In Search of the Modeless Workflow

This paper applies to the development of event-driven GUI applications. It synthesizes (1)the lessons I have learned about simplicity of application conceptual models and (2)the user-interface design concept of the mode into the process concept of the Modeless Workflow. The centerpiece of the paper is the list on page 8 of the twelve attributes of a modeless workflow.

These Ideas Come From Somewhere Else

My research started out as an inquiry into simplicity in *understanding the internals of interactive applications*. It evolved through the following ideas.

1. If an understanding of application internals is to be significantly simplified, let us think in terms of making this understanding universally accessible. Whether such a leap is a prediction or simply a research tactic doesn't matter; it's an assumption.
2. Given this assumption there is an analogy to the history of arithmetic, calendar, and writing, which have evolved from the property of a priest class to being taught in primary school. Using that analogy we can think of primary-school children as proxies for the population as a whole.
3. We can draw on the theory of childhood education, particularly with respect to teaching number sense and simple arithmetic. We have learned from pioneers such as Montessori[1,2] and Cuisenaire[3] that, for children, learning and manipulation are inseparable.
4. This lesson helped me reframe the research goal to this question: what is the software equivalent of hands-on learning? Addresing this question led to the definition of a *hands-on software development tool*[4].

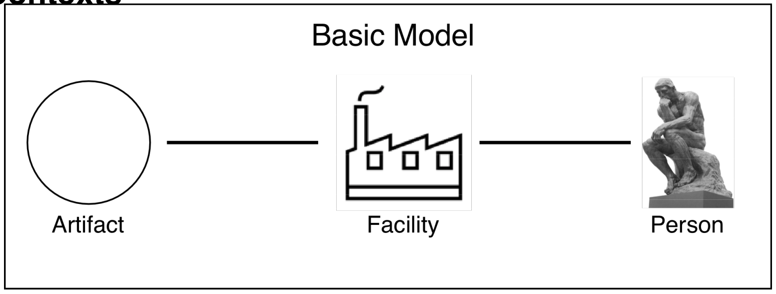
1 https://en.wikipedia.org/wiki/Montessori_education
2 *"The hands are the instruments of man's intelligence."* http://ageofmontessori.org/the-birth-of-a-mathematical-mind/
3 https://en.wikipedia.org/wiki/Cuisenaire_rods
4 http://melconway.com/Home/pdf/simplify.pdf page 12.

In this paper I'll examine how far these lessons about simplification of *conceptual models* can be applied to simplification of *application development*. I do not abandon any of the earlier lessons, in particular I keep the flow or wiring diagram as a conceptual model of an application. The approach is to combine two concepts, modeless user interfaces and hands-on tools, into the combined concept of the *Modeless Workflow*.

This paper incorporates this thesis: many existing ideas about simplification of development can be subsumed under the concept of Modeless Workflow. Having a single model for development simplicity could be useful.

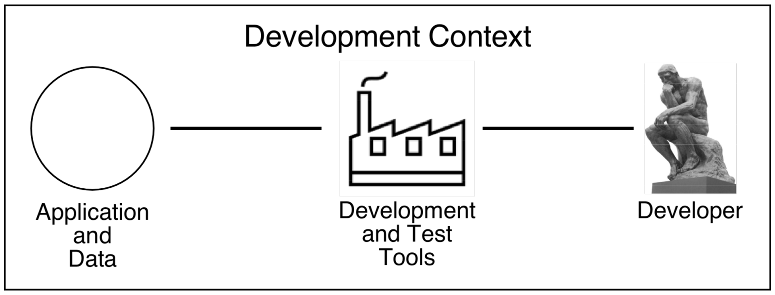
Development and Deployment Contexts

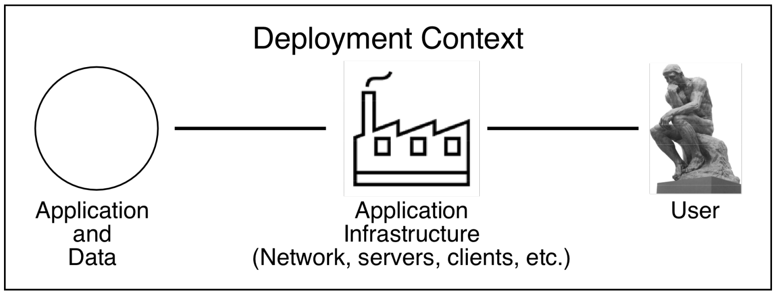
In order to talk about these ideas we need a model. Here is one.



