures. While these structures are not self reproducing, and not free living, they replicate with enormous fecundity within an environment, increasingly shaped by them, that is increasingly hospitable to them. Where once code could execute in only a handful of processors, and could be replicated only with considerable effort, much less than a century from its first appearance, code can travel around the world at the speed of light, and execute in millions of hosts. Just as random

variability fuels biological evolution, so brittleness and bugs may be important to the longer-term development and spread of computational structures:

> No one can say what contribution randomness has made to software development so far. Most programs have grown so complex and convoluted that no human being knows where all the code came from or even what some of it actually does. Programmers long ago gave up hope of being able to predict in advance whether a given body of code will work as planned. ... The software industry has kept track of harmful bugs since the beginning -- but there is no way to keep track of the accidents and coincidences that have accumulated slight improvements along the way. -- Dyson (1997), p. 124.

Note 2. In E Lanier describes today's software as "simulations of vast tangles of telegraph wires".

Note 3. Attaining the desired denseness, or continuity, in a space of software necessarily requires giving up much of the "power" of traditional computation, that is, the "power" to have arbitrarily small changes cause arbitrarily big effects. In "If what we made were real" (2017) Basman lists other powers that should be sacrificed, including "The power to construct programs that might consume unbounded time and/or space, or perhaps never terminate," and "The power to prescribe the exact sequence of operations needed to achieve a particular result."

Note 4. Bret Victor deals with these matters in his brilliant (and poignant) historical pastiche, The Future of Programming (2013), speaking as if in 1973:

> So, say you've got this network of computers, and you've got some program out here that was written by somebody at some time in some language; it speaks some protocol. You've got another program over here written by somebody else some other time, speaks a totally different language, written in a totally different language. These two programs know nothing about each other. But at some point, this program will figure out that there's a service it needs from that program, that they have to talk to each other. So you've got these two programs—don't know anything about each other—written in totally different times, and now they need to be able to communicate. So how are they going to do that? Well, there's only one real answer to that that scales, that's actually going to work, which is they have to figure out how to talk to each other. Right? They need to negotiate with each other. They have to probe each other. They have to dynamically figure out a common language so they can exchange information and fulfill the goals that the human programmer gave to them. So that's why this goal-directed stuff is going to be so important when we have this internet—is because you can't write a procedure because we won't know the procedures for talking to these remote programs. These programs themselves have to figure out procedures for talking to each other and fulfill higher-level goals. So if we have this worldwide network, I think that this is the only model that's going to scale. What won't work, what would be a total disaster, is—I'm going to make up a term here, API [Application Programming Interface]—this notion that you have a human programmer that writes against a fixed interface that's exposed by some remote program. First of all, this requires the programs to already know about each other, right? And when you're writing this program in this one's language, now they're tied together so the first program can't go out and hunt and find other programs that implement the same service. They're tied together. If this one's language changes, it breaks this one. It's really brutal, it doesn't scale. And, worst of all, you have—it's basically the machine code problem. You have a human doing low-level details that should be taken care of by the machine. So I'm pretty confident this is never going to happen. We're not going to have API's in the future. What we are going to have are programs that know how to figure out how to talk to each other, and that's going to require programming in goals.

References

Basman, A., Clark, C., & Lewis, C.(2015) Harmonious Authorship from Different Representations (Work in Progress). In M. Coles and G. Ollis (Eds) Proc. PPIG 2015 Psychology of Programming Annual Conference. Bournemouth, England, 15th-17th July 2015.

Basman, A. (2016). Building Software is Not [Yet] a Craft. Proceedings of the Psychology of Programming Interest Group.

Basman, A., Church, L., Klokmose, C., & Clark, C. (2016). Software and how it lives on-embedding live programs in the world around them. Proceedings of the Psychology of Programming Interest Group.

Basman, A. (2017) If What We Made Were Real: Against Imperialism and Cartesianism in Computer Science, and for a discipline that creates real artifacts for real communities, following the faculties of real cognition. Proceedings of the Psychology of Programming Interest Group.

