Toward Simplifying Application Development, in a Dozen Lessons

Mel Conway

It has taken me over fifty years to learn these lessons about simplification of application building. This paper describes how the lessons showed up.

Incentives Affect the Product: The Origins of Conway's Law

I was a graduate student at Case Western Reserve University between 1956 and 1961. That was the time when digital computers started replacing punched cards in IT shops. Because computers were so new there were few programming tools, mostly assemblers, written by the hardware manufacturers; it would be another decade before IBM unbundled its software, creating an independent software market. So the Case Computing Center, where I lived when I wasn't in class, was involved in compiler

Twitter: @conways_law
#HumanizeTheCraft
January 3, 2017
conway.mel@gmail.com

research. One thing we noticed was how clunky the tools being written by the manufacturers were compared to the stuff we were making. That was the first hint that *the nature of the design organization influences the designs it produces*. As I learned later, sometimes the influence is the training or world-view of the designers and sometimes it is the structure of the design organization itself.

During that period, in 1959, the first draft of the COBOL language specification came out. From the implementor's point of view it was a pig of a language, and the manufacturers and Government agencies populating the specification committee apparently envisioned compilers that looked like multi-phase magnetic tape sort-merge data-processing runs. I thought we could do better and I took on the challenge of designing a one-pass COBOL compiler. I finally published the details in 1963, after I left Case and was in the Air Force. The two papers[1,2] described four innovations; they were recognized as a game-changer and influenced the compilers of several manufacturers. During the struggle to produce this design I learned
Lesson 1: ***You can make it even simpler if you keep working at it.***

After graduating I spent 1962 and 1963 in the Air Force Electronic Systems Division, where I had a view of many of the Air Force's large-system procurements. That's where a perverse design cycle slowly became clear: risk avoidance and managerial empire-building lead to an overly large estimated project size, which rules out simpler designs that might be implemented on a smaller budget. In addition, humiliation and the (sometimes incorrect) belief that time would be lost if a project was reorganized had the effect of locking in the original design. Freezing the initial design flies in the face of Lesson 1; often it is only by trying to build something that you learn that it doesn't work very well and there is a better, simpler design. Arriving at a simpler design can suggest simplifying the organization and reducing the budget, which can be contrary to conventional incentives in an organization.

There were two rules in Government procurement that tended to degrade the products: waterfall design, and arms-length hand-off between the phases of the waterfall with separate procurements for each phase. Hardware and software procurements were independent, with hardware chosen first; this

---

[1] "Design of a Separable Transition-Diagram Compiler" http://melconway.com/Home/pdf/compiler.pdf
[2] "Arithmetizing Declarations: An Application to COBOL" http://melconway.com/Home/pdf/arithmetizing.pdf
[2] "Arithmetizing Declarations: An Application to COBOL" http://melconway.com/Home/pdf/arithmetizing.pdf

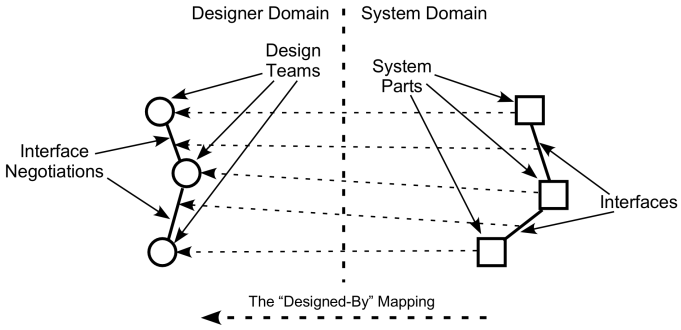
often meant that the software people were handed a target computer that was far from what might have been chosen had they been permitted to participate in developing the requirements.[3]

The Government lawyers required these arms-length hand-offs to avoid real conflicts of interest in procurement, but the rules were producing unanticipated consequences that could lead to lower quality.

