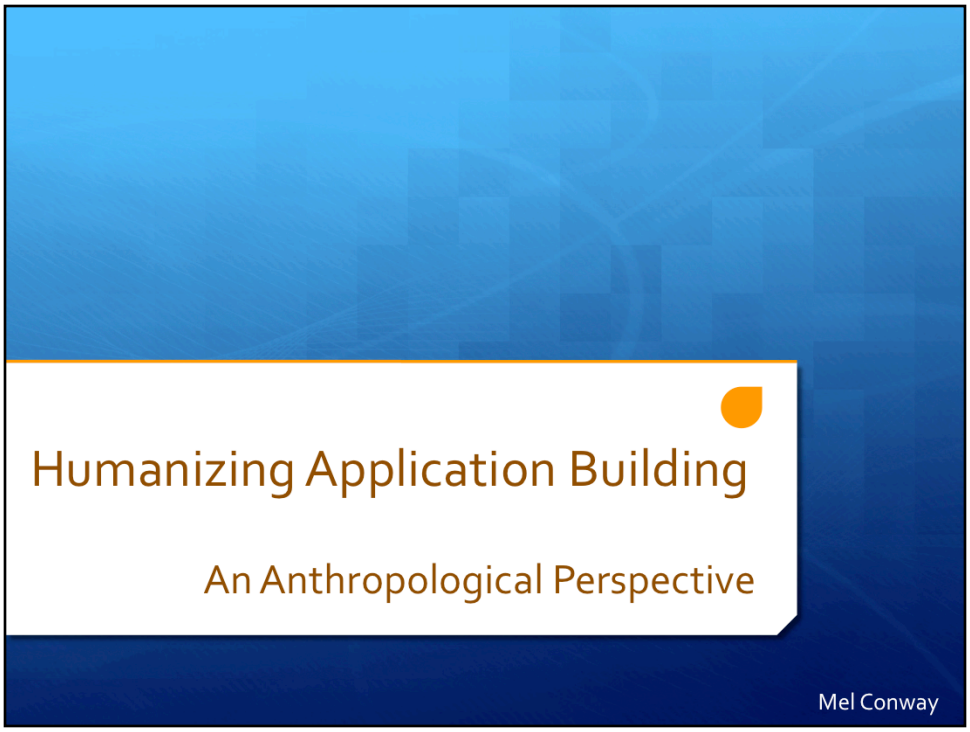
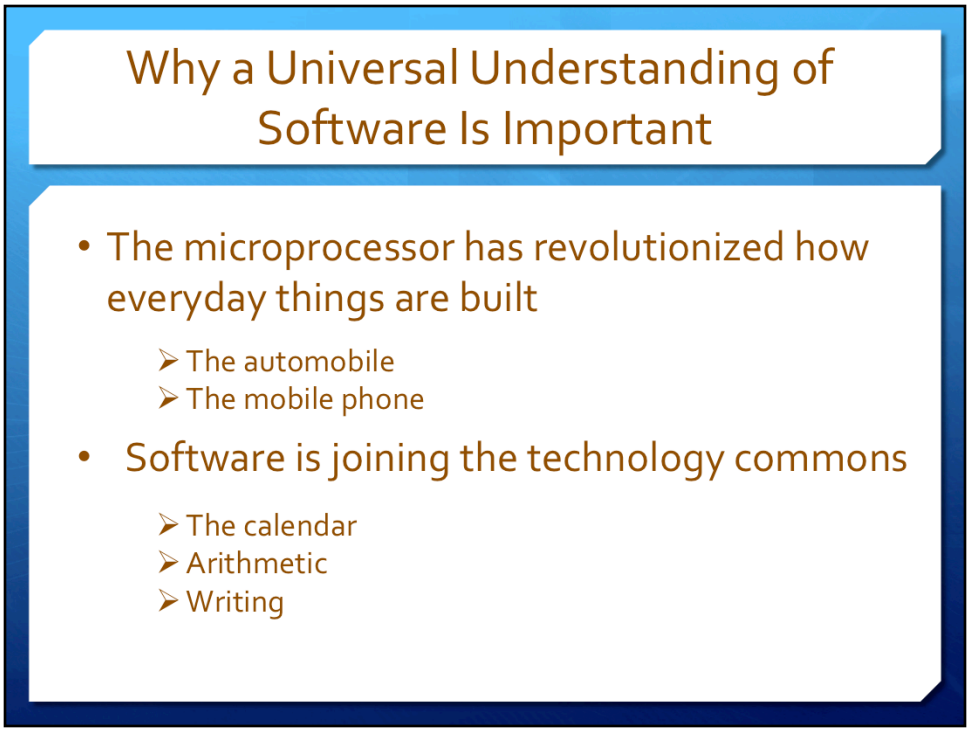


This presentation is not just about a new or better way to build software. The project is more ambitious than that.

I started out posing this computer-science problem: "Can I find a simpler conceptual model for computer applications that will be accessible to more people?" As a model evolved and I experimented with the construction tool I was building for this model, the problem started looking different. I began to see the construction of software in the context of the whole sweep of human tool use. This is a very, very long progression that began over a million years ago with chipping flakes away from a stone to make a hand-axe. Manual tool use is a big part of what makes us human, and we spend the first few years of our lives working very hard to acquire the skills that allow each of us to join this long procession of human tool users.

What hacking at a handaxe, peeling potatoes, and performing brain surgery all have in common is their dependence on the hand-eye-brain system that we are all born with and that we continue to extend for years thereafter. **My purpose is to bring software construction into this conceptual framework.** This anthropological view is the perspective from which I now think about software construction.

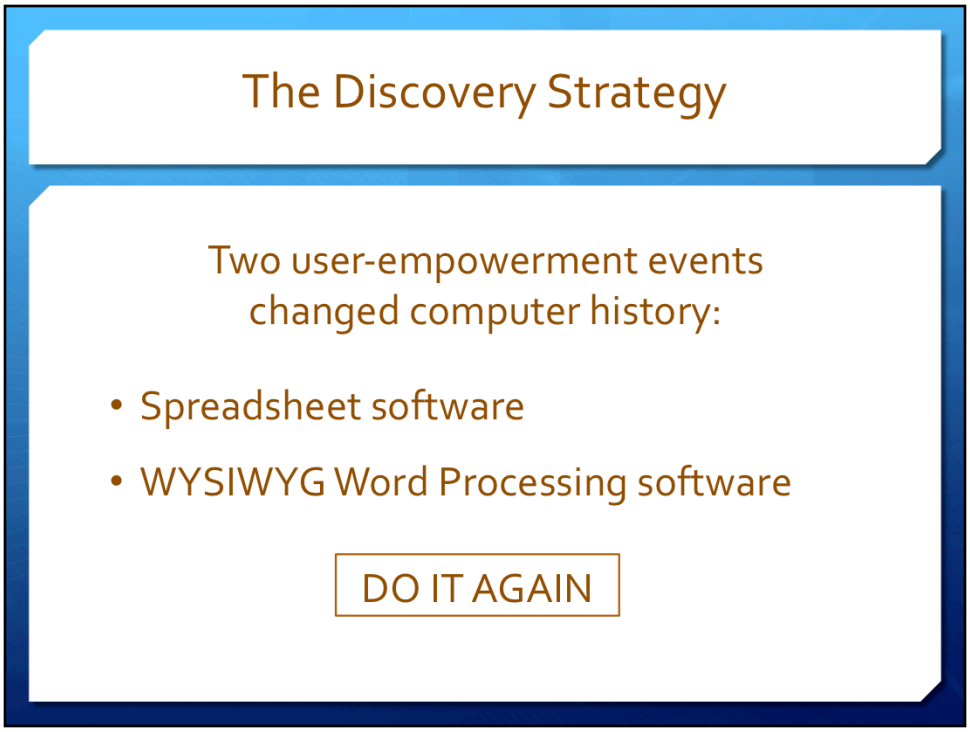


I'd like to use one slide to state the beliefs that are behind this research.

