June 2022 Talk for Camp Smalltalk 50th Anniversary

Smalltalk research through the eyes of an educational mission

By Adele Goldberg

The critical idea behind the invention of Smalltalk was that it would be the software solution to a personal, accessible device called the Dynabook. This handheld device would be a new medium in which people could share both their understanding of how something works, and, where appropriate, be challenged as to whether that understanding approximated reality. The Dynabook would offer children a thinking partner, with the modeling capacity to test ideas and upend how we educate the next generations. The mission is not complete, but how far did we get? And is the mission still critical? In this presentation, I will attempt to answer these questions.

Good morning.

Thank you for the invitation to help celebrate 50 years of Smalltalk. As I have been out of the tech industry for some time now, enjoying a return to my roots in the art world, I was left pondering what I might discuss that would be of interest to what I presumed is a very techie audience. I doubted it would be appropriate to talk about my current interests, photography as art versus craft, and the appropriate use of post processing software. Instead, I decided to take this opportunity to revisit the LRG educational mission, specifically to see what I would want to do if I were back in that well-supported, research center again, and could start over—but with today's technology. The educational mission was not completed, so perhaps this talk can serve as a nudge to those of you interested in revisiting what technically still needs to be done.

The tasks I set myself for this morning then was

- First, to reconstruct the idea of a Dynabook as more than the hand-held computer that we are all familiar with today.
- And second, to revisit some of the critical themes and questions around those themes that still represent the invention challenges.

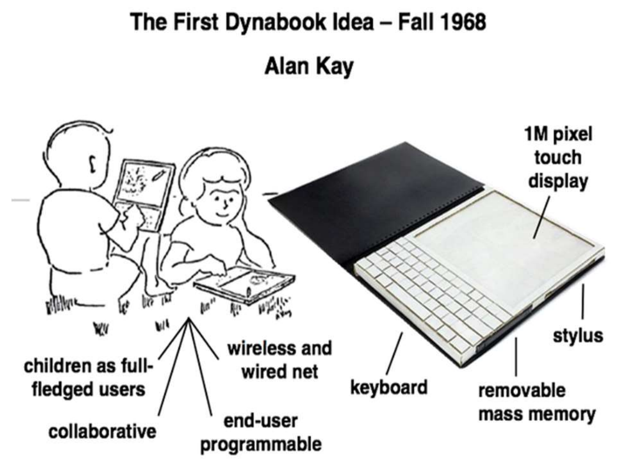
I extracted themes and research questions by giving you a reminder of six of the major research project explorations that built on the Smalltalk programming system. It is my assumption that, with the exception perhaps of the programming approach of the Simulation Kit, you already know the details of these projects.

As part of celebrating a 50-year history of Smalltalk development, there is another event to be held September 1 at the Computer History Museum in Mountain View, CA. The Museum's event will be a reunion both of the original Xerox Learning Research Group, and of the kids I delighted in teaching at a local Palo Alto Middle School.

Let's start with What is the Dynabook?

Whenever you look over the years of what was driving the Xerox Smalltalk research program, you can see the desire for technology to enable creativity, and you can see an educational mission.

The critical idea was that the Dynabook be the medium in which people can share their understanding of how things work, and be challenged as to whether that understanding reflects an approximation to reality.



I see the Dynabook, computationally, as a modeling environment. And, ultimately, I see it as a Thinking Partner with whom the user can rely on to find data or use cases that do or do not fit a proposed model. In other words, it is not purely a general-purpose programming system nor a communications device. Of course, as seen in this early sketch by Alan Kay, it has to provide resources for communication, collaboration, and visualization.

It should draw on the human senses for touching, and seeing and hearing for input and output as forms of communications, including or especially for self-reflection. Educationally, it should support the learner's ability to create models, working individually or collaboratively. And those models should be about both the physical and social worlds, both real and imaginary, to test the learner's thinking about those worlds.