This is a model of a person interacting with a software application and its data, which are here collectively called the *artifact*. The *facility* is the technology the person uses to interact with the artifact. The application's user interface is in the artifact.

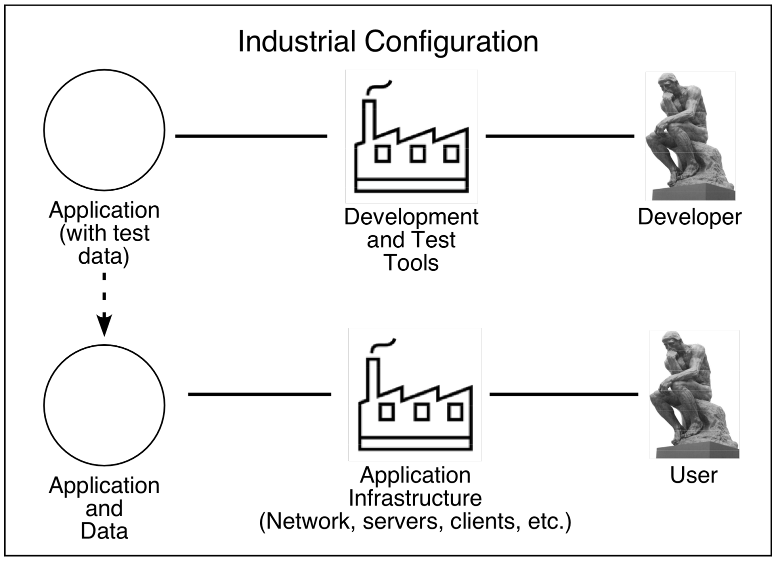
This model can be useful in two contexts: the *development context* and the *deployment context*.




Deployment Context
Application and Data
Application Infrastructure (Network, servers, clients, etc.)
User

In both of these pictures the artifacts are almost the same. That is, on occasions (whose frequency is dictated by development policy) a stabilized application is moved from the development context to the deployment context for production. We can call this the *industrial configuration*, and can represent it as follows:

Industrial and Educational Configurations

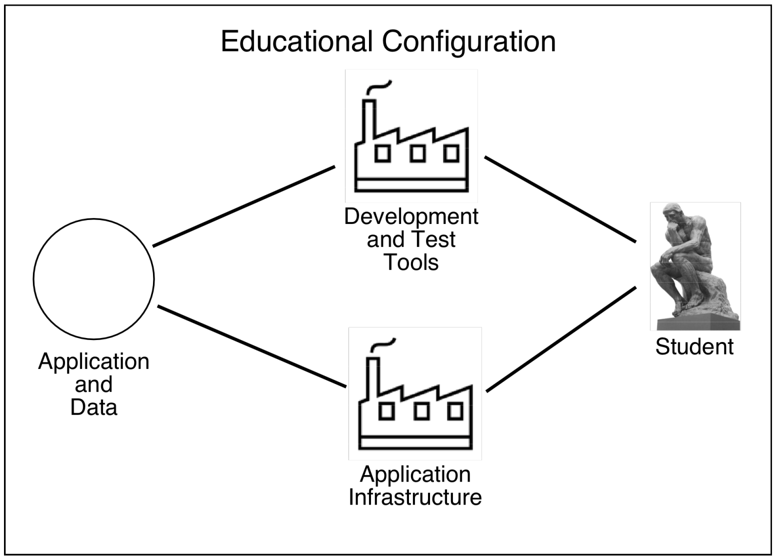

Industrial Configuration
Application (with test data)
Development and Test Tools
Developer
Application and Data
Application Infrastructure (Network, servers, clients, etc.)
User

In the industrial configuration the developers and users are distinct groups of people who don't normally communicate with each other (except in development projects, in which the users are represented by proxies).

Developers and users operate with distinct rule sets. Users are bound by a controlled rule set determined by the application, and autonomy with respect to the system is discouraged; users can't change the application and they change the data only in ways strictly controlled by the application. Developers, on the other hand, value autonomy and need to learn and adapt in order to do their job, which is to change the application.