The embedded microcomputer has changed how almost everything is built. I remember my epiphany after studying an automobile service manual: what I had once thought was a car is now really a bunch of car-shaped computer peripherals programmed to behave the way we expect a car to behave. And of course what we have been calling a phone is now an application platform.

Our civilization has three important technologies that have moved from the exclusive provinces of educated elites to universal understanding: the calendar (originally in agriculture), arithmetic (originally in commerce), and writing (originally in commerce and law). Once only priests could use them; now they are taught in elementary school. In the journey of these three technologies from exclusiveness to universality, they had to be reinvented multiple times in order to make them more widely understood. Also, strategies for teaching them had to be invented.

Because software has become central to so many things around us, I believe that the understanding of software must join this same progression from exclusivity to universality. **Behind this research is the belief that the humaneness of a technology (as I'll define it) is directly related to its potential for universal understanding.**



Going back to the original problem, how do I make application building more accessible, I looked for historical instances where a great number of computer-naïve people naturally gravitated to using particular computer applications. The idea was to learn how these applications succeeded with their new audiences and then repeat the lesson, this time with software construction.

History gives us two obvious examples: the spreadsheet and the What-You-See-Is-What-You-Get word processor.

The introduction of the VisiCalc spreadsheet on the Apple II in 1979 changed the personal computer from a hobby toy to a business machine. Business people began circumventing established IT acquisition processes, buying Apples on their expense accounts and taking them to work so they could use VisiCalc. This subversive infiltration into the office is one big reason behind IBM's development of the IBM PC. After the PC came out Lotus put 1-2-3 on it and, in the words of Wikipedia, Lotus 1-2-3 became the "PC's first 'killer application'; its huge popularity in the mid-1980s contributed significantly to the success of the IBM PC in the corporate environment."

Another widely adopted application has been Microsoft Word, especially later versions that exactly duplicate on paper the page image the user builds on the computer screen.

It's generally understood now that what the spreadsheet and word processor do that is so powerful is: **they get out of the way**. The premise is that the user is an artisan who is familiar with the conceptual model of the object he is building, and the software tool attempts to be transparent, giving the illusion that the user is working directly on this object. My initial computer-science problem then became: how can we do that for building software?

That was the starting point. Since then I have generalized the lessons, and now I'll show you what I have learned.

Here's the table of contents. Going back to the caveman-making-a-hand-axe vision, the overriding construction paradigm is that of an artisan, manipulating his working material with his hand tools. The artisan model pervades this synthesis, and it will be the basis of the definition of a humane tool in part 1.

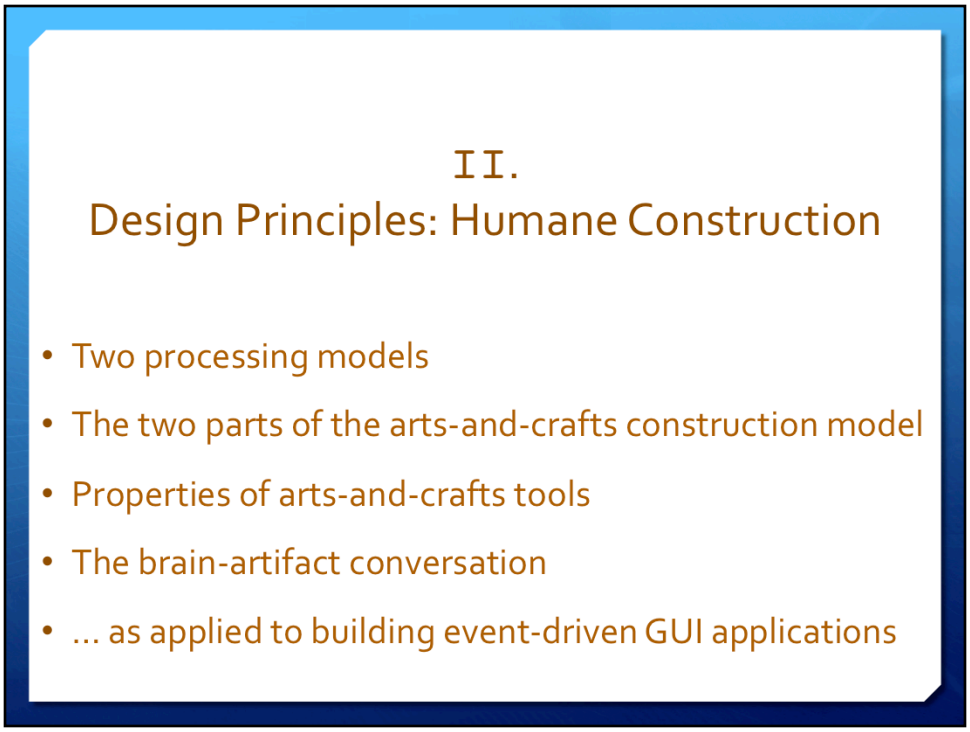
In part 2 I'll describe two major processing paradigms, the first being the input-process-output model we inherited from serial file processing and that is still with us in the Unix shell languages. This worked well for thinking about punched cards and magnetic tape data processing applications, but when the world switched to interactive event-driven applications it was no longer helpful. The second model, which I'm calling "arts and crafts," is based on the artisan paradigm and has two tightly connected parts: the tool and the working material. In part 2 I'll describe several attributes of a humane construction process.

Part 2 applies to construction in general. In part 3 I'll restrict the consideration to software application development. Here I'll present a hypothesis that each application can be factored into two parts, a dynamic part that is fixed across a set of applications, and a static part that is specific to each application in the set. I'll assert that good algorithm languages and good application languages are totally different animals, and the criteria for effective algorithm languages are quite different from the criteria for effective application languages. This is relevant because in the artisan-centered application creation model, the application language describes the working material.

A tool is humane to the extent that it rewards its user for using the hand-eye-brain system each of us has worked so hard to develop. A tool is not humane to the extent that it forces its user to employ other, less well developed, faculties.

If you study the massive effort it takes for a child to learn to feed him- or herself with a spoon or to build a pile of blocks, you come to appreciate all the internal machinery that is being built by that effort and what a large investment of human capital is being made in the hand-eye-brain system.