Simulation and Construction-based Games

There are some interesting modeling examples today for educational use, although mostly they are presented as gaming applications. And there are multimedia applications written in or extendible by an object-oriented programming language. An interesting example is Blender 3D, which has a Python API. There are Internet sites that give access to multi-person gaming, and there are multi-person construction sites like SimCity, Minecraft, or Roblox. There are also purely educational sites, such as that supported by the Kahn Academy, where the tutoring was originally a collection of short video explanations, but is now organized by standard school curricula.

Is it time to burn the disk again?

The Dynabook educational mission is far from completed. Given the extent to which Alan's Children of All Ages are growing up thinking that technology—good technology—matters, our returning to the educational mission is important. For today's talk, I wanted to look at the Smalltalk-base project history to see what we learned in order to accomplish our educational mission, including to revisit whether the objects and messages paradigm is well suited to the task.

In the past, Alan Kay would call the activity that I set out to do as "burning the disk", starting over given what we have learned, with an eye on how both technology and society have changed. Of course, each time a new Smalltalk was released, we were in essence burning the disk. Since the mission did not change, neither did the name.

In the interest of checking out the latest disks before burning them, I explored various application-level projects that were done in LRG, in other Xerox groups, and elsewhere—projects that are not always discussed from the educational mission perspective when we look back at the Smalltalk history. Some of the LRG projects I will bring up are currently in Dan's Smalltalk Zoo—a website hosted by the Mtn View, California Computer History Museum, so you can get hands-on experience with them. I will try to show a couple of clips to, if nothing else, remind you of what graphics looked like back 40 and 30 years ago. I do assume that all of you know about this history, but please, treat my projects exploration as a way to show you how I determined this draft proposal of what a future Dynabook modeling environment needs.

Please also note that I am not going to present this projects history chronologically, but rather as how I went searching for a small set of themes and related research questions to use in evaluating future proposed Dynabook solutions.

The only surprise in what I am saying might be in the need for curriculum. *When the mission is to upend how kids are educated so that they can be intelligent participants in a growingly complex global community, then we need to have the content of such an education, not just the context and tools.*

There is no doubt that it is a major undertaking to create tested curriculum content for at least the first 9 years of formal schooling. But there is plenty of evidence that the difference in acceptance or popularity of one programming language over another is often the existence of curriculum. For professionals, students and teachers alike rely on books. For self-learning, we also rely on Internet help sites and online video tutorials. For kids in school, curriculum is a critical guide for their teachers.

As history reports, the Xerox Learning Research Group started its work by creating the Smalltalk-72 programming language and environment, and testing these by creating creativity tools. When I joined, my first step was to worry about how to teach the language, and to advise the team of what might make the teaching task easier, without losing the programming power of the language. Others helped with the hardware and communications parts, and, later, our group expanded and worked on collaboration tools. My background was in artificial intelligence and its use in education to create intelligent computer-based tutors. But, at Xerox, creating any such tutors was set aside.

What does it mean to say that the Dynabook is a Modelling Environment?

A modelling environment is more than a programming system. It is a medium in which to explore how things work, so there needs to be a library of things that work. It is a medium that challenges the user to consider whether the ways things work reflect a useful approximation to reality, which implies that it provides access to testing-data or use-cases. And it is a medium for creating models that can be accessed and viewed at different levels of sophistication, and that can be shared with human mentors or even a computer tutor. There is an expectation that the user can be guided in the use of the modelling environment by well-tested modeling-based curriculum, aligned with school standards.

Programming languages designed for education.

Back in 1972, there were a number of projects worldwide that were founded on the belief that computer programming afforded a new and useful way to think about teaching problem solving skills. Notably, the pedagogy emphasized an iterative approach to developing alternative solutions, rather than button-holing students into thinking there is only two results—a right one and a wrong one, or that you either get it right or wrong. In reality, what was being taught is that you write something, experiment with it, then search for and fix apparent bugs in the solution.

