RC24510 (W0803-007) March 3, 2008
Computer Science

IBM Research Report

Design Beyond Human Abilities

Richard P. Gabriel
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

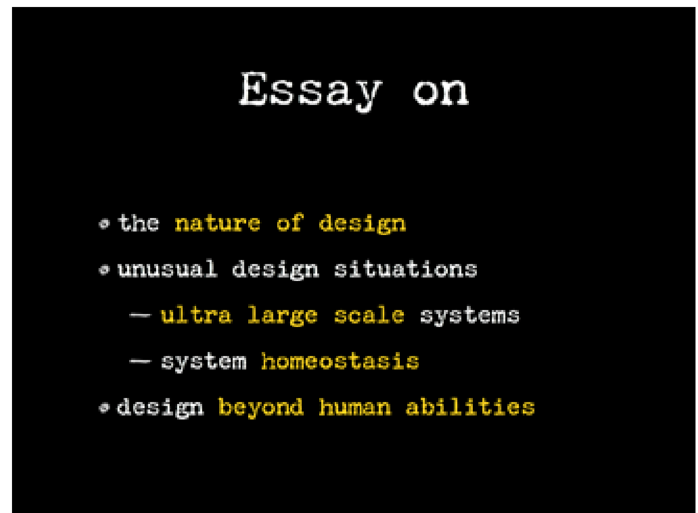
Design Beyond Human Abilities

Richard P. Gabriel
IBM Research

On November 16, 2007, I presented a talk on Design as my Wei Lun Public Lecture at the Chinese University of Hong Kong. This is a revised and reimagined transcript of that talk.

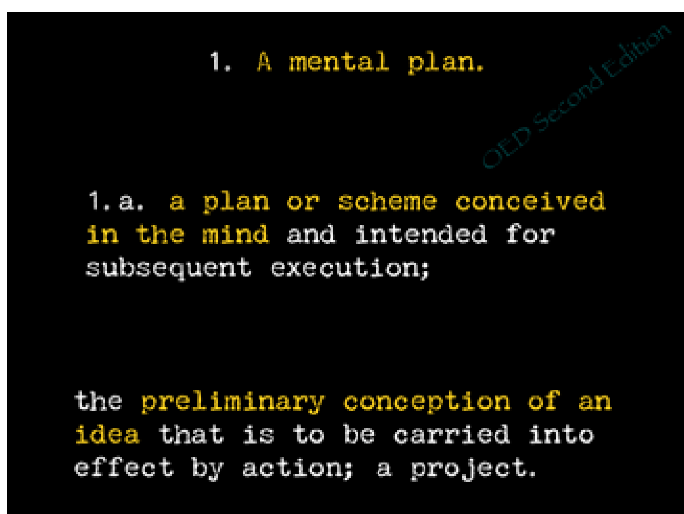
This talk is an essay on design. In the 16th century, Michel de Montaigne invented a new genre of writing he called an *essai*, which in modern French translates to *attempt*. Since then, the best essays have been explorations by an author of a topic or question, perhaps or probably without a definitive conclusion. Certainly in a good essay there can be no theme or conclusion stated at the outset, repeated several times, and supported throughout, because a true essay takes the reader on the journey of discovery that the author has or is experiencing.

This essay—on design—is based on my reflections on work I've done over the past 3 years. Some of that work has been on looking at what constitutes an "ultra large scale software system" and some on researching how to



keep a software system operating in the face of internal and external errors and unexpected conditions.

In this presentation/essay I'll look at the nature of design through the lenses these two inquiries provide. Namely, what can we learn of design when we look at an extreme design situation, and when we look at how to give a system the characteristic of *homeostasis*: the ability of a system to regulate its internal environment to maintain a stable, constant condition. In short, the exploration is what we can learn about design when we examine design situations that are beyond (current) human abilities.



As many do, I begin this inquiry by looking at a dictionary definition. Here is the definition of "design" from the Second Edition of the Oxford English Dictionary. I also looked at definitions provided by others. Here is the definition Guy Steele used in his famous "Growing a Language" talk at OOPSLA in 1999. (In this talk he illustrated his points by using only words of 1 syllable unless he had already defined them using words he was permitted

(...Let me add that better means "more good.")

A design is a plan for how to build a thing. To design is to build a thing in one's mind but not yet in the real world – or, better yet, to plan how the real thing can be built.

–Guy L. Steele Jr., "Growing a Language"

Design is the thinking one does before building.

to use at that point in his lecture.) The short definition is one I've used in the past. I intended it to imply that a design is not a particular artifact or act of producing one, but only whatever thinking one does in anticipation of building—I wanted to include the activities of what were once called "hackers" or expert programmers who seem to do little planning before they type in code; instead their designing is based either on a lot of experience and skill, or on their ability to think quickly (and in silence) about the problem. Moreover, I wanted my definition to more clearly imply than others do that the thinking can be interwoven with building.

Carliss Baldwin and Kim Clark are in the Harvard Business School, and their definition is from work where they were looking at how modularity of design can lead to new markets. In the case study that made them famous, they looked at how the PC industry was made possible by the modular hardware design that IBM produced in the early 1980s. This has led to research on how to place a monetary value on part of a design.

Designs are the instructions based on knowledge that turn resources into things that people use and value.

–Carliss Y. Baldwin & Kim B. Clark, "Between 'Knowledge' and 'the Economy': Notes on the Scientific Study of Designs"

- ideas / mental plan / knowledge
- problems
- value / purpose
- before-building time

What these definitions have in common is their reference to ideas and a mental plan, hints at problems that need to be solved, purposes, and planning before building.

Now let's look at the first of these two extreme design situations: ultra large scale software systems. This exploration is based on work I did in conjunction with the Software Engineering Institute at Carnegie-Mellon University as part of a study commissioned for the US Army, which was interested in setting a research agenda that would enable the creation of very, very large systems. The individual who sponsored the study—Claude Bolton—believed that the software community is currently unable to build what he considers small or medium scale software systems, and so he doubted that our abilities would automatically get better as the scale increased. Therefore, he believed that we would need to create a research agenda that directly addressed the scale issues. The SEI study looked at characteristics of ultra large scale systems (ULS systems), the challenges one would encounter when designing and constructing them, and the research needed to address those challenges. The re-

Imagine,
if you will,...

- trillions of lines of code (1e12)*
- millions of computers / sensors (1e6)
- real-time requirements
- life-critical applications
- adversaries

* This is exaggerated for effect. Grady Booch estimates that about 800 billion lines of code have been written since 1945.

search agenda we produced would need about 25 years to reach maturity (`http://www.sei.cmu.edu/uls/`). In the description of ULS systems, we talk about adversaries, which is intended to refer both to outside agents who wish to hamper, harm, or disable a running system, as well as to environmental factors like faults or bugs, power failures, hardware failures, climate and weather, etc. Note that in the commercial world, an adversary can be a competitor, a virus writer, a spammer, a spoofer, or any other security risk.

In this presentation I talk about trillions of lines of code in order to emphasize a scale way beyond what we think of as remotely feasible today. This is an exaggeration because Grady Booch has estimated that collectively, humankind has produced a total of about a trillion lines of code since programming as we know it began in 1945.

The phrase "imagine if you will…" refers to a TV series in the US which ran from 1959 until 1964 called "The Twilight Zone." The show was a fantasy or science fiction anthology hosted by the writer Rod Serling. Each episode was the enactment of a short story with a mind-bending premise or situation such as time travel or otherworldly aliens or magic. Many times the introduction to the show included the phrase "imagine if you will," which introduced the situation, and also always included the following, perhaps rephrased a little over the run of the show:

> *There is a fifth dimension, beyond that which is known to man. It is a dimension as vast as space and as timeless as infinity. It is the middle ground between light and shadow, between science and superstition. And, it lies between the pit of man's fears and the summit of his knowledge. This is the dimension of imagination. It is an area which we call the Twilight Zone.*

In the study we found that the realities of ULS systems would seem like the Twilight Zone to most software engineers and computer scientists, and in fact, though the study was and remains provocative, most main-

ULS Systems

an ultra-large-scale (ULS) system is one that is impossible to build* because it exceeds some critical limit of today's software engineering technology

*today

ULS Systems

the purpose of the system is to give its users significant advantages against intelligent, motivated, and capable adversaries (foes, virus writers, competitors), who also have access to technology of their own

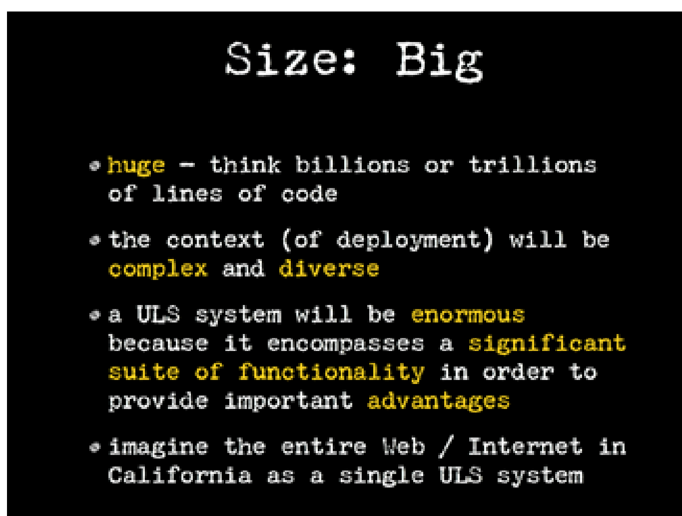
stream researchers, practitioners, and research funders believe that as difficult as the challenges are for creating and maintaining ULS systems, industry will solve those problems as a matter of course. Judge for yourselves.

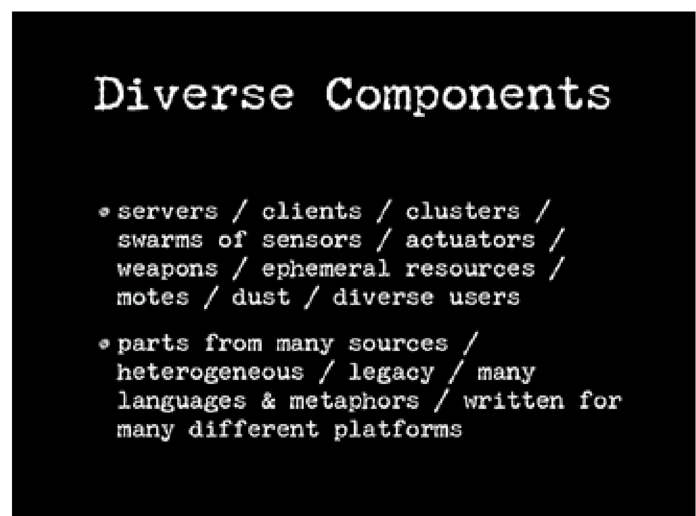
The definition we first used for ULS seemed to capture a fundamental characteristic: that a ULS system was one that we could not now build. This definition, though, foundered because it held out a ULS system as something always beyond what could be built—a ULS system was whatever happened to be just over the horizon.

The reason a ULS system is so large is that it provides lots of functionality, typically in a focused area of concern. An example of a ULS system (that doesn't yet exist) would be a healthcare system that integrates not only all medical records, procedures, and institutions, but also gathers information about individual people continuously, monitoring their health and making recommendations about how to improve it or keep it good. Medical researchers would be hooked into the system, so that whenever new knowledge appeared, it could be applied to all who might need it. Imagining this system, though, requires also imagining that it is protected from the adversaries of governmental and commercial spying / abuse.

We now look at the characteristics that make ULS systems seem like a twilight zone to many mainstream software engineers and software researchers.

The most important characteristic is that a ULS system is huge, which means that its deployment must be complex and diverse—it is not a room full of blades running Java applications. It's enormous because it provides a lot of functionality. For size, think of the entire Internet in California—but don't think of this portion of the Internet as a ULS system. This portion of the Internet doesn't support a coherent set of functionality—it's a collection, and its size is by aggregation, not by what would appear to be a design. The population of California, for comparison purposes, is about 37 million people.





The components that make up a ULS system are diverse as well as many, ranging from servers and clusters down to small sensors, perhaps the size of motes of dust. Some of the components will self-organize like swarms, and others will be carefully designed. The components will not only be computationally active but also physically active, including sensors, actuators, and in some cases weapons or other mechanisms of physical self-defense. The hardware and software parts will come from diverse sources, will not homogeneous, and will originate in different times and places. Some of the components will be legacy. If we will see a true ULS system in 20 years, some of it has already been written—perhaps or probably by people who don't believe that ULS systems can be built. Different kinds of people will be using the system at the same time and over time.

The software components will be written in a variety of languages and for a variety of hardware platforms. Even the computational metaphors will be many and diverse—don't expect there to be only object-oriented architectures.

The components will come and go, joining and leaving the system in both planned and unplanned ways. Parts of it will need to be real-time, some of it will be embedded, most of it distributed.