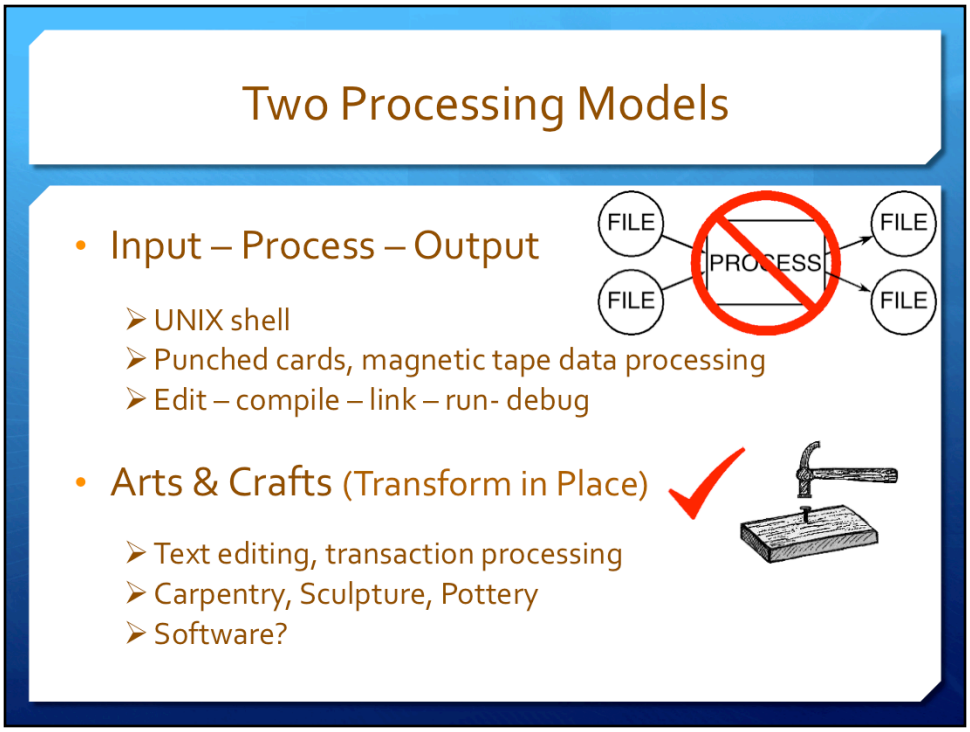
My computer-science problem then becomes: how do we employ this human investment for software construction? This anthropological way of thinking is going to lead us to some specific design criteria about how to build software applications. After developing these criteria I'll show you a construction session that puts them into practice.



In this section I'm going to apply the definition of humaneness to building things in general. There are five slides in this section.

In the first slide I'll state that the input-process-output processing model we inherited from serial file processing is not what we need, and I'll present the alternative, which I call "transform in place" or "arts and crafts." Then, given this processing model I'll present the two parts of the arts-and-crafts construction model: the tool and the working material, and I'll state what the two parts are in the case of application construction.

In "properties of arts-and-crafts tools" I enumerate specific characteristics that enable the comfortable interaction between the artisan and his working material. I call this comfortable interaction the "brain-artifact conversation." The remaining slides in this section elaborate on this.

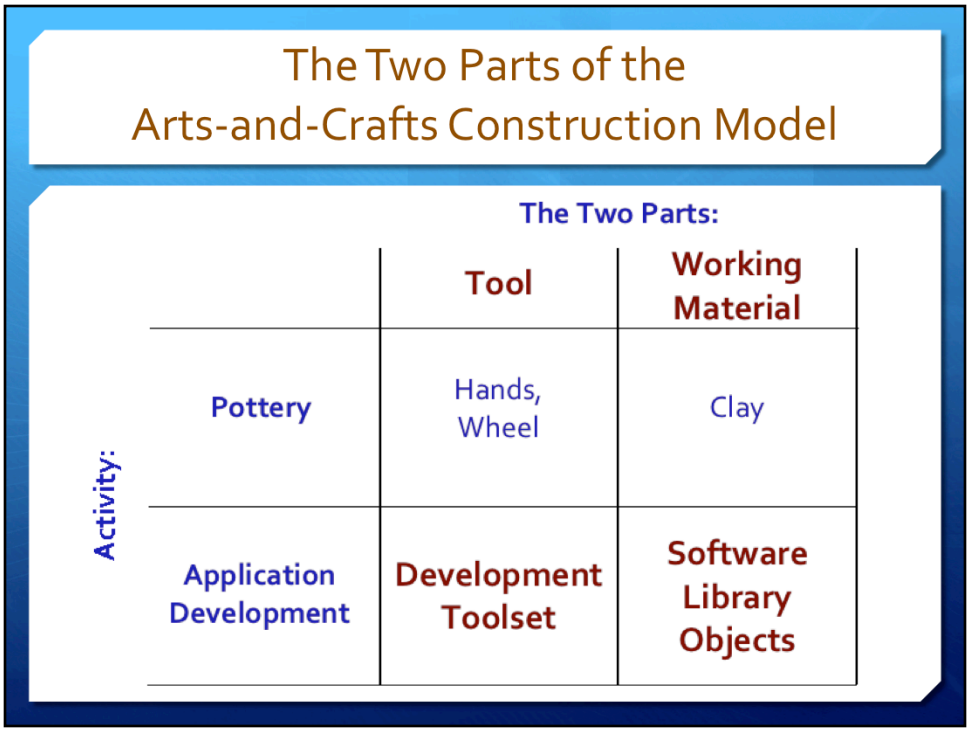


When I started programming, commercial data processing was a matter of periodically updating sequential files. At that time the common storage medium, trays of punched cards, was giving way in large applications to magnetic tape. The moving-head disc, which IBM made into a commercial reality, made transaction processing feasible, but that was several years later.

When I was starting, the dominant processing model was input-process-output, shown at the top. The two-in, two-out model shown here could do utility billing. One input file was the customer master file, the other contained the meter readings. Both were sorted by customer ID. The output files were the updated customer master and the printable utility bills. It was a very useful model, but it's hard to imagine how you can usefully describe interactive transaction processing using this model.

Notice that the traditional method of preparing computer code is an example of input-process-output.

The second approach I'll call arts and crafts. **In the arts-and-crafts model you start with the working material in place and you step-by-step transform it into its final form.** This is the processing model of the artisan and the one I'm going to use from now on. What does building software look like from this point of view? One thing we might guess is that the source code-object code distinction goes out the window; that's a holdover from the input-process-output model.



The two parts of the arts-and-crafts construction model are the tool and the working material. Think of the potter at her wheel, the cabinetmaker turning a table leg, a carpenter framing a house, a cook peeling potatoes, or a neurosurgeon doing brain surgery. In each case the toolset is what the artisan carries from job to job, and the transformed working material is what he leaves behind at the end of a job.

In the case of application development, the artisan is the software developer and the working material usually consists of instances of software library classes glued together by some code written by the artisan-developer.

Now we get into the nuts and bolts of what tools that support the arts-and-crafts paradigm have to be good at. These are ideals, of course, but they provide a useful set of design principles.

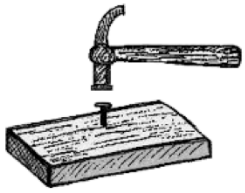
Transparency means successfully conveying the illusion that the artisan is working on the material directly with his hands.

