The Goal

A Substantial and Important Role for Non-programmers in Application Development

1

We want to make the business people, and end users in general, first-class participants in the development process.

Our Frame of Reference

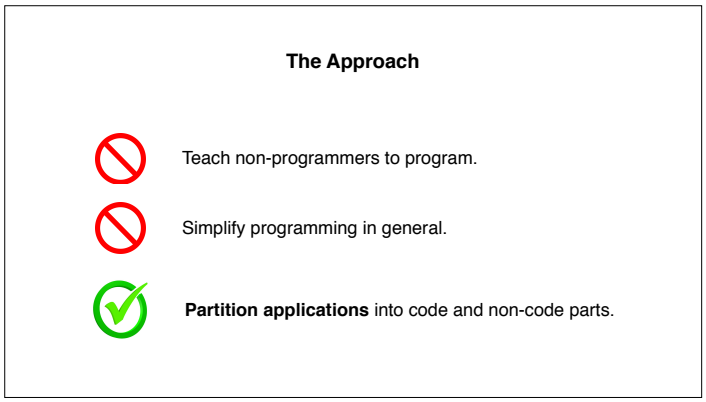
The Term	What It Means
Applications	Interactive, Graphical UI, Consumer and Business
Easy to Build	Everybody Has the Skills to Understand How They Work and How to Build Them

2

Some definitions.

Applications. We're not talking about climate modeling. We're talking about the interactive apps on your phone and on the desk of a customer service agent.

Easy to build. We avoid depending on skills that only some of us have. We favor depending on skills that we all have, i.e., that we all learned in the first few year of our lives. (Hint: that means hand-eye coordination.)

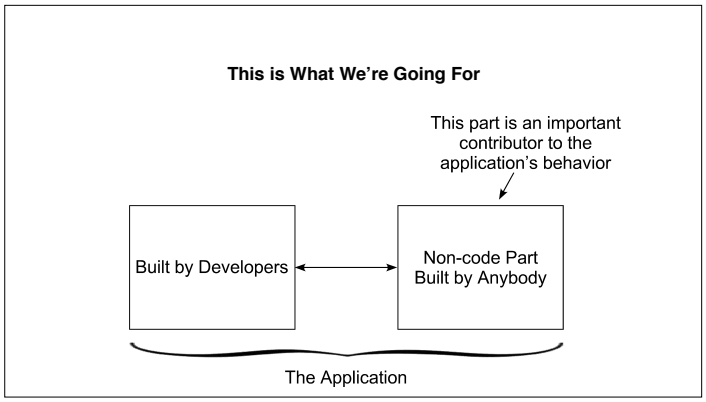


3

We are not eliminating programming. We are **partitioning applications** into

1. a part that is coded and built by developers, and
2. a part that has a much simpler no-code, no-algorithm conceptual model that can be built by non-programmers

---



4

BUT, the non-code part has to be important and relevant.

Two Guiding Principles for the Non-code Part

Humane Builder Experience	Concrete (Non-textual) Application Model
"**Hands-on design principles**" Exploit the hand-eye coordination we all share "**Fluidity design principles**" Minimize cognitive context-switching friction	"Left-to-right **Flow model**" (Plumbing, **Wiring**)

5

But first, here are the design principles behind what you will see.

The **Humane Builder Experience** principles evolved slowly over almost 60 years. You will see almost all of them in the demo. I'll elaborate on them in the next two slides.

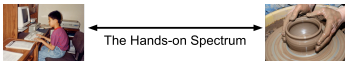
The **Wiring Model** began in 1992 and is pretty much unchanged, although its implementation in the wiring tool has evolved radically, driven by the need to realize new Builder Experience principles as I discovered them.

These principles are not separable. You will see their synergy in the videos I will be showing you.

Hands-on Tool/Application-Language Design Principles

(use universally human skills)

Immediate	You experience no delay from change to effect
Continuous	Your small changes ➔ Predictable effects
Interactive	Each step suggests what you should do next
Transparent	You have your hands on the working material
Inspectable	You can inspect any part at any time
Modifiable	You can modify anything you've built at any time
Reversible	You can easily undo your recent moves



6

Consider this thought experiment. You are in the business of making and selling clay bowls. But the technology of the potter's wheel doesn't exist. What you do is write a bowl-making script in a text editor and, once you are happy with it, you email it to China. A couple of days later (overnight if you pay extra) DHL delivers the number of bowls you ordered.

Sound ridiculous? Change a few words and that's how programming was done when I started in the 1950s.

The Hands-on idea is to push application building toward the right end of the hands-on spectrum.

7

Fluidity Tool/Application-Language Design Principles

(minimum cognitive context-shifting friction)

Unified	Source language = Execution language (no debugger)
Self-revealing	Choose among visible choices, don't construct
Symmetrical	Wiring tool & application being built are peers
Always on	No start/stop during development
Alive with your data	See application data in the wiring tool as you build

Fluidity means you don't have to pay the mental price of context-shifting, for example,

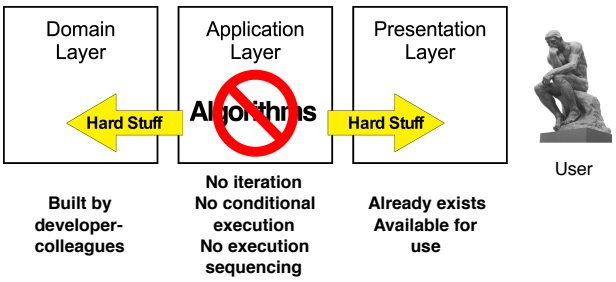
- switching from one language syntax to another,
- switching from one execution model to another,
- switching from one tool (with its peculiar rules) to another.

Everything you need is right in front of you, now. (Contrast to: HTML, CSS, Javascript, DOM, and that's just in the browser.)

All five items in this list contribute importantly to fluidity and you will see them all in the videos I'll show you.

8

To Make the Application Layer Friendly to Non-programmers:



To make the application layer friendly to non-programmers, push all the stuff that makes programming hard (namely, the stuff that makes algorithms powerful) out to the other two layers.

The question is: after removing all this power, is anything useful left?

The answer seems to be: with the right organization of the other two layers, yes. What remains in the application layer is a language for describing and executing use cases. It takes the form of a wiring diagram.