> My own experience as a practitioner was exactly contrary to the Government practice: if I know I'm going to have to build the thing I'm going to put in the extra work to make building it simpler. (On the other hand, if I'm an ambitious manager, simplifying my project and reducing its budget might run contrary to my interests.)

After leaving the Air Force I tried to make sense of my experiences, and I set out to understand how the natural incentives of risk avoidance, defensive staffing, and arms-length waterfall procurement acted perversely to degrade quality. I found that there is a one-way mapping we can call "designed-by" that goes from each part of the system being built to the design group that designs this part. That's true by definition, and the following principle derives naturally from the realities of how systems are built:

> The principle: If part A and part B of a system have to interface, then the designer of A (call it dA) and the designer of B (call it dB) have to communicate in order to agree on the interface specification. The "designed-by" mapping also applies to interfaces; the interface between A and B maps into the subgroups of dA and dB that negotiate the interface specification. If you follow this logic as you scale up your view to the whole system, you see that "designed-by" is a structure-preserving relationship that goes *from all parts of the network that is the whole system to all parts of the network that is the whole design organization*.[4]



[3] The waterfall process produces several products along the way, including both specification documents and software. Conway's Law applies to each in turn, as well at to the entire waterfall itself.

[4] This relationship is one-to-one (bidirectional) only if each design team designs exactly one part of the system. This is why a mapping always exists from system to designer but cannot always exist in the other direction.

Eventually I put these observations into a paper that was published in 1968[5].

The fact that "designed-by" suggests an arrow going from the system to its designer is contrary to the obvious and intuitive causal relationship that the design organization determines the structure of the system. That's true; the design organization does determine the structure of the system. But the mere act of organizing the design group has already influenced the design:

> *Every organization choice rules out some design choices*. If we determine the structure of the design organization first, certain system structures that don't map to the design organization's communication structure cannot be pursued by the design organization because the communcation paths don't exist.

Hence **Lesson 2:** ***If you want the cleanest possible product you have to find the simplest possible design before organizing to build, or else you have to be prepared to reorganize.***

Conway's Law is often simplified (by me as well as others) to: "Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure." This is a suggestive qualitative simplification of the principle stated above.

> So how do you use it? The importance of the principle as a guide to action is not that your design organization determines the things you can design; as a guide to action, that's not particularly useful. The importance of the principle as a guide to action is that you need to know that your design organization *is* **keeping you** from designing some things that perhaps you **should** be building. The principle creates an imperative (1)to keep asking: "Is there a better design that is not available to us because of our organization?" and (2)to be open to changing the organization if a better design is found.

When a large software project was being planned, one of the first tasks was typically to scope the budget by estimating the magnitude of the work and planning for resources. Note that the construction effort is typically sized before the design is settled; I noticed this on multiple occasions. In the construction of mission-critical applications, particularly early in the history of software after a few big failures, risk avoidance was a major consideration. There was a tendency to overstaff projects defensively. This was before (and, regrettably, also after) Fred Brooks wrote "The Mythical Man-Month"[6] in which he described IBM's painful lesson while building OS/360 that adding people to a project that was late invariably slowed it

---

[5] "How Do Committees Invent?" http://melconway.com/Home/Conways_Law.html

[6] Frederick P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition,* Addison-Wesley Professional, 1995

down. This was an alien idea for industrial-age managers who had learned that people were production units that could be piled on like bricks; hence Brooks's title.

I remember a case in which a company put out a bid for a compiler. There were two responses, one by a large corporation and one by a small group whom I knew well. I knew that the small group, which bid a much lower price, was expert at this kind of work. The big corporation got the job. It was clear what was operating: there was a fear of failure by the buyer. If the small company failed it was evidence of poor judgement on the part of the person who made the purchase decision. If the big company failed it was evidence that the problem was indeed a difficult one. Everybody understood: "Nobody ever got fired for choosing IBM."