What-You-See-Is-What-You-Get means you don't have to think at all about any distinction between source and target forms of the working material. (That went away with input-process-output.) Well-executed scripting languages are good at this.

Now, this thing I have named the "brain-artifact conversation" describes what happens in an ideal arts-and-crafts construction session. In the next slide I'll use the hammer-nail example for explaining the conversation.

The Brain-Artifact Conversation

- Unity
- Continuity
- Immediacy
- Interactivity
- Reversibility



In the ideal, I visualize a dance that occurs between the artisan and the working material that is barely conscious, in the manner of a master potter at her wheel. This dance has these properties:

Unity. There is only one thing being worked on, and it's *both* the input and the output. Nailing two boards together means transforming, in a sequence of steps, two boards and a nail into two boards nailed together.

Continuity. Small actions produce small changes. Hit the nail harder and it will probably move more. This characteristic helps to tell you how hard and in what direction to hit and when to stop. (Of course software is nonlinear and discontinuous, but in the small this is a valuable guideline.)

Immediacy. When you swing the hammer, the nail moves and the eye-brain immediately understands the new state of the working material.

Interactivity. Continual tactical planning is part of the dance. How you hit the nail this time depends on what it did the last time you hit it. The overall construction strategy might be in place but the detailed sequence of steps required to do the whole job is not predetermined; each step helps to determine the next.

Reversibility. The hammer comes with a nail puller.

The Brain-Artifact Conversation Applied to

Building Event-driven GUI Applications

- Unity
  - What you see is what executes (WYSIWE)
  - Tool and application are side-by-side peers
  - Tool and application are always "executing"

- Continuity
  - Small app changes ➔ Understandable behavior changes
- Immediacy
  - Modify app ➔ New steady state ➔ New results
- Interactivity
  - Iterative development, experimentation
- Reversibility
  - Remember the "undo"

This is what the brain-artifact conversation looks like in application construction.

Unity. WYSIWE means that what you are building remains on the screen. Putting a translator between what you see and what actually executes would destroy transparency. Note that both the tool and the application are event-driven GUI applications running side-by-side, and either one can take the next user event. You'll see this more concretely during the demo.

Continuity. If you make small changes to your application you will probably understand at every step whether you have made progress.

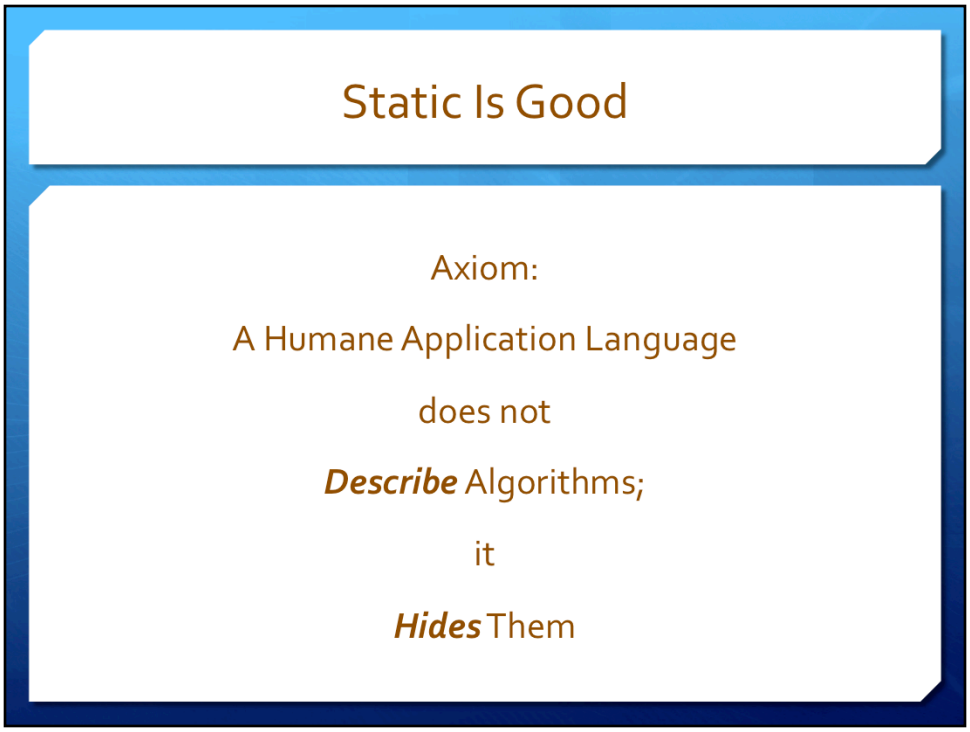
Immediacy. The consequences of a change must show immediately or else you have broken the spell of the dance.

Interactivity. **This is not software engineering. It's the way real people build things they want to use, going all the way back to the hand-axe.**

Reversibility. Nothing new here.

These two principles come from way back in my personal history when I was trying to understand the transition from serial file processing to transaction processing. So I'm not going to spend a lot of time justifying them; I'll present them as axiomatic. I have written elsewhere about the discovery process.*

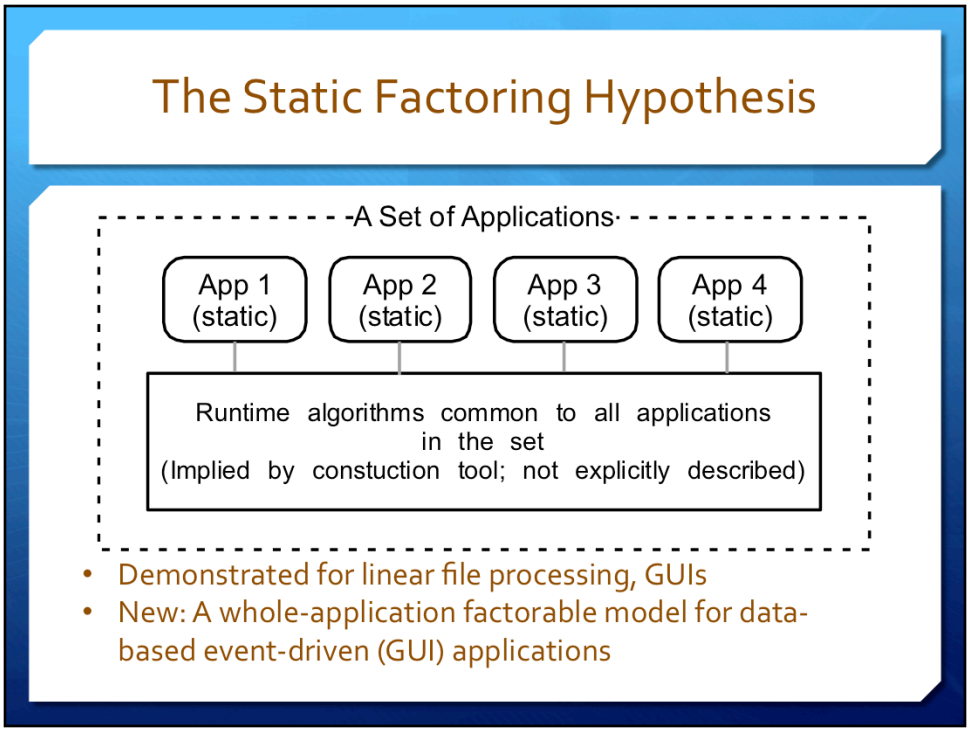
*http://heed.melconway.com/ at the "History" tab.