These projects set out to demonstrate how to teach programming, often by having the learners <u>do physical activities</u> that could be directly mapped to executable computer algorithms. Probably the most famous such projects were done with the language LOGO at BBN and MIT, better remembered as *Turtle Geometry*, and re-created in most other programming languages targeting young kids—including Smalltalk-72. Creating geometric designs as though a robot were moving forward or backward, or turning left or right, has morphed into creating simple games and animations and line drawings. Then there was Squeak, eToys, and Scratch in the Smalltalk family. I do think that Scratch was likely the more successful because of its focus on curriculum and partnering in schools everywhere, building Internet community and crowd-sourced support.

If you want to understand the level of detail it would take to create a multi-year curriculum based on doing physical activities that inform kids about possible models that they could program, take a look at the Squeak book from 2003 entitled *Powerful Ideas in the Classroom,* and written by B.J. Allen-Conn and Kim Rose. In it, they unfold how their students are to compare whether objects of different weight take the same or different time to fall to the earth from, say, a roof.

Thinking about this curriculum problem, and its complementary problem of teacher training, brings back memories of Seymour Papert jumping around on a table, pretending to be a turtle robot, in order to teach geometry. As most classroom teachers bulked at jumping around on the top of a table, the curriculum delivery approach definitely needed more universal appeal.

And of course, reaching back in history, when we think about the goal of upending how children learn, we should be reminded of Seymour Papert,'s 1980 book *MindStorms--Children, Computers, and Powerful Ideas*, an effort to shine the light on computers to be used for computation. Programming a computer was in contrast to having the computer turn the pages of online programmed instruction, or drill-and-practice, that was the rage in the late 60s and early 70s from such practitioners as Stanford University's Patrick Suppes, with whom I worked as a graduate student. Papert called his worlds *Microworlds*, but at the core, the ideas of identifying restricted libraries and visual user interfaces for introducing computer concepts is similar to some of what we tried with Smalltalk. Papert would say that in Microworlds the student is programming the computer, whereas in the computer-assisted instruction courses, the computer is programming the student. He would want us to have the students think about thinking, which is another way to talk about building models for understanding. And, of course, since he was a student of the psychologist Piaget, Papert's hypothesis was that the barrier that takes a child from concrete thinking to formal thinking is overcome by the ability to program, to making learning more active, more self-directed, and, I would add, more exploratory, as an approach to create concrete representations of formal explanations.

I do have to note at this point that Papert and I diverge on the importance of Curriculum. As a student of Piaget, Papert preferred the kind of learning that happens without deliberate teaching, although he did try to equate curriculum to the Piagetian idea of creating specific cultural surroundings that provide materials from which children learn. Thus, Turtle Geometry, defined as a microworld consisting of a particular set of objects and their behaviors, delivers a kind of curriculum. What we do agree on is that a well-selected set of interacting objects—what Papert called Microworlds and we called Kits—is likely a more engaging approach to the educational use of computer technology than drill-and-practice.

Although we did work with kids, at our lab and in their schools, and we did look to find ways to teach them how to program by creating interesting interacting objects, doing so was really informing us about the teachability of whatever was then the current version of Smalltalk. We were not researching whether students became better problem solvers or thinkers through their use of a computer programming language.

Our initial pedagogical ideas for teaching Smalltalk originated with our teaching Smalltalk-72. Specifically, we were asking whether making the elements of an executable software solution explicit in the form of object definitions and message-passing among objects, offers something special in the way of creating interactive models. The issue we ran into was that, in Smalltalk-72, the receiver of the message had control of how to parse the remaining message stream. This means that the only way learners could understand and talk to one another about a Smalltalk-72 method was to mentally execute the method associated with a message. Besides being very hard for beginners to understand, such a situation ran counter to the pedagogical idea that you learn to write by reading, and introduced what turned out to be unnecessary learning barriers. It certainly restricted the complexity of what could be taught.