Partition the Solution

From the very beginning my greatest interest has been understanding why writing software was hard for some people and impossible for most others, and then doing something about that. This led to a learning process that has continued to the present. I think that my approach to the problem has been the result of an unusual beginning.

When I started in 1956 the dominant record-processing technology was punched cards. There was no such thing as computer science, so I had to be in the Math department. The education level of the workers in a typical business IT shop was at the 4-year liberal-arts college level or less. Job training was provided by IBM. Files were decks of cards, with one card per record (if possible). File-processing functions, such as sorting, merging and separating files, and printing/totaling, were built into machines dedicated to each function. Setting up a processing run such as monthly utility billing consisted of choosing a sequence of machines to run the files through, and having an appropriate wiring panel for each machine. Wiring a wiring panel was not what we now call programming; it was mainly identifying the data fields on the cards used by each machine.

When stored-program computers arrived they were pretty wimpy by today's standards. The three computers I programmed at Case were vacuum-tube machines with typically about ten thousand characters of memory. To do any reasonable work the programmer's challenge was to stuff twenty pounds of computation into a five-pound bag. People trained in file processing turned big jobs into multi-stage file processing runs. People with some

mathematical training devised other strategies such as inventing a concise, domain-specific interpreted language whose interpreter would sit in the computer memory alongside the data being processed. In this case the technical challenge was devising a language expressive enough that, after subtracting the space occupied by the resident interpreter, the net amount of computation was substantially increased. I saw an excellent example of this on the IBM 650, my first machine.[7,8]

The 650's working memory was a 2000 ten-digit-word magnetic drum that rotated at 12,500 RPM; the sound of the drum was a loud squeal you just had to get used to. Bell Labs (remember them?) wrote, and IBM distributed, the Bell Interpretive System for scientific computation.[9] This provided a set of two- and three-address mathematical operations executed by an interpreter that occupied 1000 of those 2000 words. It implemented floating-point aritmetic and a complete set of transcendental functions, none of which were native to the 650's instruction set. A lot of computation could be done in the remaining 1000 words. This was a breakthrough in increasing access to scientific computing. My first IT job was as an IBM trainee supporting the 650 installation at Cleveland Pneumatic Tool, the manufacturer of Boeing 707 landing gear. When I was there the 650 spent its time chugging away at large matrix inversions. My exposure to the Bell System led to **Lesson 3:** ***Expressive domain-specific intermediate languages can give the combined solution a lot more bang for the buck****;* I have drawn on this lesson repeatedly.[10,11]

Static Is Good

The arrival of magnetic-tape stored-program computers to replace whole punched-card shops (most importantly, the IBM 1401, the first mass-produced transistorized computer for business applications)[12] required the conversion of the IT labor force from wiring panel wirers and card machine operators to programmers. IBM devised an ingenious tool called RPG (Report Program Generator) that required very little training over and above wiring skills. I studied RPG and learned **Lesson 4**: ***Making application***

---

[7] http://www.columbia.edu/cu/computinghistory/650.html
[8] http://en.wikipedia.org/wiki/IBM_650
[9] http://bitsavers.informatik.uni-stuttgart.de/pdf/ibm/650/28-4024_FltDecIntrpSys.pdf
[10] Example: Byte-coded transition-diagram syntax definition of COBOL used for parsing source code: http://melconway.com/Home/pdf/compiler.pdf
[11] Example: Boolean matrix representation of COBOL Data Division semantics: http://melconway.com/Home/pdf/arithmetizing.pdf
[12] http://www.columbia.edu/cu/computinghistory/1401.html

development accessible to a large number of people with general educations requires elimination of algorithms. Sequential planning is hard for a lot of people.