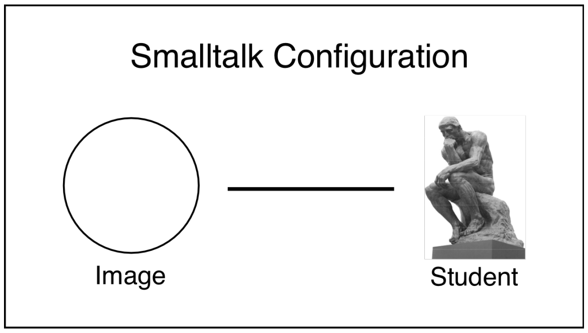
The development tools we are familiar with, compilers, linkers, IDEs, test tools, etc., are designed for the industrial configuration in which development and deployment are strictly segregated.

But there is another configuration we can call the *educational configuration*, in which the developers and users are the same people: the students.


Educational Configuration
Development and Test Tools
Application and Data
Student
Application Infrastructure

In the best instances of the educational configuration the development tools and the application infrastructure are tightly integrated, and moving between them is easy and fluid. Tools designed specifically for this kind of integration are Scratch[5] and Smalltalk; current examples of the latter are Squeak[6] and Pharo[7].

The original Smalltalk is unique in that everything is an object, and the initial libraries and development tools are made up of objects in classes that are all in a single integrated structure called the *image*. Construction of the application is an extension of the image accomplished by adding classes and methods. The development tools and the application are made of the same stuff, and might even share parts.



In the language of the basic model, the facility and the artifact in the Smalltalk configuration are inseparable. It is this tight integration of the application and the development toolset that makes Smalltalk uniquely desirable for education and for programming in general. This integration makes Smalltalk less suitable for deployment into a disciplined (and possibly resource-constrained) industrial situation.[8] This inseparability also presents security risks, because the tools for changing the application are present in the deployment.

The Best of Both Worlds?

This perspective on the difference between the industrial configuration and the education configuration (and, in the extreme, the Smalltalk configuration) raises the question: how much of the fluidity and simplicity that are so highly valued in the simpler configurations can be incorporated into

5 https://scratch.mit.edu/

6 http://squeak.org/

7 http://pharo.org/

8 There are other views on why Smalltalk has not succeeded in industrial applications; see for example RailsConf 09: Robert Martin, "What Killed Smalltalk Could Kill Ruby, Too." https://www.youtube.com/watch?v=YX3iRjKj7C0

business application development? The present approach to this question hinges on the concept of *mode*, inherited from user interface design.

What is a Mode?

A mode is a temporary and restrictive context that constrains the user into behaviors (1)that must follow rules specific to that context, (2)whose outcomes depend on the context, and (3)whose outcomes might be different in another context. A mode persists until it is exited by a user action, for example clicking the OK button of a modal dialog box. The classic example is the computer keyboard's caps-lock key, but there are many computer applications, in particular early text editors and interactive applications (many still in use) whose user interfaces consist of a keyboard and a full-screen character display, that are full of modes.

> Larry Testler put forth the concept of modeless editing[9] as part of the development of the family of concepts of modeless user interface at Xerox PARC[10,11] and Stanford Research Institute[12,13] in the 1960s and 1970s. Since that time modeless application user-interface design has been encouraged by the design guidelines of Apple[14] and Microsoft[15].

Here are two problems with modes that show up in both the development and deployment contexts.

- It has been shown that the existence of modes leads to an increase in operator errors[16], possibly because of increased cognitive load. It's also reasonable to expect that modal systems are harder to learn.
- Modes limit the behaviors of the user to those anticipated by the designer and thus reduce the power of the tool for augmenting creativity. As an example, the clipboard, first introduced into modeless text editors for cutting and pasting

9 http://delivery.acm.org/10.1145/2220000/2212896/p70-tesler.pdf
10 https://en.wikipedia.org/wiki/PARC_(company)
11 http://worrydream.com/EarlyHistoryOfSmalltalk/
12 https://en.wikipedia.org/wiki/SRI_International
13 http://www.dougengelbart.org/
14 https://developer.apple.com/library/content/documentation/UserExperience/Conceptual/OSXHIGuidelines/
15 https://msdn.microsoft.com/en-us/library/windows/desktop/ff728831(v=vs.85).aspx
16 https://en.wikipedia.org/wiki/Mode_(computer_interface)#Mode_errors

text clips, has been generalized to include many other data types in many other types of applications.