Basman, A., Tchernavskij, P., Bates, S., & Beaudouin-Lafon, M. (2018). An Anatomy of Interaction: Co-occurrences and Entanglements. In Proceedings of ACM Conference, Nice, France, April 2018 (Salon des Refuses '18)

Cho, K., Van Merriënboer, B., Bahdanau, D., & Bengio, Y. (2014). On the properties of neural machine translation: Encoder-decoder approaches. arXiv preprint arXiv:1409.1259.

Clark, C., & Basman, A. (2017). Tracing a paradigm for externalization: Avatars and the GPII Nexus. In Companion to the first International Conference on the Art, Science and Engineering of Programming (p. 31). ACM.

Dyson, G. B. (1997). Darwin among the machines: The evolution of global intelligence. Basic Books.

Feiner, S., Ganapathy, S. K., Lanier, J., Levin, G., White, D., & Pingali, G. (2004, October). Directions and frameworks for effective telepresence. In Proceedings of the 2004 ACM SIGMM workshop on Effective telepresence (pp. 69-72). ACM.

Fleissner, S., & Baniassad, E. (2008, October). Towards harmony-oriented programming. In Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications (pp. 819-822). ACM.

Fleissner, S., & Baniassad, E. (2009, October). Harmony-oriented programming and software evolution. In Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications(pp. 991-998). ACM.

Gabriel, R. P. (2006, March). Design beyond human abilities. In Aspect-oriented software development: Proceedings of the 5 th international conference on Aspect-oriented software development (Vol. 20, No. 24, pp. 2-2).

Gabriel, R. P., & Goldman, R. (2006, October). Conscientious software. In Acm Sigplan Notices (Vol. 41, No. 10, pp. 433-450). ACM.

Hissam, S., Klein, M., Moreno, G., Northrop, L., & Wrage, L. (2016). Ultra-Large-Scale (ULS) systems: Socio-adaptive systems. Software Engineering Institute, Carnegie-Mellon University.

Hu, Z., Frénot, S., Tourancheau, B., & Privat, G. (2011). Iterative Model-based Identification of Building Components and Appliances by Means of Sensor-Actuator Networks. 2nd Workshop on eeBuildings Data Models, CIB W078 - W102, Sophia Antipolis - France. 2011.

Hu, Z., Privat, G., Frénot, S., & Tourancheau, B. (2012, October). Representation and self-configuration of physical entities in extended smart grid perimeter. In Innovative Smart Grid Technologies (ISGT Europe), 2012 3rd IEEE PES International Conference and Exhibition on (pp. 1-7). IEEE.

Hu, Z. (2014). Self-configuration, monitoring and control of physical entities via sensor and actuator networks (Doctoral dissertation, Université Claude Bernard-Lyon I).

Jerde, R. D. (2017). Follow the Silk Road: how Internet affordances influence and transform crime and law enforcement (Doctoral dissertation, Monterey, California: Naval Postgraduate School).

Juba, B. (2011). Universal semantic communication. Springer Science & Business Media.

Lanier, J. (2003). Why gordian software has convinced me to believe in the reality of cats and apples. edge. org, November, 1. Available online at https://www.edge.org/conversation/jaron_lanier-why-gordian-software-has-convinced-me-to-believe-in-the-reality-of-cats

Lanier, J. (2002) The complexity ceiling. in Brockman, J. (Ed.). (2002). The next fifty years: Science in the first half of the twenty-first century. Vintage. pp 216-229.

Lanier, J. (2010). You are not a gadget: A manifesto. Vintage.

Martin, K. A., "Harmony Oriented Architecture" (2011). Electronic Theses and Dissertations. 1766. http://stars.library.ucf.edu/etd/1766

Ommeln, M. (n.d.) Epistemology in the Virtual Reality.–Friedrich Nietzsche and the Natural Science. Or: Navigation at the Contradiction with the Concept of Nietzsche's "Contrary Character of the Existence".

Privat, G. (2012). Phenotropic and stigmergic webs: the new reach of networks. Universal Access in the Information Society, 11(3), 323-335.

Victor, B. (2013) The future of programming. https://vimeo.com/71278954 Transcript at http://glamour-and-discourse.blogspot.com/p/the-future-of-programming-bret-victor.html