Powerful domain-specific application languages like RPG are mostly static modifications of a built-in run-time process that is the same for all applications in the problem domain. In the case of RPG and other file-oriented report generators of the time the underlying process was the logic of the record-processing loop. Other examples of this principle (and the corresponding underlying algorithm) were IBM's Query By Example (SQL execution) and the user-interface builder in Microsoft's Visual Basic (the event loop). Hence **Lesson 5:** ***The set of all applications can be partitioned into classes defined by their underlying algorithms; an effective application language for each class presents a static parameterization of the implicit run-time algorithm***.

Lesson 6: ***One purpose of an application-development language that is meant to be both simple and powerful is not to express algorithms, but to hide them***, makes explicit that *application-development languages and programming languages belong to different species*.

Simplify the Developer's Life

Until time sharing came along a single application-development iteration in a large shop typically involved an overnight turnaround from submission of a source-program card deck to receipt of a printout with the output of the program under test. The "operating systems" of the day queued up input source programs followed by their data for later compilation and execution one at a time. The word "interactive" had not yet been applied to computers, and programming was a cumbersome process with a 24-hour cycle time. Our experience with one-pass compilers taught us **Lesson 7**: ***Make the programmer more productive by giving him or her immediate feedback***. One-pass compilers (and the BASIC interpreter) plus the introduction of time sharing for on-line character-terminal access transformed turnaround times from overnight to often less than a minute.

As a consultant in the 1970s one of my clients was Monroe calculator, where I was exposed to programmable calculators. The "programming language" of a programmable calculator is the same keyboard the operator uses to solve arithmetic problems; the only difference is "learn mode," in which the machine captures the sequence of keystrokes the operator enters to solve a

problem and then repeats the sequence on command.[13] This led to **Lesson 8**: ***Don't make the developer distinguish between "programming" language and "execution" language; forcing this duality increases the cognitive load and introduces distracting artifacts into the development process***.

In the early 1980s a startup I helped form applied Lessons 3, 7, and 8 to the construction for Apple of Mac Pascal, which compiled line-by-line during text entry, effectively eliminating the compile delay. What made Mac Pascal possible was a compact "tokenized" internal representation of the Pascal program that could be translated back to the Pascal source text and was in practice indistinguishable from the source text, while executing at an acceptable speed. A free byproduct of this design was single-stepping and breakpointing, all at the source level, and on-demand execution of Pascal expressions during debugging.[14,15] Users' experience of Mac Pascal (and similar experiences elsewhere) raised the bar for programming tools; immediate turnaround, the elimination of debugger artifacts, and transparent source-level debugging became the state of the art. The profession was learning the importance of taking human factors into account in the development process.

Humanize the Craft

The introduction of the Macintosh and Windows in the early 1980s was a painful disruption to the programming community. Developers had to change their application models from run-to-completion or wait-for-the-next-input to event-driven applications in which the program gave up control, waited for the next user event, and then executed a process determined by that event. Tools like Microsoft Visual Basic helped to reboot the application-development process; they applied Lessons 4 and 5 to user-interface design by offering the application developer a static pictorial user-interface builder. I began to wonder: do these same lessons suggest *a static representation for the whole of an event-driven interactive application*? The obvious answer being pursued at the time was the dataflow (wiring or plumbing) network. A complication was that in a network with data sources

---

[13] I found FORTH to be a particularly interesting example of learn mode: http://en.wikipedia.org/wiki/Forth_(programming_language) .

[14] The 16-bit internal token language, which was derived from the formal definition of Pascal, was another example of a Lesson-3 domain-specific intermediate language that augmented the power of the tool within the memory constraints of the ealy Mac.