This axiom is worth reading carefully so I'll pause for a bit.

It is the basis of my assertion that application languages and algorithm languages are different animals.

Initially I based the idea that static is good on the evidence that a lot of people are not very good at devising sequential strategies. But then the static factoring hypothesis came along and I saw that Static is Good and the Static Factoring Hypothesis are two sides of the same coin.

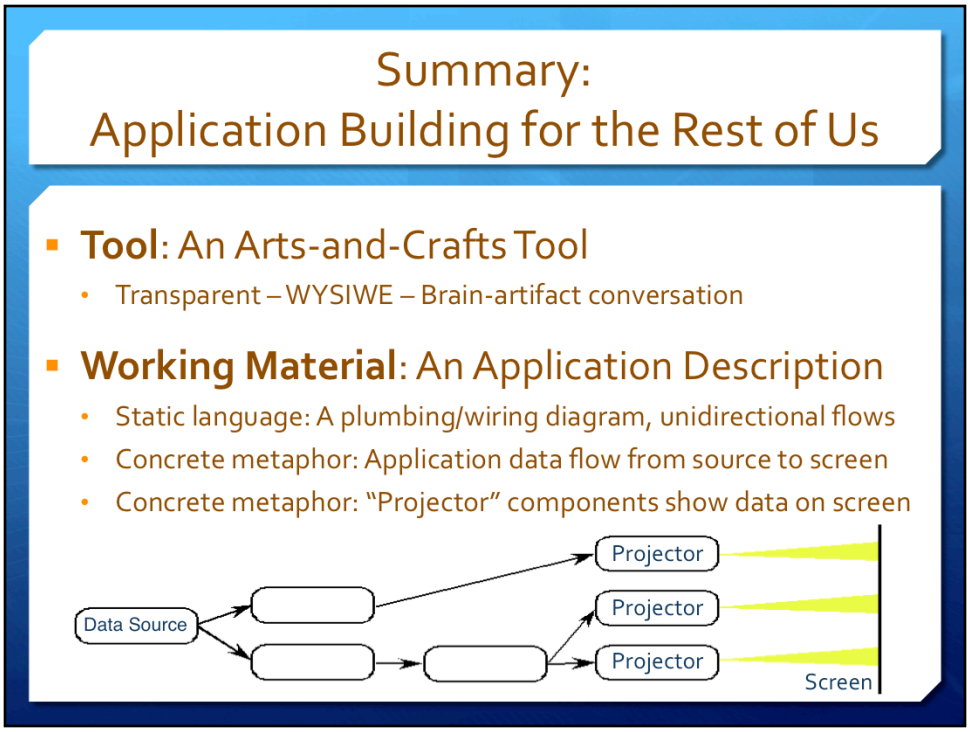


The static factoring hypothesis says that **the universe of applications can be partitioned into disjoint sets according to common underlying algorithms that are fixed within each set of applications**. Then the specification of a particular application within each set is a static parameterization of these algorithms.

Of course the hypothesis is trivial if each set contains only one application. But the first such set I encountered that fits the hypothesis is the whole set of serial file processing applications, first with punched-card files and then with magnetic tape files. In punched-card processing a processing run consists of running decks of cards through about a half-dozen specialized-function machines, such as sorter, collator, reproducing punch, and a totalizer/printer. Each machine is programmed with a wiring panel, but it turns out that almost all of the wires just format data, and the underlying algorithm is built into the machine. So many batch-processing business applications fit the hypothesis. The software industry recognizes the hypothesis because it has created tools called “report generators” with mostly static specification languages.

The user-interface builder in Visual Basic also illustrates the hypothesis. In this case the underlying algorithm is an event dispatcher and the static description is one or more pictures of windows and dialog boxes.

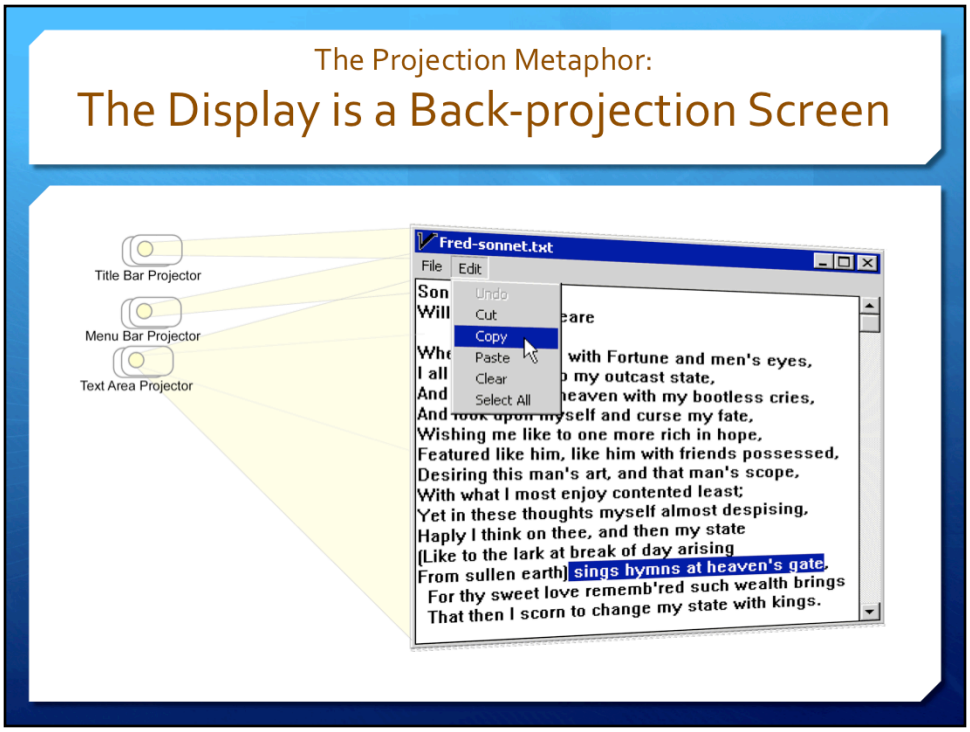
What I **wasn’t** able to find was a a way to fit whole event-driven GUI applications, from database to user interface, into the hypothesis. That’s what I’ll now introduce.



Here is a wrap-up of what I have said so far. You are an artisan building an application using the "transform-in-place" or "arts-and-crafts" construction model. This model has two parts: the tool and the working material (that is, the stuff of your application). The tool has some special properties: it supports the illusion that you are manually manipulating your working material, what you are building on the screen is what executes, and it supports the dance I call the brain-artifact conversation.