The system will last for a long time—something that large won't be used once or briefly. It probably won't be installed in the sense we know today, but will grow in place while running. People won't run `make` to produce it

Ephemerality

- some parts (hardware, software, versions, connections) will be joining and leaving the system
- some of it / all of it:
  - real time
  - embedded
  - distributed

Longevity

- installed (at most) once
- some parts will be very old - 25+ years
- system must withstand infrastructure technology changes

or fix bugs. Some parts will be very old (at least 25 years old, just as is the case today with some of our systems). This means that a ULS system needs to survive infrastructure and technology changes.

Because, in general, there will be adversaries (competitors, changing and dangerous environments, perhaps literal enemies), the ULS system will be part of an ongoing arms race where the design and realization of the system will be evolving over time and sometimes simply changing. The operating environment will also be changing (different and new hardware, changing topologies and geometries, etc) and complex, and therefore the ULS system must be able to adapt to these changes.

Continual Change

- an arms race (with adversaries) where the design and realization of the system will be evolving and changing over time
- its operating environment will be complex and also changing - the ULS system must adapt

Continual Change

- changes made during execution
- emerging conditions
- machine and communication configurations
- users, demands, load
- upgrades, tuning, policy-based alterations and adjustments, and reconfigurations
- data / encoding as standards change

The changes must be able to be made while the system is running, not as part of a large reboot or reset. Installation cannot be part of a change. The conditions under which the system must effectively operate will be emerging over time and will rarely if ever be static or predictable.

The hardware and software configurations will be changing, the users, loads, and demands will be changing, upgrades will take place, tuning will happen, alterations and adjustments will be made under policy constraints and sometimes in an *ad hoc* manner. The system will be sufficiently long-lived that data and encoding changes must be survived as standards change out from under the system.

Despite our best efforts, the ULS system will suffer continual failures of both hardware and software as both

Continual Failure

- a fairly constant level of hardware failures
- software components will be stressed beyond their designed-for capabilities
- a fairly constant level of software failures

Continual Failure

- operating environments will be changing as failed hardware is replaced, hardware and software are upgraded (and sometimes downgraded), and configurations of components are modified
- the ULS system needs to continue operating (performing the functionality it is capable of) while repair and self-repair take place

hardware and software components are stressed beyond their designed-for capabilities. Operating environments will be changing as failed hardware is replaced (sometimes), hardware and software are upgraded and sometimes downgraded (meaning that different versions of the same components will be running at the same time), and configurations change. While repair and self-repair take place, the ULS system must continue to deliver whatever functionality it can.

A ULS system can be thought of as a system of systems, though a system of systems hints at a more coherent architecture and design than we imagined a typical ULS system would exhibit. Moreover, we didn't believe that a ULS system would be as clearly hierarchical as a system of systems implies. But, the scale of the ULS system is beyond what a human operator can deal with—while both monitoring and making adjustments—except at a very high and coarse level.

Hard to Operate

- interconnected networks of long-lived systems of systems
- no human operator(s) will be able to monitor and adjust all of it alone except at a very high and coarse level of detail
- ULS systems must (be able to) run mostly or fully autonomously

Without Boundaries

- open: interacting and interoperating with other systems and with the physical and human worlds

Therefore, a ULS system must be able to operate mostly or fully autonomously, monitoring itself and adjusting itself. In this, a ULS system begins to demonstrate some of the characteristics of a living system.

One of the most difficult-to-understand characteristics is that the system is not closed in the traditionally understood sense. It interacts and interoperates with other systems, which can be thought of as ephemeral parts of the same ULS system, and also with the physical and human worlds. A ULS system is quite like an agent in the real world, sensing, making sense, and acting; and also like an agent in the human world. The people who interact with a ULS system are actually part of it. To see how this can be, think of some part of the ULS system that is monitoring another part, adjusting it, and thereby keeping it in trim. You can imagine an exception handler or an event handler for this example. Now picture that a person is serving that role: observing and correcting or adjusting the behavior of the system, much as a dedicated system administrator does, or an operator at a nuclear

Decentralized Changes

- developed by dispersed teams with different schedules, processes, goals, and stakeholders
- revisions and changes to the components will not be strongly coördinated
  - ...may appear discontinuous or even random

power plant. In an important sense, it doesn't matter too much whether code or a person is on the other side of an API or communication protocol.

Not only do ULS systems fail to live up to Fred Brooks's ideal of being designed or architected by a single mind (in order to achieve conceptual integrity), a ULS system cannot be designed by a single team of any size. The teams that design and create a ULS system are dispersed, with different schedules, different processes, different stakeholders, and different goals. In fact, not only are they dispersed in space but in time, which means that the fundamental ideals, principles, and philosophies of the teams can be totally disjointed and incompatible. Therefore, revisions and changes will never be coordinated in any sense, and probably not even coherent; they will appear discontinuous or even random.

Inconsistent

- requirements will be uncoördinated and will be defined at disjoint times and places
- the design will likely become a bit out of kilter, even while the ULS system must continue to operate
- ULS systems must tolerate inconsistency
  - e.g. multiple versions of the same component / subsystem might be running on the ULS system at the same time

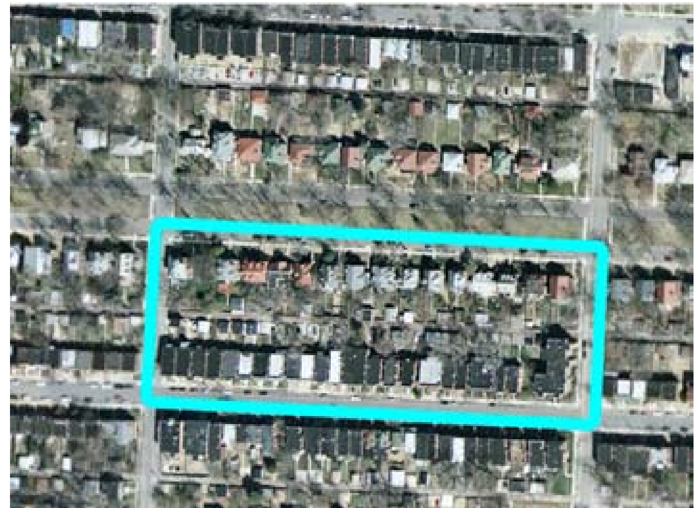
Requirements engineering is thought by many to be the secret to good and high-assurance designs. But because ULS systems are so off-scale and the design efforts so uncoordinated, the requirements—if a complete set can even be determined—will be uncoordinated, disjoint, out of kilter because the stakeholders are so disconnected. The requirements will be defined at different times, in different places, by different people, using different methods, and operating under different philosophies. With inconsistent requirements, the design will appear out of kilter, but the ULS system must tolerate its own inconsistencies.

Beyond Human Comprehension

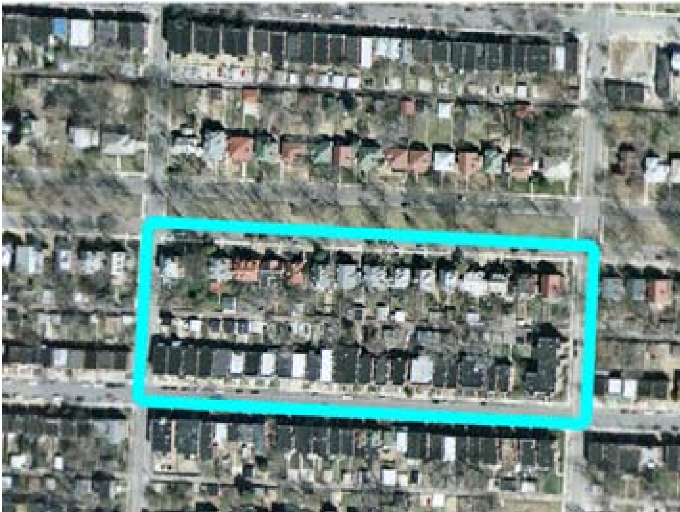
- size of the system is beyond human comprehension, perhaps even beyond automated comprehension because of dislocations and disconnections
- the sheer volume of the problem, design, and solution spaces exceed present capabilities for reliable development and evolution

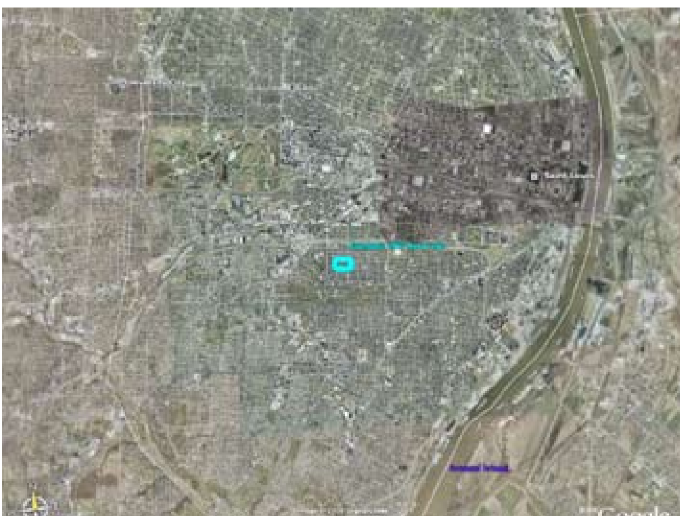
These observations have hidden within them an even more difficult fact. The size of a ULS system and its complexity make it beyond human comprehension. Its design is also beyond human comprehension. And perhaps both it and its design are beyond machine comprehension. Therefore it cannot have conceptual integrity of the sort Brooks says we need. Attempts to understand a ULS system will need to use the techniques of natural science. The sheer volume of the problem, design, and solutions spaces—if such terms even make sense—exceed present capabilities for reliable design, development, and evolution.

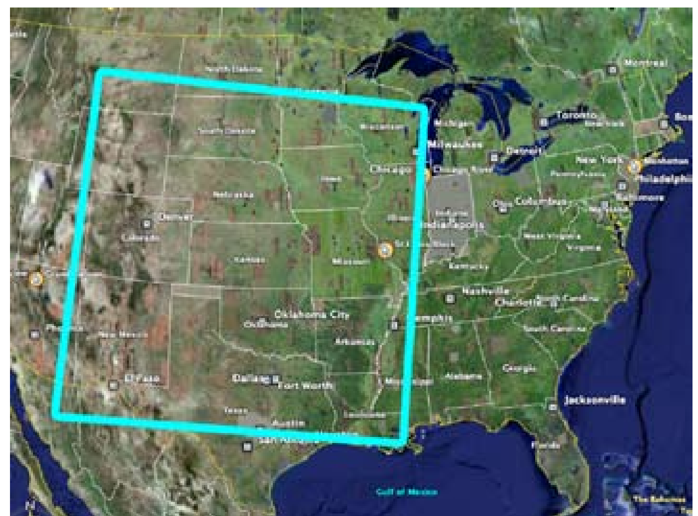
Because the most obvious and seminal characteristic of a ULS system is its size, I want to take a few minutes to give you a sense that is less abstract than number of lines of code of how large a ULS system can be.



One way to look at it is how difficult it would be to produce a system of the right size by coordinating the work of a number of individuals. Imagine that you have been given 2.5 pages of distinct Java code from every person alive on the planet today (about 6.7 billion people as of the date I gave this talk). Though this really just equates to lines of code, it links the complexity and size of the system to the complexity and difficulty of trying to integrate the chunks from so many people, and presumably the difficulty of communication among them to





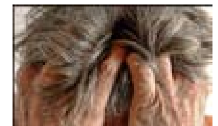


produce something coherent. This makes it easier to see that the difficulty of the ULS system is not in piling up the right (large) number of components or lines of code.

Another way is to try to map the software system to a physical analog and see how large that analog is. Let's look at the number of classes / objects that would make up a ULS system were it implemented that way. There are about 187 lines of code in the data and method definitions for a single class. There are therefore about 5.34 billion classes / objects in a large ULS system—that is, a ULS system of the size I've been trying to scare you with. The analog I'll use is a house. The first picture (above left) shows a block from a suburb of St Louis, Missouri, a large city in the American midwest. The turquoise rounded rectangle has about 32 houses or buildings within its borders. Let's take that density as the uniform density of classes / objects in a ULS system. To put that another way, let's assume that the ULS system is a tract of houses that is as dense as this region throughout, and let's want to see how large a geographic area the ULS system takes up.

The picture in the upper right is a zoom out from the first one. The turquoise border is still visible. The picture in the lower left above is a further zoom out—here the Mississippi River is visible, with the central part of St Louis shown as a darker rectangle. The final image is the lower right, which shows how large an area is required to hold the housing tract that represents the size of the ULS system. For those who have a more gut-level feel for geography in Europe, the picture at the top of the next page shows roughly the same area as the area in the US.

When reckoned with this way—this visual way—it indeed looks as if building ULS systems will be difficult almost beyond human capabilities. But in the study we also concluded that ULS systems will appear even if we do nothing—by the slow creep of size that has been going on since the beginning of programming.



The question, though, is how we get there from here in a reasonable, deliberate manner. I've heard many people state that the answer is to aggregate components, or modules. In this way, ULS systems can be achieved almost through an additive process, or at worst like a process similar to putting together a jigsaw puzzle. The argument goes that we know how to build modules or components, and making a large system—a ULS system—is just a matter of aggregating and/or integrating them. The analogy is the we know how to design and build houses and buildings, and so building a city should be easy: it's just building a lot of houses and other buildings.