[15] Two more Lesson-3 examples: as with COBOL, the Pascal syntax was byte-coded for parsing by a syntax engine, and the entire analysis-code generation process was driven by a byte-coded stack machine.

at one end and the user interface at the other, there were flows in both directions: data toward the user interface, and events away from the user interface. You could devise a wiring language to do that (teams at IBM and Apple, among others, did) but there were practical problems. The biggest obstacle was reuse: building new wired components by encapsulating already-created wiring diagrams and reusing them in unanticipated contexts. It wasn't fruitful. The fact that data flows and event flows were different types and went in opposite directions made the encapsulation of wiring diagrams into reusable components that were broadly useful practically useless. One obvious solution was to combine data and events into a single type with a single connector type, but this led to bidirectional flows and terrible complexity problems in implementation. After literally years I found a hybrid solution in **Lesson 9**: ***Event-driven applications can be described with unidirectional flow diagrams***. The wiring connectors on components handled only unidirectional data. Event messages were not part of the flows but were hidden, and their paths were implied by the data paths.[16]

Now the problem was: what kind of an executing program does such a flow language translate into? I could imagine a diagram-to-code translator, but compiling a flow diagram into a conventional program would violate Lesson 7: make the internal and the external representation of the program the same. I needed an execution engine that *was* a flow network, so I created one. I built a prototype wiring/execution tool on a laptop and started showing it around. When I showed up at the large corporations such as Sun and IBM I was confronted with the usual sign-in form at the reception desk that says: Anything you show us belongs to us unless you have previously disclosed it. Realizing that a flow-based computer was indeed an invention, I solved the prior disclosure problem by patenting the prototype's execution engine[17].

Two events over 30 years apart combined to reframe the inquiry in more humane terms.

1. In the 1970s my wife and I were involved in the formation of two private primary schools whose educational philosophies were inspired by the work of primary-education pioneers such as Maria Montessori

[16] Event flows were rendered unnecessary by the addition of one data type; see http://vimeo.com/151020589 . (Time: 25:58)
[17] US patent 6,272,672 "Dataflow Processing With Events" https://www.google.com/patents/US6272672

and Georges Cuisenaire. That's where I learned that *the path to a child's brain is through the hands*.

2. A few years ago I had an epiphany while watching a baby grandchild struggling to grab a Cheerio and put it in his mouth: *I was witnessing one step in the years-long process of building the hand-eye-brain system built into every human*.

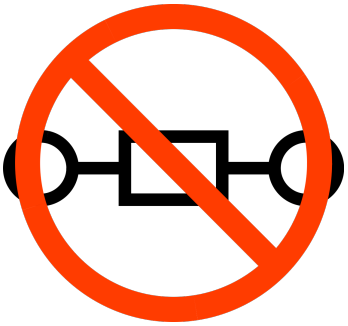
After the Cheerio incident I realized that coding at a keyboard is an unnatural act, and something different needs to replace it if we are to humanize application building. The turning point was **Lesson 10**: ***The way to make application development universally accessible is to harness the tremendous investment Nature has made in every person's hand-eye-brain system***.

Why *universally* accessible? Why not just *more* accessible? One simple answer is that it's just a research tactic: force yourself to think outside the box by shooting for the moon.

So the goal then became: make application development accessible to everyone. That suggests it's going to be taught in primary school. We have three important examples of key technologies that started as the property of priests (in the case of software, that's us): arithmetic, writing and calendar. *They are now taught in first grade.* These powerful existence proofs offer hope that the goal is approachable for software.

Primary-school children are a useful target audience because (1)they stand as a concrete proxy for the population as a whole, and (2)Montessori and Cuisenaire have given us powerful guidance for this group: *make it hands-on*.

In order to move away from thinking about coding at a keyboard I started to think in terms of hands-on artisans. The picture of the potter at her wheel presented itself; this is what building software as a hands-on activity might feel like. What guidance does this image give us? Certainly, the file-oriented input-process-output model we have taken for granted in the development process doesn't represent the potter at all well. Yet the input-process-output model is importantly embedded in our history; consider compilers, sequential file processing, and Unix pipes. But we must abandon this model; hence



Lesson 11: ***The input-process-output application-building model must be replaced by a transform-in-place model***. Instead of source and object files we should be thinking in terms of *modifying the working material in stages*.