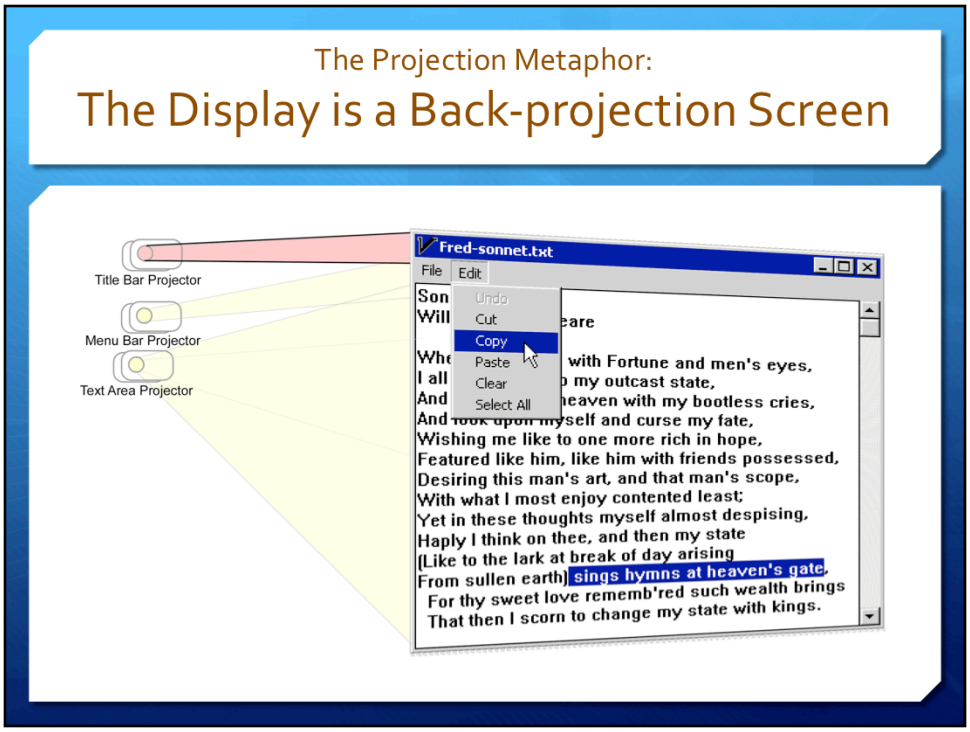
Now to the working material. I've already made the case that it is a static description; in this case it's a plumbing or wiring diagram. We've all played with dataflow models; this is one of those. It's shown schematically at the bottom of the slide. Data flows through components from left to right, with the files or databases at the left and the user-interface screen at the right. At the right end of the plumbing network are "projector" components that show the data on the screen.

Before I show you a couple of slides on projectors, let me mention event handling. The biggest problem with the dataflow model is how to handle user-interface events. The temptation is to handle an event at the user interface with a right-to-left event flow. This approach leads to nothing but trouble. In the present model, some projectors (such as those that project buttons) can take events from the user. There are no right-to-left "event flows." Event handling is not an add-on but is an intrinsic property of certain components and works well within the unidirectional left-to-right model.

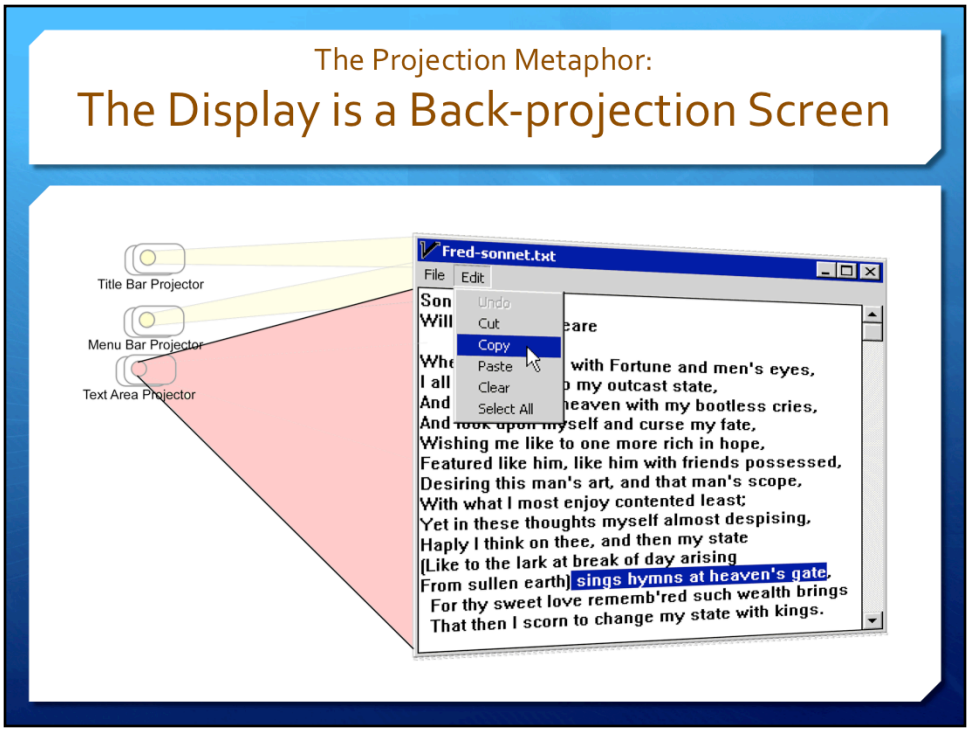


Here is a screen shot of a simple text editor I built with the prototype tool.

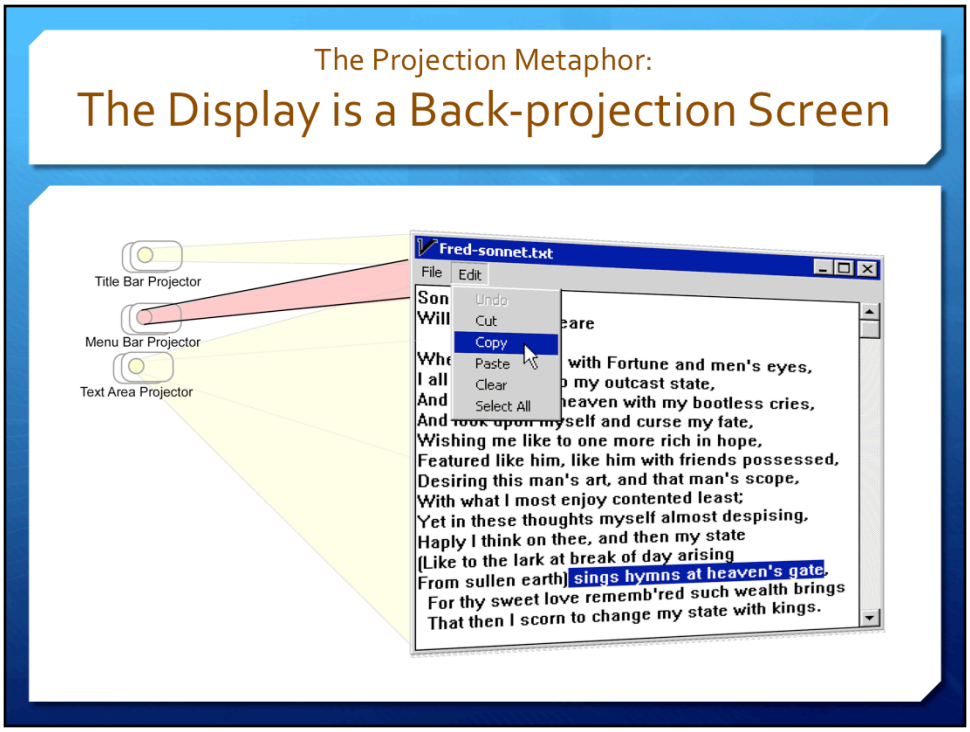
This window has three projectors. There is a projector for the title bar, another for the menu bar, and one for what shows up in the main part of the window, in this case a single rectangular area that has text-editing behavior. The three projectors are terminal components at the right end of the plumbing network for this application. Flowing into the sink connector in the back of each projector is some sort of application data object. The question is: what gets fed into the back of each projector?