Remember:

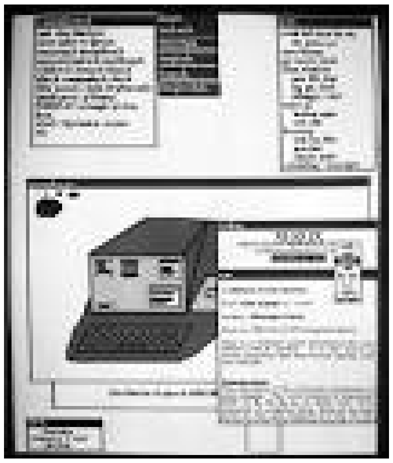
The context of a learning environment based on modeling is as much about understanding how an existing model works, as it is constructing a new model.

As a consequence, the ability to read an implementation <u>matters</u>.

This "read-to- write" approach to the use of Smalltalk was critical to our educational efforts and, most importantly, our ability to attract users who were not programmers by profession. Smalltalk as the modeling language evolved as we explored the pedagogical idea that the students start with models that they can read, and then refine those models or reuse them to create new ones.

The NoteTaker

Let me start looking at projects from the past with a story some of you might have already heard. In 1978, we worked with a group of hardware designers at Xerox, led by Doug Fairbairn, in order to build a portable…carry it with you…computer we called The NoteTaker.



By 1978, we had published Smalltalk-72, and we had released a major new language named Smalltalk-76. In 1978, Doug's team at Xerox built a portable physical device based on Intel 8086 processors, that included a mouse, a small display screen, and an Ethernet board. The resulting computer was indeed *luggable*, but fit under an airplane economy seat, and ran Smalltalk-78.

But why call it a "NoteTaker"? I take responsibility for what might seem an unlikely name. I suggested the following scenario: A student is asked to research how socializing among the servants in the household of the English and French royalty might have influenced alliances and disputes. Since it was 1978…16 years prior to the introduction of the Internet... this research challenge talked about going to a brick-and-mortar library and browsing the books in their collections. But the idea still applies in today's Internet age.

Generally, the educational plan was for the student to:

- First, formulate a hypothesis consisting of: with whom, with what, and where such socializing among the servants might have taken place.
- Then, gather data on significant historical events occurring in the targeted timeframe.
- Third, identify any way in which information flowing among the servants could have affected the instigation or outcome of these events.
- And finally, create a story that you can poke at, visualize, show to others, get their critique—i.e., learn by sharing a model and the data you used to formulate that model, where the model represents your understanding of what could have taken place.

The core focus was clearly to be on how the model could be represented and, using identified data, executed and displayed in an interactive format, with the ability to access the implementation details of the model.

The simplistic idea was to think of this knowledge gathering as creating active notes, hence the name NoteTaker. The not-so-simplistic notion was to broaden the modeling conversation we were having to include topics outside the physical sciences.

> *Anyways, that is my story of how the 1978 Smalltalk machine came to be called the NoteTaker. And it is my introduction to you of the educational goal for the next iteration of the Dynabook as a modeling environment.*

Is there now software that could provide the *modeling* solution I was thinking about when I stated the software challenge for the NoteTaker? At first, nothing popped up in my search except to be reminded of the interesting visualization ideas from Silicon Graphics, and the Xerox PARC NoteCards project in the early 80s.

Notecards was hypertext-based and an early contributor to the notion of personal knowledge databases. When doing a search for NoteCards, I mistakenly started by thinking Stuart Card was the responsible researcher, rather than his longtime collaborators Randy Trigg, Frank Halasz, and Tom Moran. As a consequence of one of those chance encounters that browsing the Internet should be famous for, I found a 2012 ACM SIGCHI talk by Stuart Card in which he is pitching the idea of a Lambda.Book. In this kind of book, *any text is computational,* so that the text itself can be activated, potentially to help maintain a detailed chain of causation.

In thinking about the NoteTaker challenge, there is a similar idea, that the student capturing information in the library search does so in the form of *actionable text that can itself become data used by the student to identify patterns of relationships and information flow*.