The picture just below is a sample of the sorts of hous-

es and buildings we can build. The one in the lower left of that picture is a (fancy) mud hut in Africa; the one to its upper right is a house in Levittown, New York (on Long Island); the one to its upper right is the Frank W. Thomas house in Oak Park, Illinois, designed by Frank Lloyd Wright; and finally the upper right is the CSAIL Building at MIT—formally it's the Stata Center and was designed by Frank Gehry.

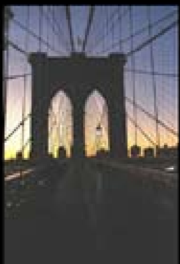
The metaphor seems to hold when we look at New York. An aerial picture reveals that Manhattan is a large, mostly ordered collection of buildings. One of the approaches the ULS study group determined was useful to explain the complexities of ULS systems is to liken them large cities, so this analogy is good. However, unlike the metaphor that says that a ULS system is simply like a city which is a collection of buildings, the metaphor as explored by the ULS study group requires us to look deeply at what a city is and how it operates in order to appreciate that a ULS system—like a city—cannot be simply an aggregation, and how scale changes this qualitatively

Let's look at New York more closely. First, notice that there is a network of streets and alleys. Some are quite broad and others narrow and inconvenient for most or any traffic, but aid with deliveries and garbage collection. Streets enable people and goods to move about and to and from the city.

There is a system of subways—some parts are actually underground but some run overhead. These are in place for two reasons: Manhattan is so crowded that traveling by car is not practical, and cars and taxis are too expensive for many, particularly when commuting. Further, there is a shortage of parking.

Another mass transit system uses the streets: buses. There are a variety of tradeoffs regarding when and how to use these three mechanisms for transport, but many cities use all three and sometimes others.

There are systems of power, communications, and sewers in New York. Power and communications are sometimes above ground; sewers are thankfully below ground. Without these three things, New York would not be considered to be out of the 19th century. Additionally, water is transported to the city and distributed to people and companies.



There are bridges, docks, and in general the Port of New York. Bridges enable motor traffic to and from other parts of the country; docks enable the import and export of goods and travel to and from New York—both to the rest of the country and overseas. The Port is what made and makes New York one of the great cities of the world.

Because people live in the city, there is an economy that makes the city work. In fact much of the design of the city is driven by economic factors: where companies and stores are located, where housing is built, zoning laws to roughly organize the expansion and evolution of

the city are in many cases determined by economic factors. A large fraction of the activity of the city is dedicated to feeding its inhabitants.

There is a culture to the city, determined by its people and the cultures of their ancestors, but also there is a culture that is the product of the attraction the city has for artists and craftspeople. When someone near New York wants to see a play, hear the symphony or opera, look at paintings and sculpture, that person heads to the City.

Because there are many immigrants to New York, it is multi-lingual and multi-cultural. This means that sometimes you need to use a simplified English or a sort of pigdin or even an ad hoc sign language to communicate, or engage a translator on the spot.

New York is a combination of the old and new: old and new buildings, old and young people, old and new technology, etc. It was not made at once and uniformly.

The city exhibits a variety of scale—from immense and immensely tall buildings down to cardboard boxes under bridge overpasses. Scale of size, scale of cost, complexity, convenience, etc.

Because New York is a large and complex human place, it needs sets of services like police (law enforcement), fire departments, and garbage collection. These maintain order, safety, and sanitation. There must be a system

of governance to maintain order. Laws, regulations, zoning keep the city running well and evolving in a reasonable manner.

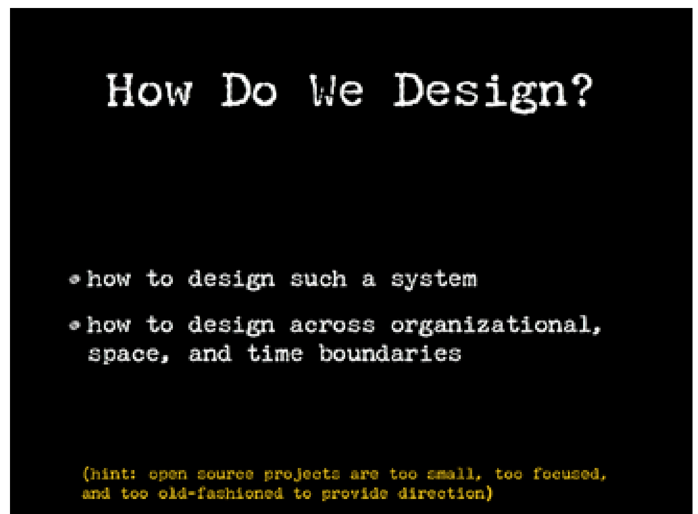
The city is constantly being repaired and built. Old structures are modified or torn down; new ones are built, damage is repaired, streets, power lines, communications, the sewer system, transportation, etc are all repaired as and when needed. Cities continue to grow until their usefulness requires them not to.

And when repairs aren't undertaken, the city falls into decay then rot and finally ruin and disappearance.

When viewed this way, a ULS system seems unlike—very unlike—what most software engineers and computer scientists consider a proper system or program. In an email to a mailing list I'm on, Eric Raymond said, "The mathematician in me craves [the] ideal world of verifiably correct components assembled into provably correct systems." He went on to say that all the hard benefits in such a statement hide behind the fact that people—who make mistakes—need to produce all the designs, the code, and the proofs. That is, all the ideal benefits are anchored to the fallible human. This seems to be a weak retreat in the face of the overwhelming differences between the image of the clarity of mathematics and the messiness of the ULS system as analog to New York.

The difficult question ULS systems beg is how we will be able to design such systems—so large, so complex, so diverse in components, scale, and age, and so widely dispersed in its builders and deployment. Design of such systems don't take place in one place, at one time, and by one organization, so we need to be able to design under those conditions.

Open source might seem to be a model for how to proceed, but open-source projects are actually very small scale, too focused in functionality and scope, and too old-fashioned in terms of the process and technology that is used. What open source can teach us, perhaps, is how to design in the face of geographically dispersed develop-



Sufficient information for a craftsman of typical skill to be able to build the artifact without excessive creativity.

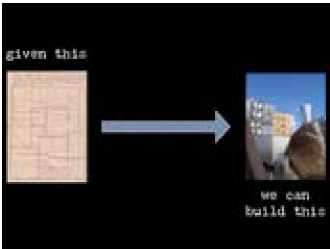
–Guy L. Steele Jr., over dinner at Cascal Restaurant, Mountain View, California / Valentine's Day, 2006

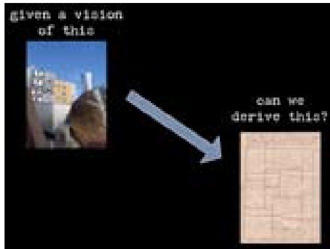
ers—but even here almost all open-source software is designed by a single person or a small group, and only the fringes of the system are available for extension by volunteers.

Returning for a moment to how various people define design, Guy Steele said on one occasion that design is

sufficient information for a craftsman of typical skill to be able the build the artifact without excessive creativity.

This definition echoes Carliss Baldwin's a bit, and mentions two aspects of the building part of design & build that might strike some as not obvious: that building requires craft and creativity. For many, the designer has done all the creative work, and for software the craft perhaps means the ability to write clean and stylish code rather than being a skilled software designer.





MetaDesign Space

- 3 different broad approaches to design
- under which conditions can each approach to design work?
  - — based on how much can be known and at what times

Now I'll turn my attention to some work by Adrian Thompson to see how we can approach designing ULS systems. He defines three of what I call *MetaDesign Spaces*. A metadesign space is an approach to how to design based on what is known at what points in the design process. Pictorially, I depict a design as a blueprint, and from the blueprint, a builder can build the design, much as Steele says, with some craft and creativity. The design process, then, is this: given a vision of what is needed to be designed (and the vision can be as complete or incomplete as is necessary) how do you go about deriving the blueprint?

In the first of the three spaces—*MetaDesign Space 1*—it is assumed that it is possible, given the vision, to work all the way backward to the design. That is, it's as if you considered the mapping from a blueprint to the built thing a function, and that function were invertible. It might be that the inverse produces a "blueprint" that doesn't always exactly produce the same building—the craftsman using understandable but not excessive creativity might build something slightly off, but not off so much that the result is not recognizable as the right building, but that doesn't matter too much to the basic idea.

MetaDesign 1

Inverse model is tractable: If there is a tractable 'inverse model' of the system, then there is a way of working out in advance a sequence of variations that brings about a desired set of objective values.

–Adrian Thompson, "Notes on Design Through Artificial Evolution: Opportunities and Algorithms"

(waterfall???/ spiral???)

If we were to approach designing a ULS system assuming that the inverse model could work, what would we have to do in order to handle the scale-related issues that make ULS systems not like systems today. This ap-

Design as Abstraction

Design refers to thinking that is up a level, up in the abstract, in the realm of ideas, up toward the Platonic ideal of our intent.

–Steve Metsker, email

proach means coming up with the entire design before building any of it—the classic, or should I say caricature, waterfall method (without any feedback loops). The key hint comes from Steve Metsker who says that design takes place "up a level" in the abstract, in the "realm of ideas." Because ULS systems represent an extreme, design-up-front requires our abstractions to be giant sized and super powerful, and, more importantly, the machine of abstraction itself needs to be ever more powerful. We need better programming languages—languages much further away from both the machines on which they run and also from the mathematical foundations on which many of the programming concepts we use are based. We need, also, a better understanding of the concepts we do use—such as what an error is and how to handle it.

We also need to learn new designs, design frameworks, and design approaches from naturally occurring, ultra large scale systems (e.g., biology, ecology).

Current mathematical models for software treat it as a closed system, divorced from other software, development tools, and, most importantly, the real world.

We need to be able abstract over things we haven't abstracted over before: control, time and history, events, relationships (and their harmony or disharmony), and errors. Abstractions that are intended for use as models for the real world need to be tied to semantics that relates to the real world in such a way that other abstrac-

ULS Design using MetaDesign 1

- metadesign 1: inverse model is tractable
- raise abstraction level, not simply larger aggregations
- better languages
- better understanding of concepts
- learn from naturally occurring, very large systems (e.g., biology)

Reëxamine Foundations

- current foundational models treat software as abstract, isolated, closed-world, mathematical programs
- abstraction, modularity, correctness

Abstraction

- raise level of abstraction
  - — control
  - — events / state of computation
  - — trends
  - — transitions
  - — relationships
  - — errors
  - — time / history

Abstraction

- tie software abstractions to
  - — the real world
  - — (semantic) models
  - — other computational artifacts

Modularity

- spatial compartmentalization versus common origin of functionality
- single or primary function versus multiple functions
- information hiding versus transparency
- tight cohesion versus diffusion
- hierarchy versus semilattice

tions that operate on the things already modeled have a chance of making sense when put together. And because software artifacts need to exist in a computational world, these artifacts have to be able to participate along with other such artifacts in the making and building that goes on there.

One of the concepts we're familiar with as computer scientists is *modularity*. As we've come to know it, modularity is almost entirely about separation or spatial compartmentalization—originally (as best as I can tell) from the conceptualization of modularity by David Lorge Parnas who was looking for how to break up work. What some people later called a *module* (Parnas didn't seem to) was a work task that could be separated for a person or group to work on with minimal communication with those working on other modules. In order to do this, the minimal information to be shared became what we now call an *interface* or *API*. All the other information (such as the implementation) was hidden—not because of

Modularity: Liver

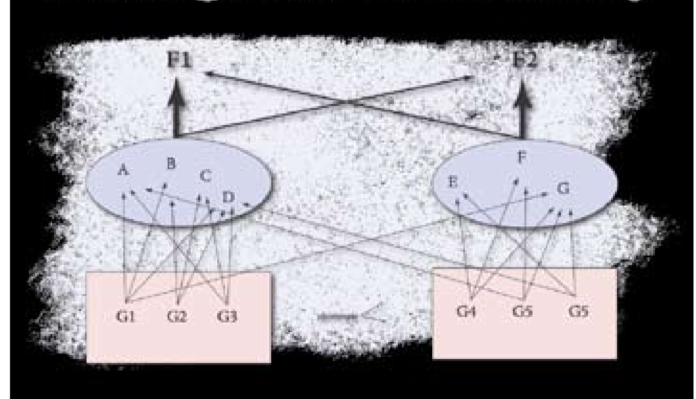
- stores and mobilizes energy
  - controls blood sugar (glucose) (regulates glycogen & fat storage)
- aids digestion (produces bile)
- regulates blood clotting
- manufactures clotting factors & other blood proteins (during gestation, forms blood)
- produces several (non-reproductive) hormones
- manufactures cholesterol

Liver

- filters blood (eliminates bacteria, detoxifies external and internal poisons, breaks down drugs)
- produces vitamins (e.g. vitamin D)
- stores minerals (e.g. iron)
- produces essential immune system factors
- monitors & manufactures numerous blood proteins

some mechanism of the programming language, but because by hiding a lot of information, the people working on one module didn't have to talk to the other teams much. Later this was turned into language features.