The title bar projector is simple: a text object flows into its back end. The source of this text object can be anything that sources text objects. In this case it's a concatenation of a constant ("Fred") and a variable (a file name).

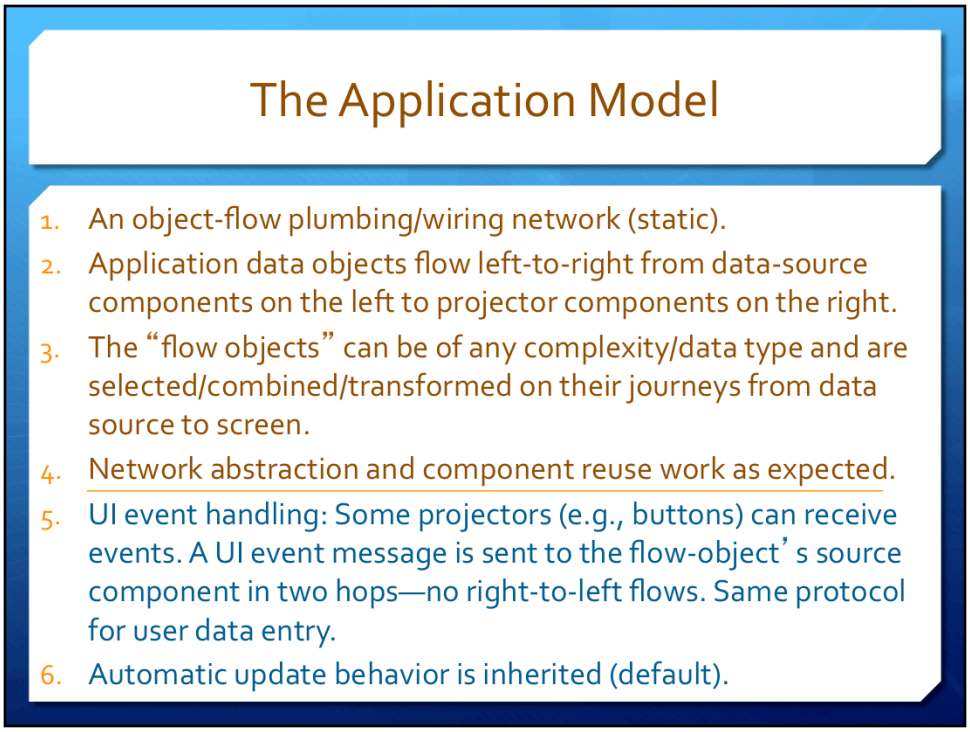


The main area of the window also displays a text object, with some additional features such as the blue selection. In this case the projector component embodies the text editing functions and acts on the drag event that defines the selection. In other cases the component that acts on a user-interface event can be deeper inside the plumbing network. That is, projectors take user-interface events, but they don't always act on them. Sometimes a projector just tells the object being projected that it has received an event, and this object sends a message to the component from which it originated. This is what happens in the case of the "copy" command object being projected by the menu.



So what comes into the back of a menu-bar projector? In this application model, a menu bar is a collection of menus, each of which is a collection of command objects, such as the “copy” command object. A command object phones home when its projector tells it that it has received an event. This phone-home message eliminates any need for right-to-left event flows, which don’t exist in this application model.

Incidentally, the phone-home message is really a sequence of two messages: projector to flow object, and flow object to the component that originated it. This two-step sequence is true for all inter-component communication. It effectively decouples components, which facilitates the practical matter of component reuse.

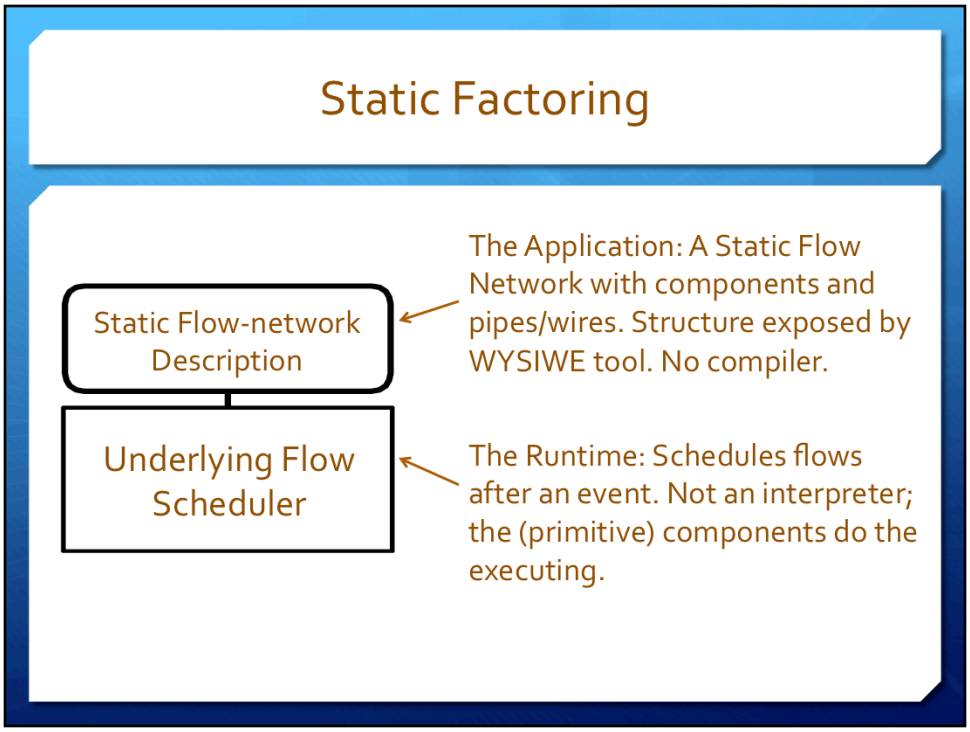


This summarizes what I have said about the application model. One: It is a static left-to-right object-flow model. Two: If a flow object flows into a projector it shows up in the display region handled by the projector.

Three: Flow objects can be instances of any data type. A lot of what the components do in a business application is manipulate collections: parsing, combining, and selecting.