Lesson 12: ***To simplify application development to the point of being accessible to the entire population, the tools must act like hands-on tools***. What does that even mean in the case of software?

The Eleven Properties of a Hands-on Tools, and Experimental Development

Spending time playing with my prototype gave me more insights into the hands-on application development process. I learned that the concept of "starting" a program is alien to the transform-in-place model; as soon as you drag a component onto the workspace, even before you attach the first wire, it is running. Attaching or removing a wire changes an input to a component, and the consequences of that change ripple though the network.[18] This (plus a few other rules) guarantees that the behavior of the application under construction and its appearance will remain in sync in response to every change to the application. This continual synchronization between the model that you have your hands on and the outputs of the executing application makes it possible for the model to be a convincing proxy for the application.

[18] The implicit underlying process for this class of applications is the scheduling algorithm that brings the network back into equilibrium.

A hands-on construction tool will have the following eleven properties.

1. **Unified**. There is only one program representation, no "source/object" duality.
2. **Symmetrical**. The tool and the application being built are peers. Your next move can be on the user interface of either one or the other.
3. **Choosing over Composing**. You are never asked to construct grammatical input; rather you are shown enough information to make a selection among choices presented to you.
4. **Always on**. When a component instance is created in the workspace of the tool, it is already running.
5. **Inspectable**. All parts of the application can be inspected and the values so obtained can in turn be inspected.
6. **Intervenable**. Provided that doing so does not contradict the definition of an existing component used by the application, you can modify any part of the application.
7. **Immediate**. Every modification you make is immediately reflected in the behavior or the program you are building.
8. **Predictable**. No surprises. Small changes lead to predictable outcomes.
9. **Transparent**. The tool supports the illusion that it is invisible and you have your hands directly on the working material.
10. **Interactive**. You are in an easy dance with the tool and the working material, like a child playing with a construction toy.
11. **Reversible**. You can undo your most recent changes.

These eleven design principles for a hands-on tool, when combined with the unidirectional flow application model, summarize my current thinking about humane application building.

If well executed, a tool based on these principles will support and encourage an experimantal development style. There is an important corollary to these design principles that must be experienced to be appreciated:

> *Taken together, these principles make possible routine development with live data. Developing with live data qualitatively changes the concept of experimental development.*

With live data flowing through the construction tool, I am finding that my workflow is becoming more “childlike”, and I can play around with the near-bulletproof flow model just to see what happens. This is hacking in its original sense. On reflection I find this to be OK; it tells me that the tool is becoming more friendly to my target audience.

Epilogue

The next step of my research is to apply this set of insights to current thinking about distributed applications. My prototype started as a monolithic application. I have modified it to the extent that the user interface components now can run in a web browser rather than being integral to the construction tool, with the rest of the wiring diagram sitting behind a web server. The challenge is now to generalize the wiring connections so they work over sockets or any other messaging medium.

> This is the current challenge: the developer must be able to build interactively any application whose components can be anywhere on the network and that is represented in its entirety on the user interface of a tool that conforms to all the hands-on design principles.

• • •

Of the twelve lessons presented in this paper, the first is the most important:

> You can make it even simpler if you keep working at it.

The journey described in this paper is the best proof I can offer of the power of this lesson. The twelve lessons appeared, one by one, over a half century, culminating in the unidirectional flow model and the hands-on design principles.

The task has only begun. Here is the long-term goal:

> A single developer will be able to wrap her mind around,
> and build, a system of any size that has, and can keep, integrity.

Whether or not this goal will ever fully be reached, it is the North Star I believe we can use to guide our research.

Revision History

November 15 2016 version: the list of properties of a hands-on construction tool has been expanded from 6 to 8.
December 7 2016 version: the list of properties of a hands-on construction tool has been expanded from 8 to 10, and the discussion of experimental development style is expanded to include using live data.
January 3, 2017 version: the third design princople is added.