But other disciplines use the concept of modularity—biology, for instance. Biologists think of a module as representing a common origin more so than what we think of as modularity, which they call *spatial compartmentalization* (it's from biology that I took this term). There are a couple of examples. One is the liver, which we might think of as a module because it is something that can be physically separated from a living being. But a liver does lots and lots of things, so it doesn't really conform to our naïve idea of what a module should do—one or a few related things, and in Parnas's vision, those things usually depend on a particular representation choice.

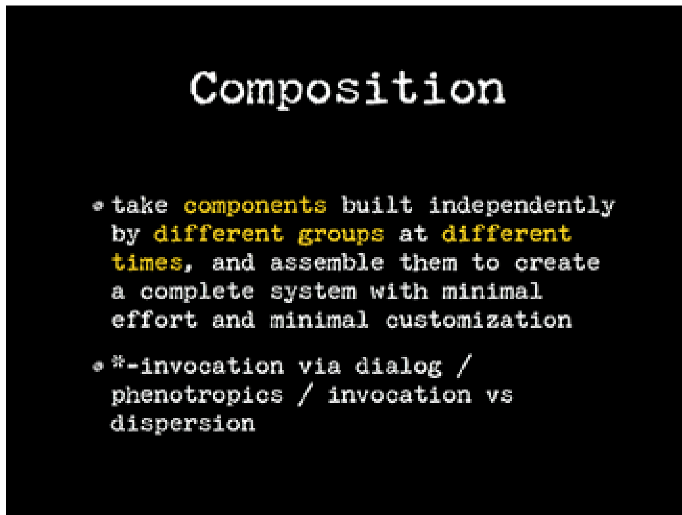


The common-origin definition from biology is related to the Parnasian approach in that a module is something that is produced by a single person or a single team. A group of genes is called a module if that group (by producing proteins at the right time etc) mostly builds one recognizable thing (an organ, for example) that mostly

does one thing—but it's ok to contribute to building some other things, and it's ok for the main thing built to do some other things as well.

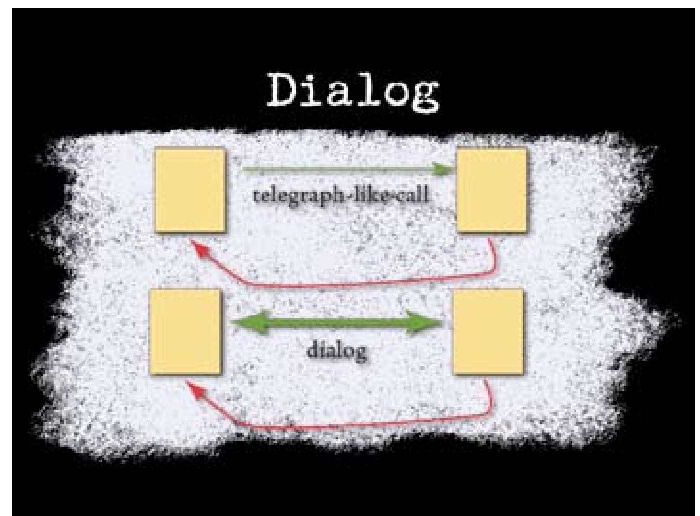
Another apparent module (in the computer science sense) is a cell within an organism that has an immune system. A cell has an outer membrane that separates it from its environment (which is within the body of the organism). This reminds us of the concept of *encapsulation*. However, there is no absolute information hiding going on here. For the immune system to work, other cells "communicate" with the innards of a cell using proteins that protrude through the membrane from the inside of the cell. These proteins contain identity information (*is this cell part of self?*) as well as other indicators of healthy operation. When a cell is not healthy, an outside cell that's part of the immune system can command the cell to destroy itself without spreading toxins.

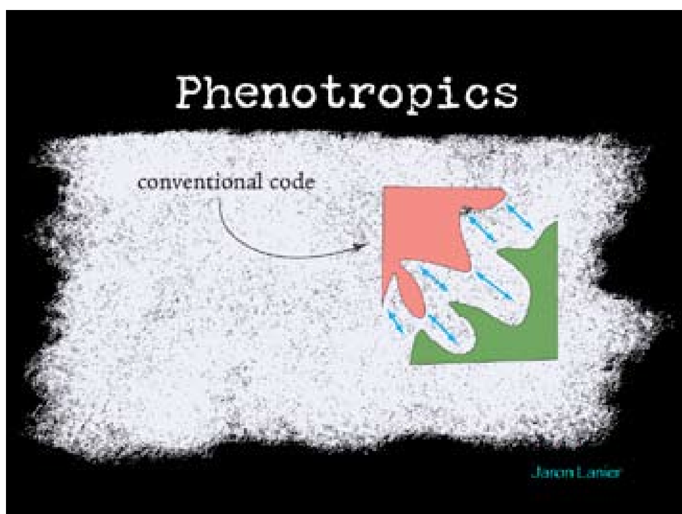
This goes against our understanding of how modularity should work—and contradicts our view as computer scientists that encapsulation means information hiding. This is what I mean when I say we need to rethink our concepts in view of ultra large scale systems. Biological systems are very much larger than anything (coherent) that people have built. If we are to design with a blank sheet (so to speak) before building (metadesign space 1), we should look at these other, naturally occurring systems and learn how our concepts are like and not like the ones that operate there and adjust accordingly.



We need to learn how to compose components designed and built independently—by different groups, at different times. We need to look at other kinds of ways of getting things to happen in the system—other mechanisms than procedure/function/method invocation.

One idea is for components to engage in dialogs (or do the equivalent) rather than just sending commands (or the equivalent). Today our systems act like a telegraph system, complete with a code for how commands (and arguments) are passed; these commands, the code, and how arguments are passed are generally very inflexible, and this is because by being that way, compilers and other tools can find particular sorts of errors and the system can run fast. What ULS systems teach us is that this inflexibility hinders our ability to design and build the systems in the first place.





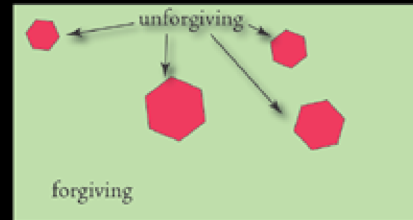
Another idea is *phenotropics*, a concept devised by Jaron Lanier. Roughly "phenotropics" means surfaces relating to each other. Using phenotropics, two entities would interact by literally observing (through pattern recognition) the surfaces of each other, thereby causing "control" and activity.

Another approach is for components to give off "smells" that indicate their needs, desires, states, etc, to which other components can react. Such an approach would enable systems to do something when normal, or-

New Views on Correctness

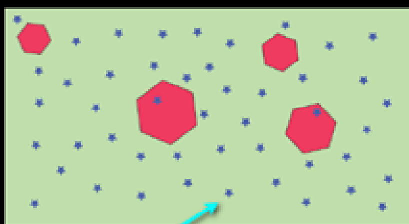
- perfection versus acceptability
- unforgiving sections require perfection
- forgiving sections require acceptability

Forgiving / Unforgiving



Martin Rinard

Forgiving / Unforgiving



bugs

Correctness

- ignoring errors can* provide more (but degraded) functionality to users
- assertions and exception handling can* decrease usefulness (by halting instead of continuing with acceptable errors)

*sometimes

dered communication fails. One could imagine one component happening upon another and thinking (or saying), “you look ill; can I help you somehow? How about a blanket? How about being reset? Can I garbage collect your internal stuff for you?”

Likewise we need to think about correctness differently. In the minds of most computer scientists and programmers, a bug is a real problem, and each one must be fixed. Martin Rinard questions that assumption, and argues that systems with bugs can still be valuable and useful. And that sometimes the act of trying to correct a flaw can cause more problems than are eliminated—by incorrect assertions and flawed exception handling. An example of this is a video game. Suppose you are playing a first-person shooter. The graphics are probably based on triangles to represent texture etc, and sometimes some triangles are degenerate (actually display as lines). When you do trigonometry on such triangles, you might divide by 0—a real horror. What to do? Instead of worrying about it, if this divide always returned 7, for example, there would be no real problem—because perhaps all that would happen is that one pixel is the wrong shade of green for one frame. Who would notice; who would care?

Rinard says that there are two types of code: *forgiving* and *unforgiving*. Forgiving code provides useful, adequate functionality even when there are some bugs, while unforgiving code is code that must be correct in order for the software to produce usable functionality at all. There is probably, he says, a lot more forgiving code than unforgiving in a system. And if bugs are uniformly distributed in the code base (which is usually false, but that might not matter if the sizes of the two regions are incomparable), then there are not too many bugs that *must* be fixed. He and some of his students are looking at real systems and proposing simple mechanisms that make the code more reliable and still work pretty well by ignoring or covering over errors.

But: More is Different

The [belief that science at one level can be easily constructed from the fundamental laws of the science at the next level down] breaks down when confronted with the twin difficulties of scale and complexity. The behavior of large and complex aggregates of elementary particles ...is not to be understood in terms of a simple extrapolation of the properties of a few particles. Instead, at each level of complexity entirely new properties appear, and the understanding of the new behaviors requires research....At each stage, entirely new laws, concepts, and generalizations are necessary, requiring inspiration and creativity....

—P. W. Anderson, "More Is Different," Science, Volume 177, Number 4047, pp 393–396, August 4, 1972.

For many years the progress in physics was such that it was believed that once the fundamental laws and principles at the lowest levels of the physical world were understood, so would be everything above it. Once we knew about subatomic particles, we would know how they combined to make up atomic particles. Then we'd know how those made up atoms, and how those made up molecules, etc, all the way up to psychology, sociology, and all of the rest. That is, there was a strong belief that once the lowest levels of the "physics stack" were understood, we'd be able to simply apply various aggregation laws to understand all the other physical phenomena above.

But in 1972, P. W. Anderson wrote a paper that many physicists thought was baloney, and the paper said that every level above required its own science, as it were, because the scale and complexity of those upper layers meant they had their own distinct laws, not predictable easily or at all from properties of the lower levels. This insight happened, probably, in conjunction with other insights that later became the basis for complexity science. I believe the same principle holds for software at ultra large scale: understanding the lower layers—no matter to what degree of faith—doesn't help us understand the whole system better or even at all. This means that the metadesign space 1 approach won't work completely.

This leads to *MetaDesign Space 2*; it is based on the idea that if we cannot fully invert the blueprint to building mapping, maybe we can invert smaller steps of the process. That is, we can probably predict well enough what a simple design / building step will do given an already partially designed and built system.

MetaDesign 2

Inverse model is not tractable, but forward model is: In this case, we can predict the influence of variations upon the objective values, but the system is not tractably invertible so we cannot derive in advance a sequence of variations to bring about a desired set of objective values. This implies an iterative approach, where variations carefully selected according to the forward model are applied in sequence. This kind of iterative design-and-test is a common component of traditional approaches.

—Adrian Thompson, "Notes on Design Through Artificial Evolution: Opportunities and Algorithms"

(iterative / agile??? / Alexander's Fundamental Process)

This is an iterative approach; it is similar to any of the agile methodologies and Christopher Alexander's Fundamental Process. The agile methodologies, by the way, were not originally discovered by programmers in the 1990s. The principles and practices have been in place in every design and building discipline for millennia. The agile folks mostly provided some good names and descriptions for what was already known in the programming world but without those names and descriptions. These served to help break us loose from the grip that programming methodologies based on metadesign space 1 (design before building) had on us.

Alexander's design and building process is essentially to design a little and build a little. This is followed by a learning and evaluation step, followed by more design and building.

Let's look at another definition of design—this one by Jim Coplien. It fits in pretty well with metadesign space 2 in that it talks about a series of steps each of which reduces the distance (though he doesn't use the concept of distance) between the current (problem) state and the desired (solution) state. This definition views de-

Design as
Problem Solving?

Design is a process of intentional acts to the end of moving from some conception of a problem to the reduction of that problem.

A problem is the difference between a current state and desired state.

—Jim Coplien, email

sign as a problem-solving activity, which probably many computer scientists and programmers would agree with or at least feel at home with.

How would we design and implement a ULS system using the metadesign space 2 approach? It turns out this approach is the most likely to succeed in the near term. Essentially, design and implementation would be done as a complex adaptive process in which individual organizations and designers operate under a set of local rules mediated by shared artifacts. This is how Alexander's pattern language works and his Fundamental Process. What is already designed and built is sitting in front of us. We can learn from that as Coplien indicates. A pattern language (or any other set of design rules as Carliss Baldwin calls them) gives us hints about what to do next. We do that and repeat until we're done.