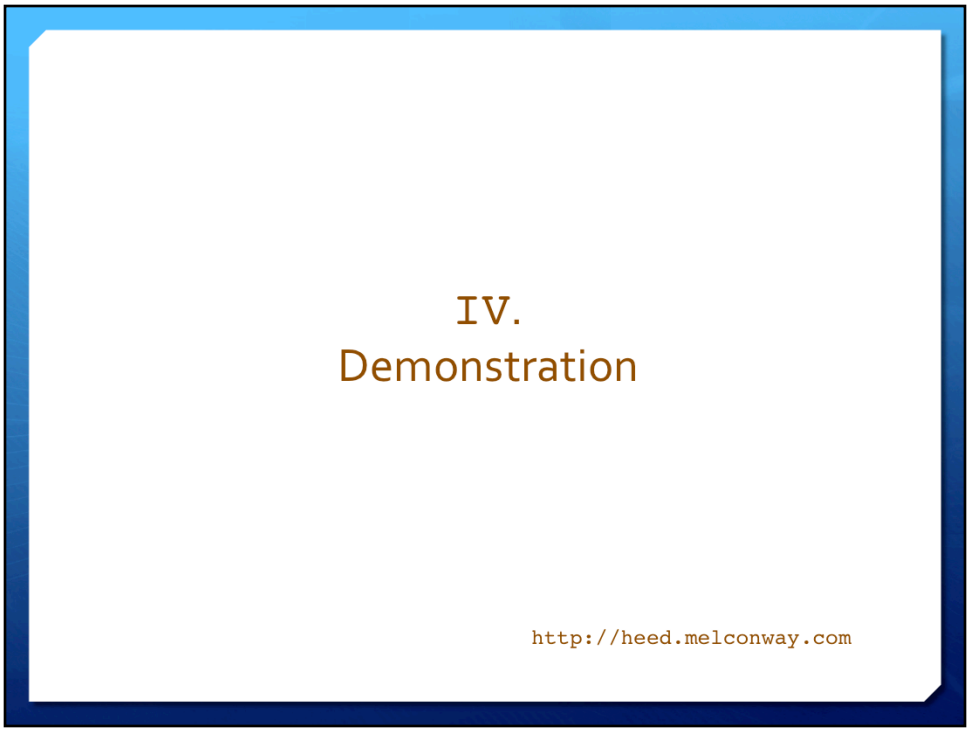
Item four says that the model scales. You can turn a flow network into a component that you can reuse without knowing whether what's inside is code or plumbing. There are two kids of components: primitive components and composite components. Primitive components have code inside and composite components have plumbing inside. From the outside they are indistinguishable. By the way, this works if what flows are application data objects; it doesn't scale if what flows are messages, as in some early flow models.

The bottom two items are more about implementation. There is a protocol governing flows that all components inherit. When something comes out of a source connector at the right of a component, it will arrive at the sink connectors of all connected components, and those components will be notified of the arrival. Each component, in turn, guarantees that its outputs will be correct, given its inputs. As a result, any event that causes a data value to change will lead to a sequence of flows that eventually stabilize into a new equilibrium. This behavior determines the common underlying algorithm for this application model.

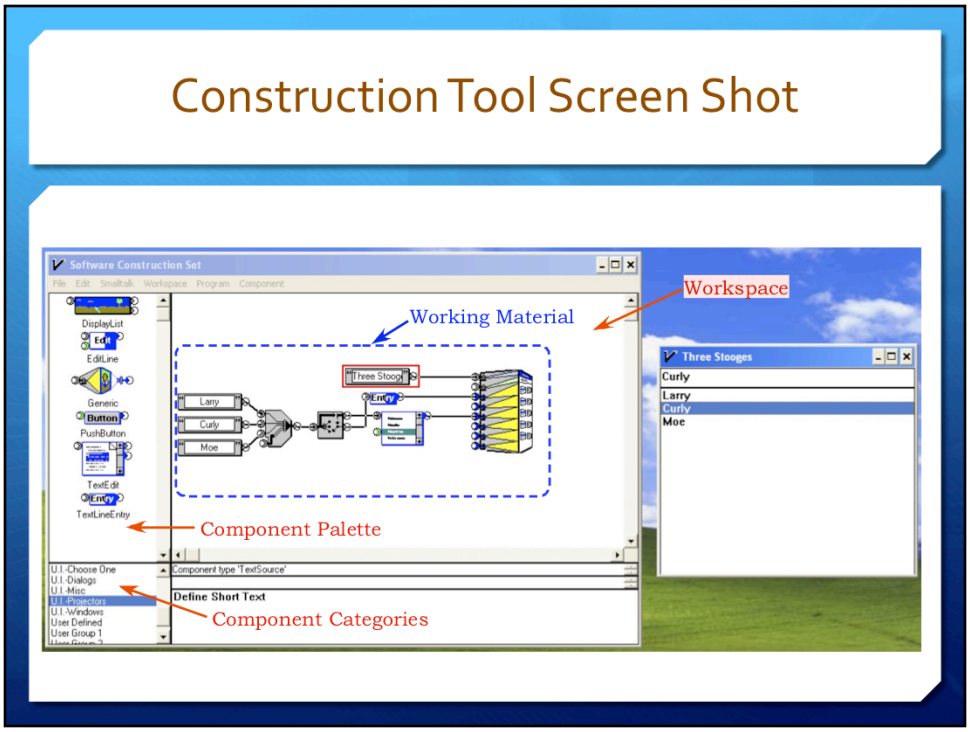


This slide states that the application model is an existence proof of the static factoring hypothesis. The common runtime contains the scheduling algorithm that causes the network to stabilize into a new equilibrium state after any change caused by receipt of an event.

As far as I'm able to tell, this model works for all event-driven applications with graphical user interfaces. That is, the static factoring hypothesis holds for the entire class of such applications.



Now I'll show you a realization of these design principles. But before I switch over to the construction tool I'll show you one slide that will help you with what you are going to see.



The description of this simple application is the network drawing inside the dashed blue rectangle; I drew this by dragging the components out onto the workspace from the component palette at the left, and dragging wires between the connectors. This is the working material of the arts-and-crafts construction model.

The “Three Stooges” window at the right is the behavior of the application. The window frame is projected by the large component at the right of the workspace. Of the three inputs to this component, the top is the text for the title bar and the other two are the projectors for the text line and the list box.

The square component to the left of the list box projector makes a selection from the collection that comes into its sink connector at its left. (The icon art is meant to suggest a rotary switch.) The collection that comes in at the left is routed up to the top source connector and on to the list box projector, which displays it. There is a tight coupling between this selector component and the list box projector so that whatever the user selects in the list box comes out the lower source connector of the selector, which then goes on to the text line projector. The collector component to the left of the selector wraps its three inputs into a single collection object.

Please keep in mind that what I’ll be showing you is secondarily a presentation of an application language, and is primarily an illustration of how the design principles for humanistic construction can be realized. So now I’ll bring back the slide in which these principles are summarized.

The Brain-Artifact Conversation Applied to

Building Event-driven GUI Applications

- Unity
  1. What you see is what executes (WYSIWE)
  2. Tool and application are side-by-side peers
  3. Tool and application are always "executing"

- Continuity
  4. Small app changes ➔ Understandable behavior changes
- Immediacy
  5. Modify app ➔ New steady state ➔ New results
- Interactivity
  6. Iterative development, experimentation
- Reversibility
  7. Remember the "undo"

Here is the mindset I'd like you to have when you are watching this. You are watching an artisan, say a potter at a wheel, working with **a plastic medium, pushing it around into different forms.**