Modes are not always bad; they can be useful to constrain the user's choices in small, specific contexts. A common example is a modal dialog box for answering a question, which must be either answered or abandoned before the user can return to the rest of the application.

The following twelve attributes, derived from the definition of a hands-on tool, characterize a Modeless Workflow for developing event-driven GUI applications. Some of these attributes imply design requirements for the application conceptual model; some imply design requirements for the tool; some imply both.[17]

(The terms *artisan*, *artifact*, and *working material* are motivated by the potter-at-the-wheel metaphor for hands-on development. There is no distinct boundary between the working material and the artifact. the working material is continually evolving into the artifact by stages of in-place transformation. Just as software development can be unending, this evolution can be unending.)

[17] The fact that the designs of languages and tools are coupled by the Modeless concept needs to be better understood.

The Attributes of a Modeless Workflow

1. **Unified**. *The artisan is not asked to alternate attention between "input" and "output" conceptual domains of the software being built; the thing being manipulated and the product are in the same conceptual domain*. That is, there is no "source/object" duality. A corollary necessary in order to simplify the overall process and eliminate debugger glitches: *The underlying application being built is isomorphic to what is in the artisan's hands*.
2. **Symmetrical**. *"Build" and "Run" are modeless*. The tool and the application being built are peers. The artisan's next move can be on the user interface of either one or the other.
3. **Alive with actual data**. *The artisan is not asked to alternate attention between building and testing*. The working material exists with real data present; the effect on the appearance to the user of a change to either working material or domain data is seen or can be examined immediately.
4. **Syntactically undemanding**. *The artisan is shown enough information to select among self-explanatory choices*. There is nowhere a requirement for text input according to a formal grammar.
5. **Immediate**. *Every modification the artisan makes to the working material is immediately seen in its behavior*. There is no perceptible delay introduced by a translation phase.
6. **Always on**. *During construction there is no concept of "starting the application"*. When a component instance is created in the workspace of the tool, it is already running, and it continues to behave according to its definition. (See 5 above.)
7. **Continuous**. *From one step to the next there is obvious continuity in the working material's behavior*. Small changes lead to predictable outcomes.
8. **Interactive**. *The result of each change helps to suggest the next change.* The artisan's brain is unconsciously engaged with the working material, like a child playing with a construction toy. (See 5 above.)
9. **Transparent**. *The tool supports the illusion that it is invisible and the artisan's hands are directly on the working material.* Metaphorically, the working material is embedded in the hand-eye-brain feedback loop.
10. **Inspectable**. *At any time all parts of the application can be inspected and the values so obtained can in turn be inspected*.
11. **Intervenable**. *The artisan can modify any part of the application* (provided that doing so does not contradict the definition of an existing component used in the application).
12. **Reversible**. *A good UNDO means no regrets*.

In attributes 1 and 3 the language “the artisan is not asked to alternate attention between….” is a signal that a mode is being avoided. The need to switch attention betrays a mode that requires that the artisan’s state of mind shift between user interfaces, languages, rule sets, or conceptual domains.

Here are the alternations being avoided in the first four attributes, cited by their numbers.

1. **Unified**: The alternation between the source language, used to write the application, and the target or execution language, used during debugging.
2. **Symmetrical**: The edit-build (compile,link)-run cycle. In the former the artisan is working with a text editor and other tools; in the latter the artisan is working with the user interface of the artifact.
3. **Alive with actual data**: The build-test cycle. This might differ from the above if the artisan is working with test tools.
4. **Syntactically undemanding**: Command line, shell, query, and scripting languages go here. Also, the artisan might need to specify individual parameters or function calls; this requires learning the syntax and constraints of each parameter or function call.

Ideally, in the absence of these (and possibly other) modes the workflow is fluid, and each step is dictated by the immediate (and partly unconscious) perceptions of the artisan rather than a progression of mental state changes from one frame of mind to another. This absence of friction is, in my observation, a common property of easy-to-use development environments.

The question is the extent to which the concept of Modeless Workflow can be fully applied to the development of business applications. This question can only be answered by experience. I have a prototype that demonstrates the first eleven of the twelve attributes which, when it is reimplemented, will be useful to test this question.