ULS Design Process

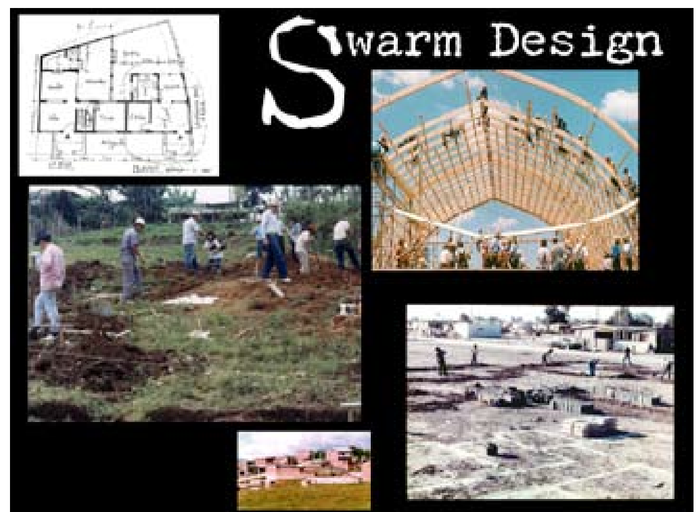
- metadesign 1 (inverse model) cannot work
- the overall design activity will operate as a complex adaptive system – it cannot be centralized
- individual organizations and designers will operate using a set of local rules mediated by shared artifacts
  - design decisions/rules, standards, pattern languages

This is the first place where we see what is meant by "design beyond human abilities." By using a complex adaptive system approach, it's possible to think about how to design and build things that no individual or individual group could. As you think about this, keep in mind P. W. Anderson's quote: the scale and complexity of the thing designed and the design organization make it impossible to predict what individual acts of design and implementation accomplish at that next higher level.

This is essentially swarm design. The original name of the process that Alexander uses was *piecemeal growth*. Another aspect to it is machine design, in which computers or some other mechanical process assists people. Swarms of termites are able to construct elegant nests by using local rules and signals from the physical world (called, in biology, *stigmergy*). New York, for example, was built using swarm design and piecemeal growth. Natural features (the rivers, existing animal paths) laid down some constraints; a few people built and newcomers adapted to what was already there; organizations came in when trade began (aided by the rivers and port); streets sprang up on the existing paths, and people's proclivity to building in rectilinear fashion made streets more regular (as did the tendency to build towns to look like the ones already seen); later zoning and other

Design Beyond
Human Abilities

- swarm design / piecemeal growth
- machine design



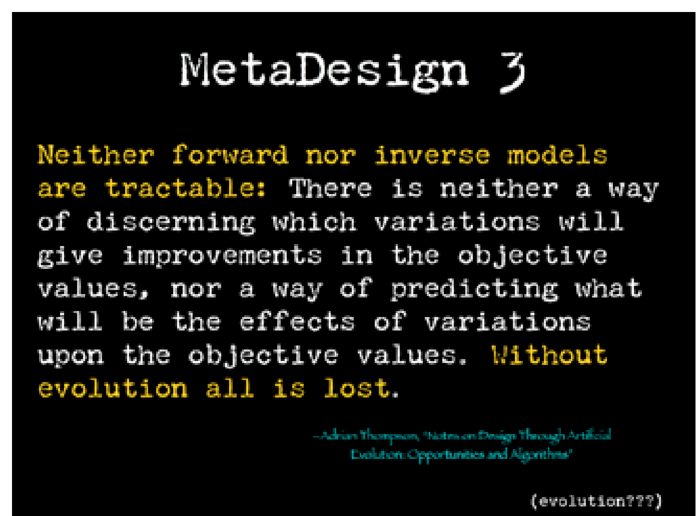
building codes laid down further constraints. New York was built over hundreds of years this way. No one really foresaw what New York would be just as we cannot see what it will become. Part of this has to do with who comes to live there—people bring their culture and thereby change the existing culture. Will New York be a financial center as it has been for 150 years or more already? Will be more of a cultural center like Boston has been? Or an educational center (again as Boston has been)? Will it be progressive like San Francisco or conservative like Atlanta? Depends on who moves there which depends on economic and other winds—sometimes literally the winds of climate: what if it becomes a lot colder, or warmer?

Now we get to the interesting part. How do we design something that we can envision but which we have no real clue about how to achieve? We cannot design from scratch because the jump is so big, and even being able to predict what small changes would achieve can't work. The leap is too large and the territory too unfamiliar.

This is really the realm of pure creativity—though it's possible to be creative with mere bushwhacking and a sense of what is good. The only alternative, according to Adrian Thompson, is *MetaDesign Space 3*: evolution.

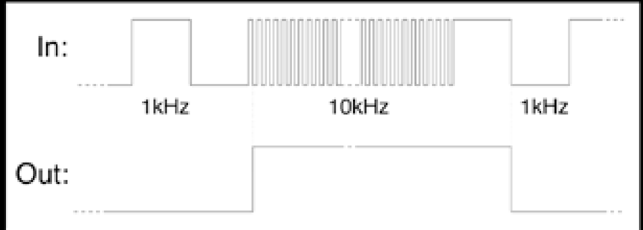
He means evolution in the senses of both natural/biological and digital evolution. Evolution is really just a randomized search for how to achieve a predicate-defined goal, along with a strong component of combining good parts of existing solutions.

One of the doubts about this approach is how practical it could be. We know (most of us, anyway) that biological systems have achieved their scale and complexity through evolution, but the scale of the natural organic world in both size and time is beyond, in both cases, what we need for ULS systems, and we certainly don't have



MetaDesign

- MetaDesign 1 (of 3): whole design before building (inverse model)
- MetaDesign 2 (of 3): stepwise design; design / construction intertwined (forward model)
- MetaDesign 3 (of 3): evolution as design (evolution)

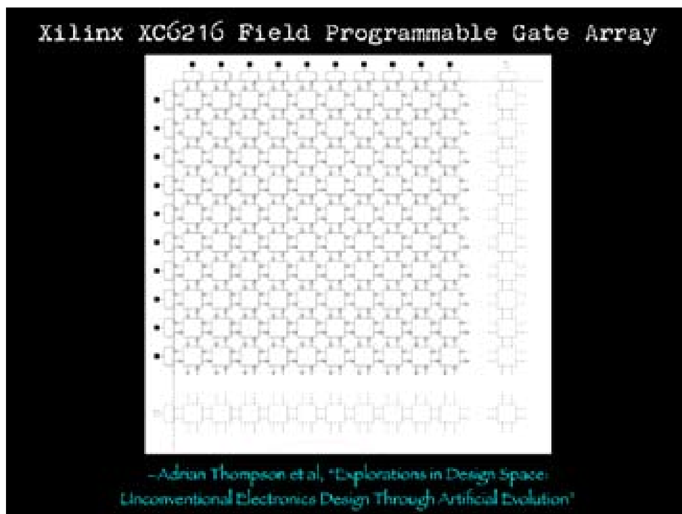


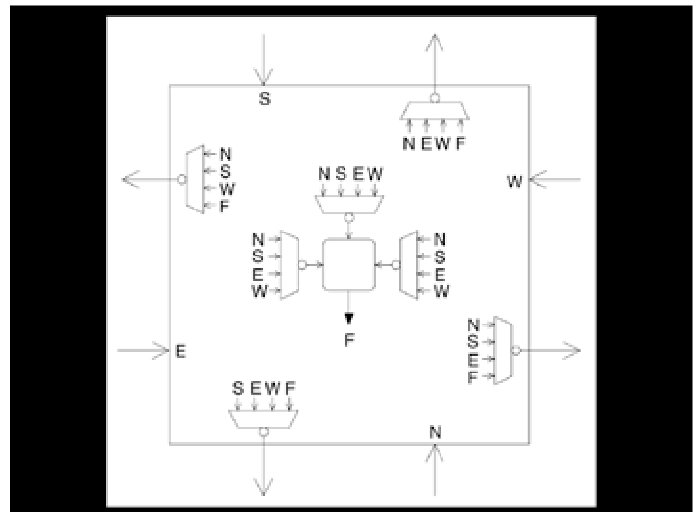
Create a circuit that can distinguish a 1kHz squarewave from a 10kHz squarewave

no clock

millions or billions of years to design and implement ULS systems. The question is what can be designed and built quickly.

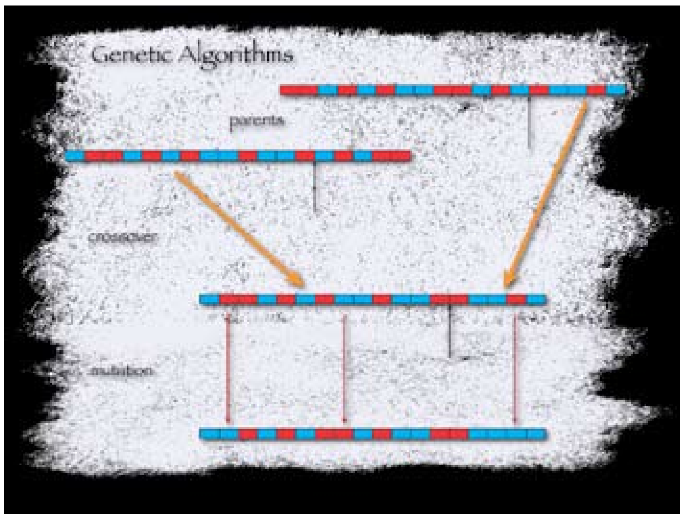
To show that evolution is not a ridiculous idea, I want to look at an experiment Adrian Thompson's group did. The idea was to design a circuit that could distinguish between a 1kHz and 10kHz square wave. The circuit would have a series of bursts of 1kHz and 10kHz signals intermixed and return a 0 whenever the 1kHz signal was present and a 1 when the 10kHz signal was. The raw material is a 10x10 section of a field programmable gate array (FPGA). The FPGA part (the actual chip) that was chosen has a 64x64 grid, but only the upper lefthand corner (10x10) was used. Each cell of the array looks like the figure below and to the right. There are 4 inputs—





North, South, East, and West—and 4 outputs. Each output is selected by a multiplexer that, for each direction, chooses from the outputs of other 3 directions along with the output of the function unit. The function unit can compute any boolean function of 3 arguments, and each of those 3 arguments is selected, by a multiplexer, from the 4 inputs to the cell. An FPGA is called "field programmable" because the choices made by the multiplexers and the function computed by the function unit can be specified/supplied after the part has been built and installed—out in the field. Eighteen (18) bits is required to specify all the multiplexers and boolean function, and so 1800 are required for the entire 10x10 FPGA that was used in the experiment.

A genetic algorithm was used to determine what those 1800 bits should be. In the genetic algorithm used, a population of vectors of bits is used to generate a set of offspring which are tested for fitness. A child is created by selecting at random 2 parents. Then a cut point is chosen at random. The child's bits are determined by taking the bits to the left of the cut point from one parent and putting them in the corresponding positions in the child, and the remainder are taken from bits to the right of the cut point from the other parent. Then the child under-



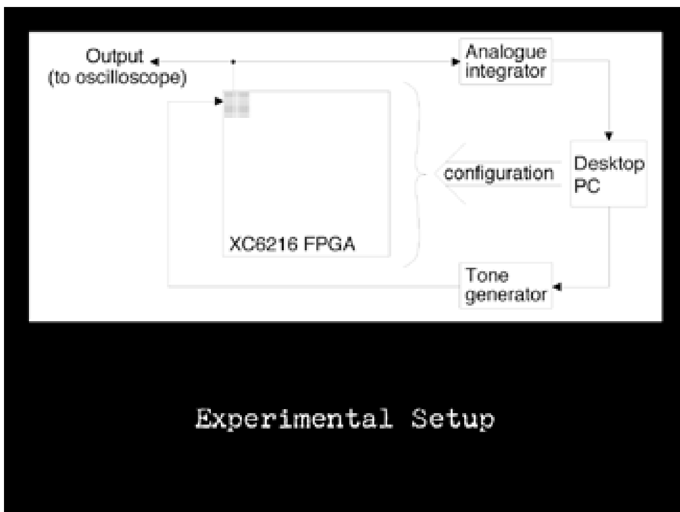
Genetic Algorithm Details

- population: 50
- genotype: 1800 bits
- crossover probability: .7
- bitwise mutation probability: 2.7 bits changed expected
- elitism
- 5000 generations (no improvement after generation 4100)

goes a set of mutations which changes some of the bits in its vector. In this experiment; the population was 50 vectors, and genotype was 1800 bits; the likelihood a vector in the population will be chosen to breed was 70%; the number of bits expected to be mutated in each child was 2.7; the best 50 of the combined original and child population were chosen as the new generation (elitism).

After 5000 generations, a good solution was found.

The experimental setup was this: An actual FPGA was used. It was hooked up to a computer that ran the genetic algorithm. To test fitness of a vector in the population, the computer loaded the configuration from that vector into the FPGA. Then a series of test signals were sent through the FPGA, and the integrated results were used as the fitness. This means that the integrator measured how long the output signal was high or low over the testing period for each configuration, resulting in a number, and this as compared to the expected number to yield a fitness result.



Here is what was evolved. The circuit needs to see the first entire high waveform of the 1kHz or 10kHz sequence to get the answer, and the answer is asserted (the output signal goes high or to zero, as the case may be) within 200ns (nanoseconds) of the end of the falling edge of that first high waveform. The delay though each cell in the FPGA is 5 orders of magnitude shorter than the duration of the shorter of the 2 possible waveforms. That is, the circuit cannot be directly measuring the duration of the waveform because there aren't enough elements (by a factor of 1000) to do the measurement. And although the FPGA part used typically uses a clock (to enforce digital logic), the circuit was not supplied with one. This means that though FPGAs are thought of as digital devices, the evolved circuit operates as an analog circuit.

What is Evolved

- needs just 1 high waveform to get the answer
- sets proper answer within 200 ns of the end of the falling edge of waveform
- delay through each component is 5 orders of magnitude shorter than the input waveform
- no clock or outside timing circuitry

In one of the papers on this work, Adrian Thompson and his co-authors state that the resulting circuit is "... probably the most bizarre, mysterious, and unconven-

"...probably the most bizarre, mysterious, and unconventional unconstrained evolved circuit yet reported."

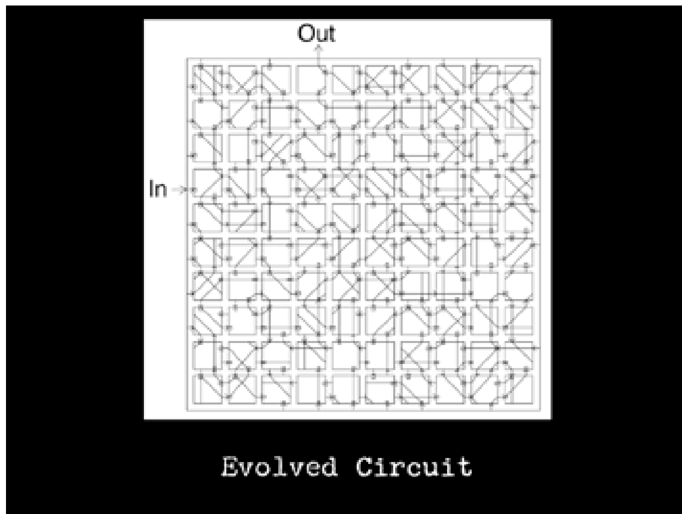
–Adrian Thompson et al

tional unconstrained evolved circuit yet reported." As we'll see, it is bizarre beyond anyone's wildest hopes for strangeness.

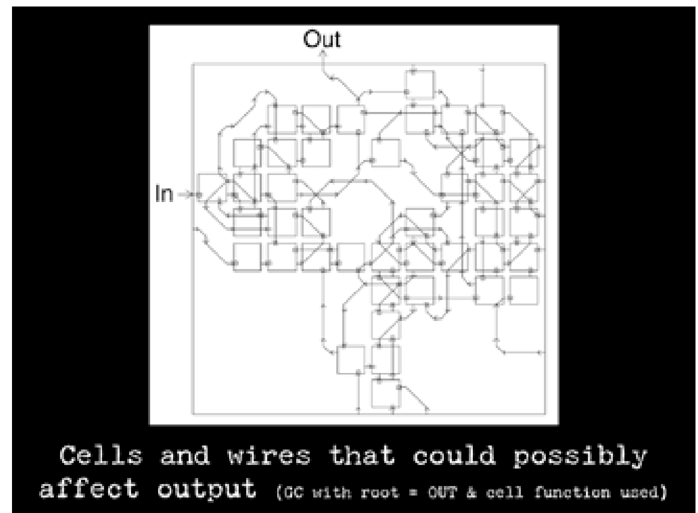
Looking at the connections in the 10x10 portion of the FPGA we see some oddball things like loops and cycles. These diagrams don't show the functions computed by the function unit. The obvious question is how the circuit works.

There are several ways to try to figure it out. One is to start at the output and trace backward to the input, marking cells whose function units are used. This is like a garbage collection of the cells considering the output cell as the single root. This doesn't help much, it turns out.

Another way is with a technique called "clamping." One by one, each cell's function unit is set to return a constant 0 and the entire FPGA is tested to see whether it still works as it did before the clamping. When we do that, we find a more manageable circuit—the diagram with the red cells below shows it. This diagram begins to show what is peculiar about this circuit. The function units

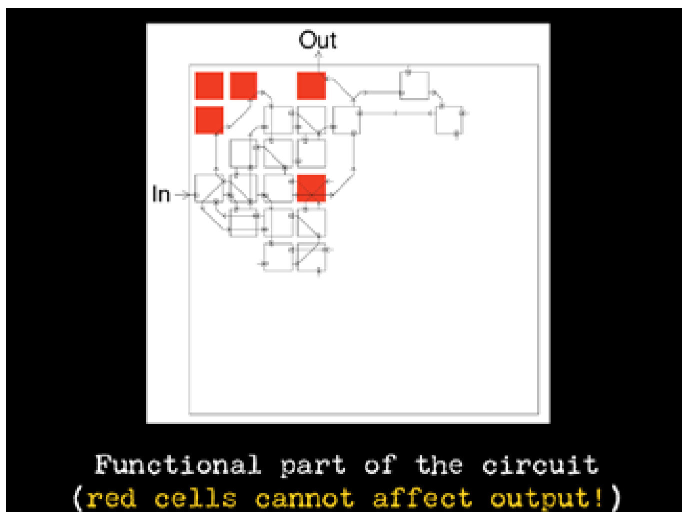


Evolved Circuit



Cells and wires that could possibly affect output (GC with root = OUT & cell function used)

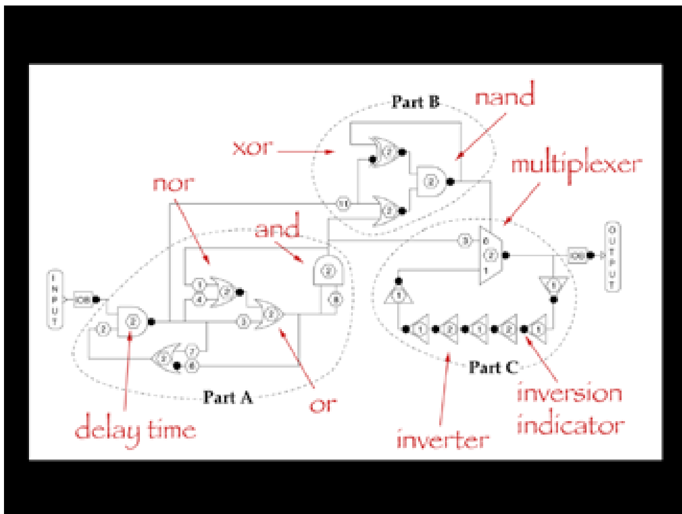
of the red cells are not used by the circuit, but nevertheless, when those units are clamped, the circuit doesn't perform. Ignore the upper lefthand cell in the diagram: The remaining red cells are used only as routers—that is, they are essentially wires in the circuit (logically).



Functional part of the circuit (red cells cannot affect output!)

That upper lefthand red cell, though, is particularly interesting: It is not directly connected to the active part of the circuit at all. It doesn't appear in the diagram that shows all the cells that could be involved in the functioning of the circuit. Yet if its output from the function unit is clamped, the overall circuit doesn't function.

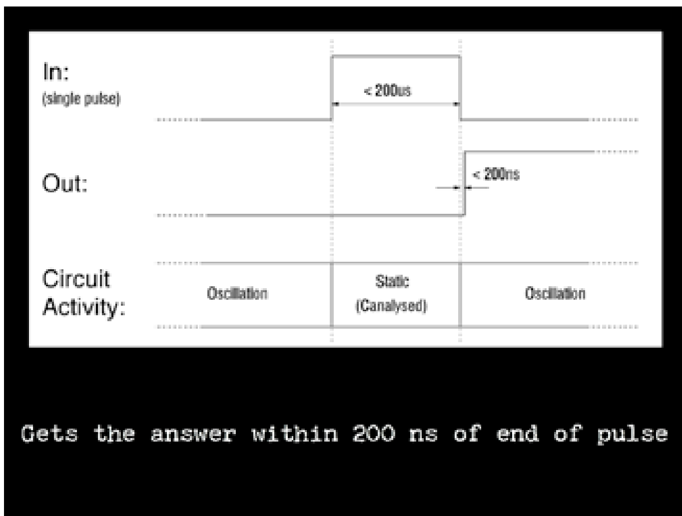
Another way to understand the circuit is to look at the circuit diagram for it. That's on the next page at the top left. The circuit has 3 parts, A, B, and C. Each involves a feedback loop. Part C is odd because its feedback loop is just an even number of inverters, which amounts to a delay.



How it works is a mystery

- degraded performance in another FPGA
- CMOS discrete circuit does nothing
- logic circuit simulation does nothing
- does not work when the red (unused) cells are clamped (outputs fixed)

Here are some strange things about the circuit. Recall that the training / evolution process used an actual FPGA; when a different physical part is substituted for the one trained on, the circuit doesn't work as well—the evolution was particular to the physical part that was used. If you make the circuit in the diagram with CMOS, it doesn't work. If you simulate the circuit with one of the commercial simulators, it does nothing.



The most intriguing behavior is how the circuit gets the answer. As noted earlier, the right answer (did it just see a 1kHz or a 10kHz waveform) is gotten 200ns (nanoseconds) after the trailing edge of the first high waveform, which means that whatever interesting that's happening is happening when that waveform is high. But, as far as anyone can tell, absolutely nothing is happening in the circuit at that time.

When the input goes high, the feedback loops in parts A and B are cut, and therefore those parts of the circuit revert to digital logic and are static. Part C is observed to be static. Observing the circuit when the input is high reveals that the power levels are quiescent, there is no activity in the feedback loops, there are no short pulses or glitches anywhere, there is no thermal activity, there is no rf activity nor response to rf—putting the whole affair in a Faraday cage doesn't change its behavior. In short, no observation shows anything happening at all. To be fair, some of the observations are intrusive enough that they could be masking the phenomenon they are trying to detect.

When input is 1 (high):

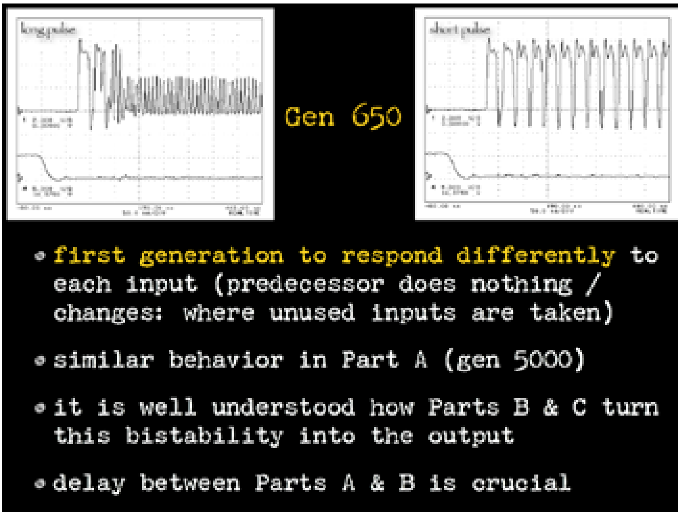
- feedback loops in Parts A & B are cut
- therefore, Parts A & B revert to digital logic, and become static
- Part C is observed to be static
- notice Part C is very weird

When input is 1 (high):

- power levels are quiescent
- no activity observed in the feedback loops
- no short duration pulses (glitches)

even though

- power supply and wires are shielded / no decoupling capacitors
- entire circuit in a Faraday cage



The final way to understand the circuit is to look at its evolutionary history. Generation 650 is the first one to exhibit any difference in behavior between the long pulses and short ones. When a long pulse is high, there is no activity in part A through the feedback loop. Right after the pulse goes low, the feedback loop starts to ring quickly. After a short pulse, the loop rings more slowly. Generally, this behavior is retained until generation 5000, and the rest of the evolution produces parts B and C to turn this bistability into the correct output—and how parts B and C achieve this is well understood. What's not understood what causes this bistability in part A.

"...most bizarre, mysterious, and unconventional..."

• it is unknown how the bistable behavior arises in the final circuit

(circuit activity, metastability, and thermal effects have been ruled out)

Let me make sure this point is clear. No one understands how the circuit fundamentally works because as far as any observation can reveal, during the crucial period, absolutely nothing is going on at the physical level. Most likely, whatever is happening is not possible to detect with ordinary techniques.

When the experiment is changed so that several parts are used simultaneously at different temperatures (one in an oven, one in a refrigerator, and one at room temperature—this is slightly exaggerated), a circuit is evolved that works well and is understandable. Similarly when a logic simulator is used instead of the real part.

Evolutionary Design

• no distinction between design and implementation
• can explore regions of design space beyond conventional methods
• not constrained by understandability, modularity, or beauty / elegance
• intermediate best results* can be absurd

*e.g. the tone-discrimination circuit

Evolutionary Design

"...more precisely formulated questions regarding the organization of behaviour must replace fuzzy notions of 'understanding' and 'explanation' rooted in functional* decomposition."

–Adrian Thompson et al

*and other kinds of

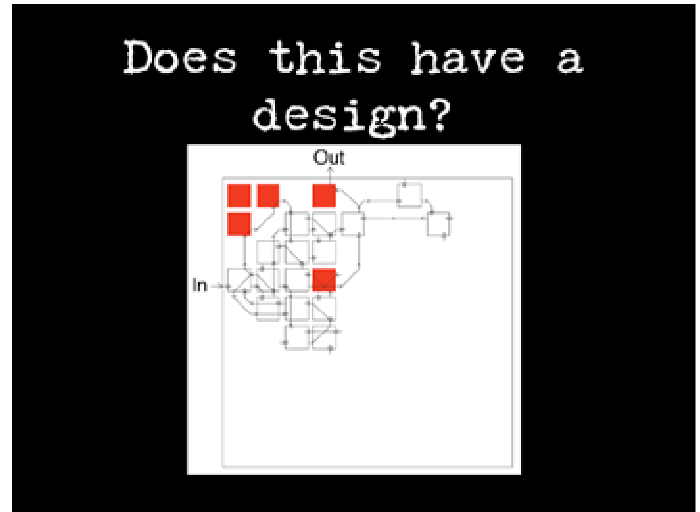
Using evolution, there is no real distinction between design and implementation. You basically build the thing, test how it works, and then combine/mutate to search the design space via the implementation space. As we've seen, the technique can explore regions of the design space that are not usually explored. The method doesn't rely on human understanding, which means it doesn't care about understandability, modularity, beauty, or elegance. And watching the progress of the process might not reveal a coherent direction. If a student reported any of the intermediate circuits much before generation 4100 when it settled into something that worked pretty well, any professor would easily conclude no rational progress was being made—the behavior of the in-

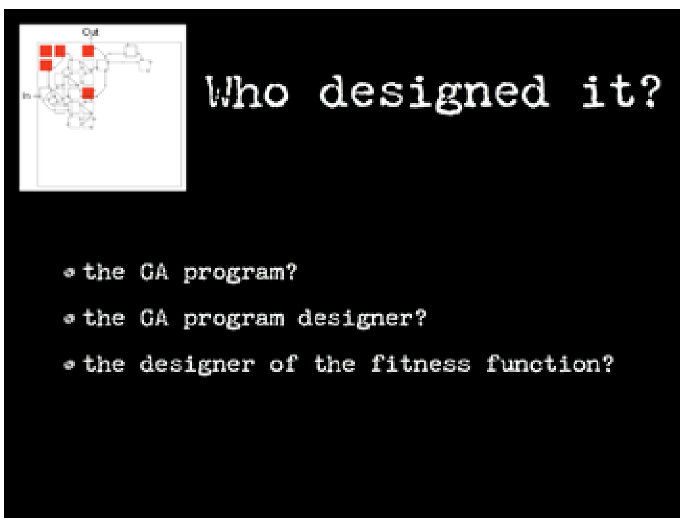
termediate stages are ridiculous except when considered in the aggregate (from the point of view of the analog integrator).

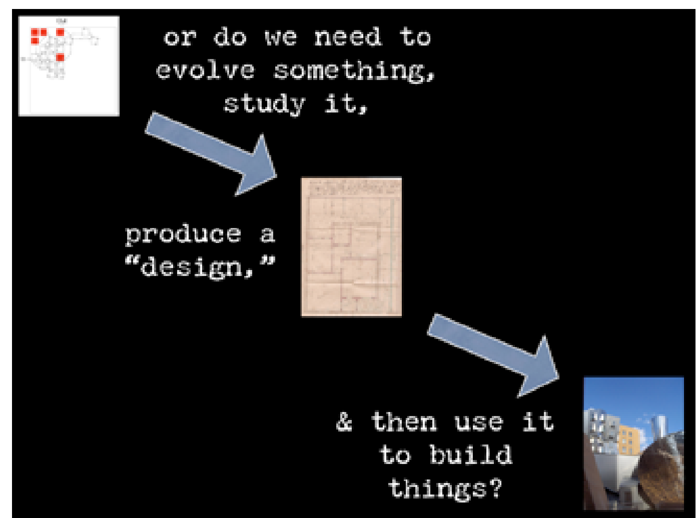
In response to this, Adrian Thompson questions whether using functional decomposition is the best way to think about the concepts of "understanding" and "explanation." In short, he is asking whether our reductionist way of thinking (which reaches its apex in computer science and software engineering) is serving us well.

Further is the question whether a process like digital evolution is actually producing a design. Do we require a human for something to be a design? If not, then who, for example, designed the circuit we've been looking at? The genetic algorithm? The person who designed the genetic algorithm program? The person who designed the fitness function?

Or do we need to understand the thing created and produce an understandable design before we can say that evolution has produced a design?







Getting back to ULS systems, it seems that evolution asks us to look even more deeply at the idea of designing beyond human abilities than even metadesign space 2 (complex adaptive systems) did. We need to look at mechanisms that design and produce code, perhaps from models understandable to people, but whose results might not be (at least not easily). Automated reasoning might be an important technique—this was the thrust of a lot of *automatic programming* work in the 1970s and 1980s.

But the tough question is what we can trust digital evolution to design.

As mentioned, ULS systems will contain multiple versions of the same component. Rather than permitting chance to dictate which one is used where, perhaps digital evolution could be used to produce code that selects

Design Beyond Human Abilities

- high-level mechanisms must be used to help produce the code for the ULS system
- generate parts of the system from models, use domain- and task-specific languages
- use automated reasoning as part of the generation process
- digital software evolution

What can we trust to digital evolution?

- combining multiple versions
- completing the details of implementation
- details of homeostasis

or combines the versions depending on context, arguments, conditions, etc. One could think of this as an extension of the idea of polymorphic functions—methods whose selection is based on its arguments. Other types of dynamic method selection an evolutionary approach would generalize include two-dimensional method selection which is based on the arguments and identity of the sender and context-oriented programming which selects a method based on the arguments, the identity of the sender, and any part of the visible context. The evolutionary approach is different because the selection can be more dynamic and also the selection process can be tuned by a fitness function directing an evolutionary process.

Or, evolution could be used to complete the implementation details below some threshold. An example would be completing the loop and loop-termination details in a program. These details are often the subject of errors by programmers.

Or, evolution could be used to determine the details of *homeostasis*, the property of a system regulating its internal state to maintain equilibrium and stability. It's worth noting that while a ULS system already is required to be self-monitoring, self-correcting, and self-repairing, if we go to this next step, perhaps questions arise whether it could be self-aware, self-designing, or even quasi alive.

ULS Systems

- self-monitoring
- self-correcting
- self-repairing
- self-aware?
- self-designing??
- quasi alive???

That is, ULS systems could become more like living systems. Many of the feedback loops in a living system are involved with the regulation of production of components, proteins, and other biochemical material that in turn regulates the production of other things including the regulators just mentioned. This is essentially the concept of *autopoiesis*. Autopoietic systems are

> *systems that are defined as unities, as networks of productions of components, that recursively through their interactions, generate and realize the network that produces them and constitute, in the space in which they exist, the boundaries of the network as components that participate in the realization of the network.*
>
> –Humberto Maturana, *Autopoiesis: A Theory of Living Organization*, 1981

An autopoietic system is one that is continually (re) creating itself. It is an interrelated network of components that build that interrelated network of components—that is, itself. And even though there might be deep connections to an outer environment, the boundaries of the system are distinct.

The concept of autopoiesis flows, in a way, from Immanuel Kant's analysis of living organisms:

Autopoiesis

"systems that are defined as unities, as networks of productions of components, that recursively through their interactions, generate and realize the network that produces them and constitute, in the space in which they exist, the boundaries of the network as components that participate in the realization of the network."

–Humberto R. Maturana, "Autopoiesis" 1981

... In such a natural product as this every part is thought as owing its presence to the agency of all the remaining parts and also as existing for the sake of the others and of the whole... the part must be an organ producing the other parts, each consequently reciprocally producing the others.

–Immanuel Kant. *The Critique of Judgment*, 1790

In living systems, living is the ultimate goal. And so the production of more living stuff and its adaptation of that living stuff to the surrounding conditions is in many ways more important than the nature of the living stuff itself—as long as it can self-perpetuate (more or less).

Our programmed systems are not thought of this way. What is important about a programmed system is what it does. The software systems we produce are *allopoietic*: Allopoiesis is the process whereby a system produces something other than the system itself. We are interested in this other stuff and don't care about the mechanism that creates it beyond the fact that it creates it. The health and stability of the mechanism is beside the point as long as the desired production happens. Allopoiesis is the manufacturing process abstracted.

Kant has the same idea about what mechanical devices are:

Autopoiesis

"In such a natural product as this every part is thought as owing its presence to the agency of all the remaining parts and also as existing for the sake of the others and of the whole . . . the part must be an organ producing the other parts, each consequently reciprocally producing the others."

–Immanuel Kant, "The Critique of Judgment" 1790

Allopoiesis

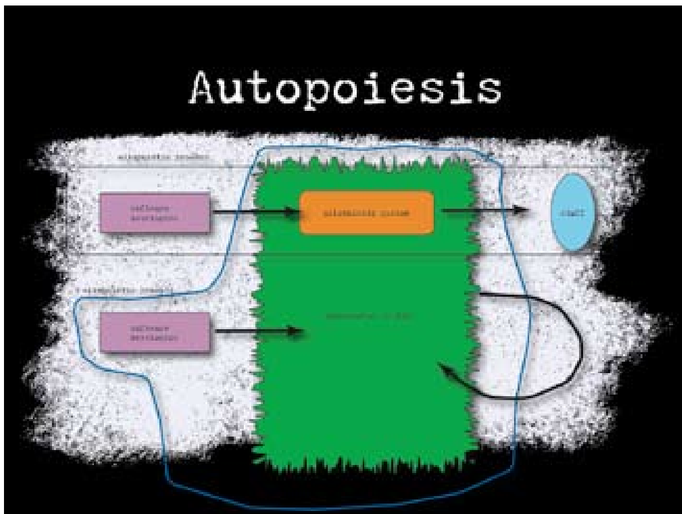
Allopoiesis is the process whereby a system produces something other than the system itself.

In a watch, one part is the instrument by which the movement of the others is affected, but one wheel is not the efficient cause of the production of the other. One part is certainly present for the sake of another, but it does not owe its presence to the agency of that other. For this reason also the producing cause of the watch and its form is not contained in the nature of this material. Hence one wheel in the watch does not produce the other and still less does one watch produce other watches by utilizing or organizing foreign material. Hence it does not of itself replace parts of which it has been deprived.

–Immanuel Kant. *The Critique of Judgment*, 1790

When a complex, programmed system needs to live for a long time, living becomes the ultimate goal. Coupling this with the need to produce the right answers, we face the following daunting problem: How can we structure a system which needs to recursively generate, realize, and produce itself as well as correctly produce something other than itself? That is, how do we combine the correct and efficient function of an allopoietic system with the urge to live of an autopoietic one that is continually recreating itself?

If we try to make a system more robust by adding lots of explicit exception handlers and error detection code the program becomes hard to understand and very difficult to maintain. It goes beyond what people are able



to write reliably. This is what Martin Rinard is warning us about.

Earlier research into building systems with autopoietic characteristics tried to define a *single* programming language or programming system to serve both purposes—to operate both at the level of arithmetic and logic, procedure calls and method invocations, and also at the level of feedback, visibility, noticing, and living. Any programming language that can operate at the allopoietic level is necessarily fragile, and so injecting those elements in the autopoietic system will break it as well.

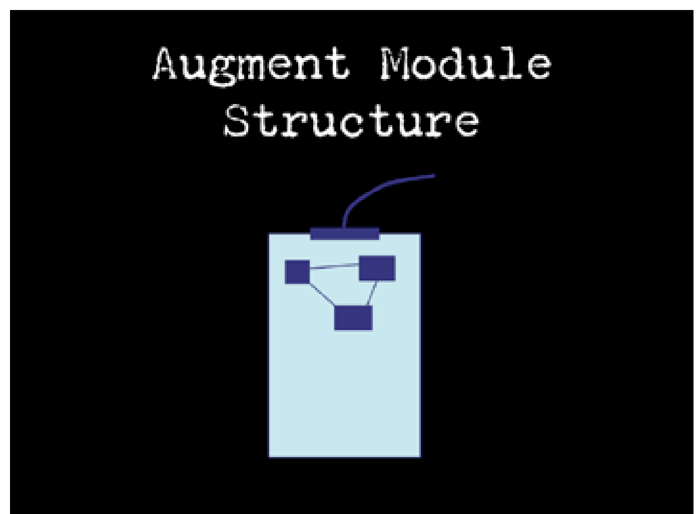
Perhaps it would make sense to separate the allopoietic part of a system from the autopoietic part. The encompassing autopoietic system needs to observe itself and its environment to know how to change and recreate itself. It must also have some ability to observe and affect the operation of components of the allopoietic system. The requirements of the two types of systems don't line up, and so designing an autopoietic language / environment for allopoietic purposes is unlikely to succeed. Maybe the allopoietic part of the system should remain in an allopoietic language (and runtime when that's appropriate) while the autopoietic part can be in a separate language—designed to do different things and with radically different characteristics.

The autopoietic language should have no ingredients that can cause errors of a nature as to crash a program written in it. A way to think of this is that it must be impossible to write a program with a certain class of bugs in it. Naturally, it's not possible for a person to write a program with no bugs, so what I mean by this must be elaborated. Some bugs are about the ultimate problem, such as getting differential equations wrong, updating cells in the wrong order, or forgetting parts of a specification, while others seem centered around the business of programming *per se*, such as typos, fencepost errors, failing to initialize variables, etc. These two classes of errors feel very different.

The structure of a system made this way would be of the allopoietic part of it—the part that does what the designers intended, such as banking or web serving—embedded in the autopoietic part, which would be responsible, in a sense, for keeping the overall system alive.

A possible structure for this is to attach a monitor or autopoietic interface—called an *epimodule*—to each or some allopoietic components in an allopoietic system. The purpose of the epimodule is to monitor the behavior of the component and to alter it when needed. For example, an allopoietic system would include all its test code. The epimodule would be able to pause the component, execute its tests, and examine the results. If a new version of a subcomponent becomes available, the epimodule can direct the component to unload / unpatch / ignore the old version, install the new one, run tests, and then either proceed or undo the whole operation.

One of the more important sets of capabilities of epimodules would be to be able to clone and then kill a module. Or, instead of killing the module, the epimodule could tell it to self-destruct.



Here is another possible way to envision the allopoietic / autopoietic structure. The blue boxes represent allopoietic code—the code that does the real work and red blobs indicate autopoietic code. The main big blue box is a component in a larger (allopoietic) system. The bump and string on top represent its normal interface and connections—think of it as the method signature and call graph. The darker (lower) bluish box inside can be thought of as the epimodule interface. Its job is to implement health- and repair-related operations within the

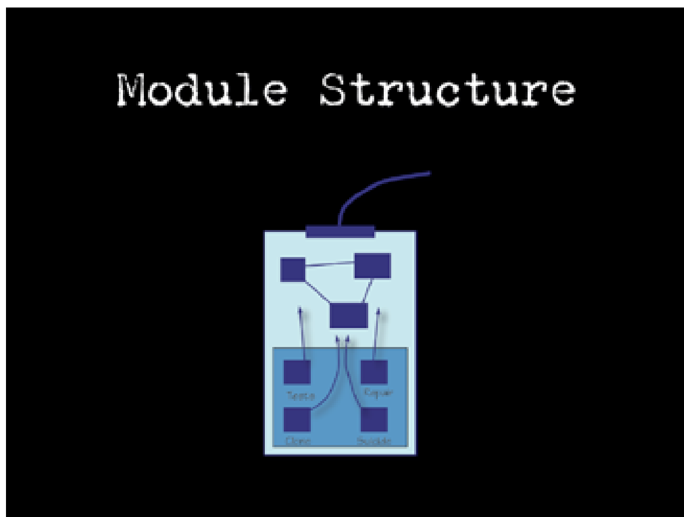
component, including things like the ability to run tests, to clone the component, to safely kill the component, or to repair data structures. There could be others as well, such as a tap into the local exception handling and event environments.

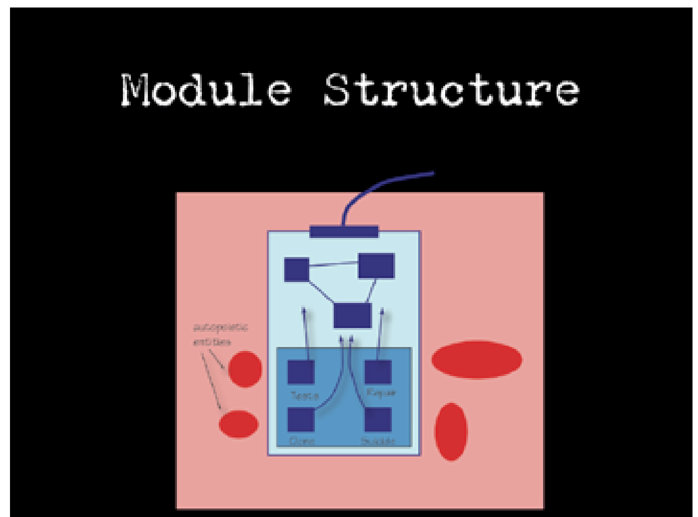
The red shapes outside the box, including the encompassing reddish rectangle with the rounded corners, represent autopoietic code. The odd shapes that change shapes and shades from the outside to the inside of the component (from oval to square and from red to blue) represent the epimodules that assist that component by keeping it healthy and keeping healthy the entire system within which that component exists.

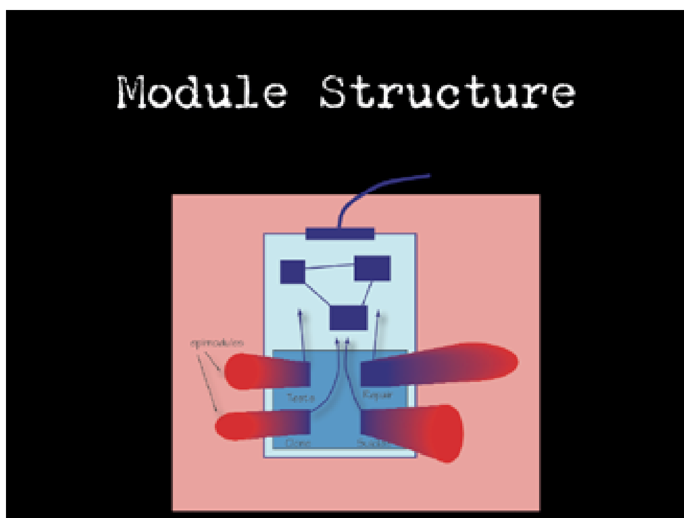
There is no need for epimodules to execute on the same computer systems as the allopoietic code it oversees. Epimodules can execute on another computer system or systems onsite or offsite. Or perhaps in the future, some of the many cores in the CPU can execute the autopoietic code. Epimodule execution can also be asynchronous with respect to allopoietic execution. For example, the epimodules (the autopoietic system) can include extensive simulations to set parameters (e.g. recompiling an allopoietic component with different compiler switches to better tune it for actual use), determine the best configuration, design and implement more appropriate glue, monitoring, or protection code.

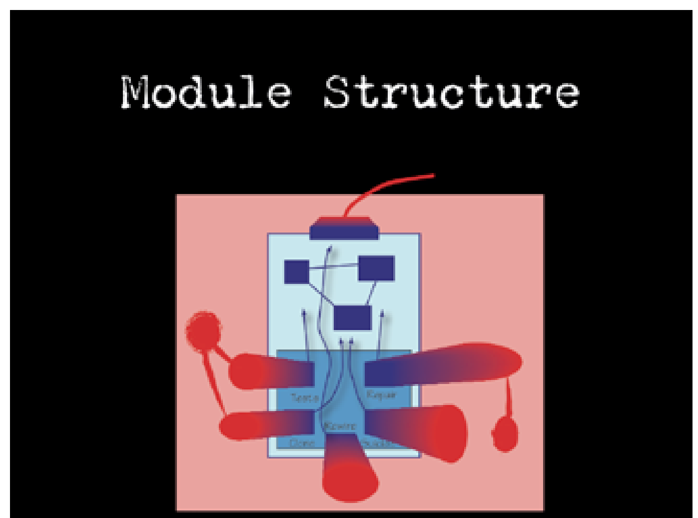
The red, blobby, autopoietic components communicate with each other through the reddish background substrate and perhaps by the action of autopoietic entities that operate in the substrate but which are not attached to specific allopoietic components.

We don't know much about what the autopoietic parts should be like. But my intuition says that nudging, tendencies, pressure, and influence will be more important than exchanging control and data or traditional control structures and composition techniques.









At the conservative end, the autopoietic parts could be declarative, mostly descriptive—perhaps a set of wiring diagrams for putting together the system from its components, and specifications of the conditions under which to run tests, to clone components, to do internal repair, etc. These descriptions and specifications would be interpreted much as a constraint system is to maintain the functioning of the system and its components. As such, the descriptions might be ineffective but never broken.

At the more speculative (and intriguing) end, the autopoietic parts could be written in a language that might be able to express only programs that are dimly aware of themselves and their surroundings, but aware nevertheless. The red, blobby epimodules could be like living cells or other living organisms that are attracted to components either generally or specifically—that is, some epimodules might be tailored for certain components while others could be general. The epimodules would attach themselves to components and sense conditions, both inside and outside the component, and exert pressure on the component to run tests, etc. They wouldn't interact through a traditional interface, but would sense as if sensing the density of chemicals and exert pressure as if by a suggestion or a push.

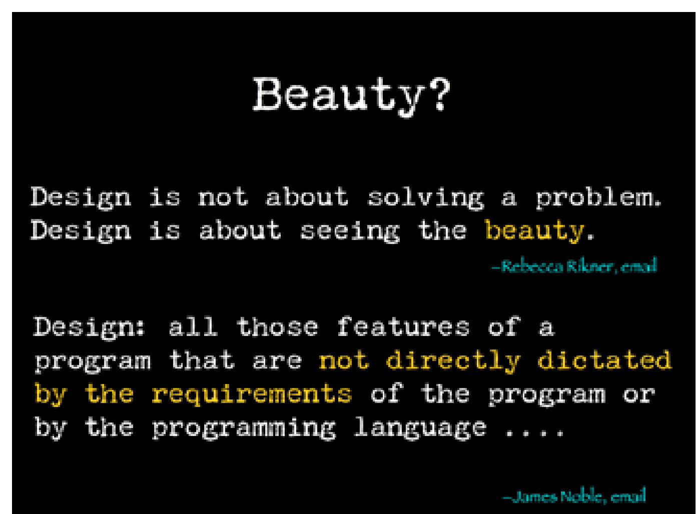
The substrate could be thought of as a medium for "smells" that would influence the activities of the epimodules. So, performance monitoring of a component could exude a hotspot smell, which when sensed by other epimodules would cause a migration of a set of components to a stronger cluster or a faster machine. Perhaps it would be useful for the substrate to have a geometry so that concepts of distance and expanding signals could be used to model accumulating evidence or to disregard irrelevant, anomalous spikes.

At this end of the spectrum, concepts like population and crossbreeding could make sense so that the epimodules could learn or evolve to be more effective. Perhaps epimodules would live from nourishment provided by a healthy system and its components. Perhaps epimodules would effect the initial installation using surrogates to optimize placement and connectivity, the real installation taking place after some initial experimentation—or perhaps the surrogates would assemble components or grow them from found / sought parts and pieces in a sort of in-place development / growth process.

The autopoietic part of the system to be subject to digital evolution—either in real time along with the allopoietic part of the system or on simulations that permit faster evolution.

But this leaves out what to many people is the most important part of design—design is about beauty. This aspect has not as much to do with problem solving and engineering, though many designs-as-beauty have or require such things. For many people, when they talk about a "good design," they are talking about something that is pleasing about it—what makes Apple products well designed and Dell products utilitarian (designed well for the job they need to do, but nothing to gaze upon lovingly). One could debate the extent to which beauty requires subjectivity, but that's the topic of a different talk.

Looking at this through the lens of biological systems—prompted by our exploration of digital evolution—the concept of *canalization* seems relevant. This concept is about the course of a developmental process: such a process is or becomes canalized if or when it reaches a state where the outcome is determined regardless of fluctuations in the remainder of the process, including the environment in which the process takes place. The word comes from "canal," and canalization roughly means that once you end up in one of the tributaries to a river, you end up in the river. It's an inevitable outcome.



Metadesign spaces 1 and 2 (design up front and incremental design / complex adaptive system-based design) are different from metadesign space 3 (evolution), possibly, because the first 2 are done (mostly) by people, and therefore a sort of canalization takes place based on our humanity. Namely, human designs are all canalized by beauty, elegance, modularity, understandability,

Canalization

A developmental outcome is canalized when the developmental process is bound to produce a particular endstate despite environmental fluctuations both in the development's initial state and during the course of development.

– (adapted from) Andre Ariew, "Innateness is Canalization: In Defense of a Developmental Account of Innateness"

Canalization

maintainability, and mathematics. Try as we might, we cannot explore all of the design space, because the parts of it that contain designs without those characteristics are impossible for us to see. The design of the tone discriminator we looked at is not comprehensible *at all*. When people are asked to design a tone discriminator using a 10x10 portion of the FPGA, they cannot do it as compactly as the design we looked at.

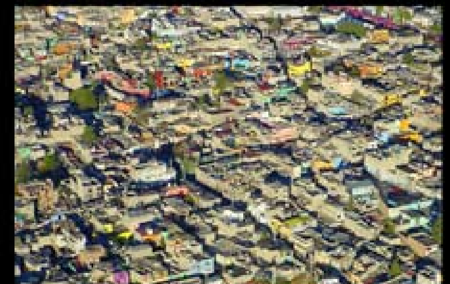
This means there is a blind spot in our design visions. There are things we cannot see.

Do we understand design better now? I looked at some extreme situations, as many philosophers do to try to understand something. In this inquiry I've aimed high, not to get you to accept my ideas, but to get you to make your own, better ones.

Human Designs are Canalized by:

- beauty
- elegance
- modularity
- understandability
- maintainability
- mathematics

Design Beyond Human Abilities



Richard P. Gabriel