Moreover, this actionable text becomes part of a chain of causation so that a reader, seeing a pattern emerge, can capture "why" the pattern makes sense. Of course, expecting any system to be able to explain itself is likely the reason why intelligent computer-based tutoring programs are both hard and tedious to create, and why embracing the machine-learning idea of teaching the computer how to learn with statistics and probability theory, basically analyzing even crowd-sourced examples, is so attractive.

Stu Card also pitches something I mentioned earlier, having learned the idea from Stanford CS professor Donald Knuth: which is, that *you learn to write by reading*. Stu states this idea by saying that a Lambda-Book has two domain-specific languages: one is a User Interface language for readers, and the other is a Builder language for authors. This duality is often seen in many of today's construction or gaming software. But it introduces the question as to whether the duality is actually necessary, as it is not in, for example, the Squeak family of languages.

I think we can agree the NoteTaker challenge remains an unsolved problem, or at least I have not as yet found an existing solution.

Smalltalk Modeling Kits

In the 70s, we began to work out the idea of a "Kit", somewhat building on the Microworlds concept.

I will use the term “kit” when I am talking about a specific set of interacting objects and the ways in which those objects relate to one another, and how they can be accessed and refined. I also use the term “modeling” rather than “programming,” since, as you now know, creating, sharing, and challenging the veracity of models comprises a particular approach to learning that interests me, an approach often referred to as an *inquiry-based approach*. This approach is best understood as having the learners do experiments and observations. The learning expectation is that the students would extract some understanding of what kinds of elements are present in the observed activities, and some knowledge of how they think those elements relate to one another. This approach is the basis for the learning activities and exhibits at the SF Exploratorium, where I volunteered for some years.

Happily, incorporating modeling activities in school is not new. As a short list, consider what you might have done in school, to learn:

- how your natural language is composed
- how mechanical things interact
- how electrical things transmit power
- how social and political networks operate
- how the Earth and its inhabitants evolve
- how our planet exists in a larger galaxy of space
- how our body works as a system of organs and connectivity
- how we acquire and spend money
- how human history, cultural evolution, and especially cultural differences, landed us where we are today

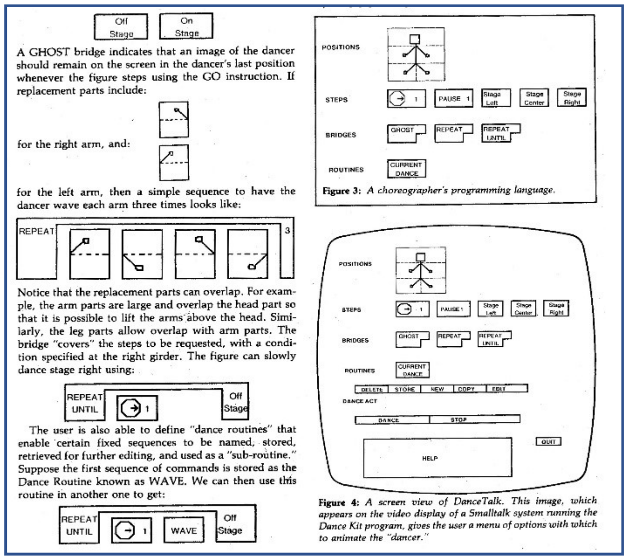
And schools do teach the application of *modeling tools*, such as number lines in early math, timelines for history, mapping, graphing, and flow diagramming, to name a few. All of these existing modeling tools and modeling examples provide us with the material we need to fulfill school expectations, and to check that a proposed Dynabook modeling environment is in alignment with standard curriculum expectations for learning outcomes. But the Dynabook mission is to change the process of learning so as to deliver the inquiry approach and to improve equality of education.

In my research for this talk, I did encounter some papers written about how to teach beginners concepts of object-oriented programming which were based on specific microworlds. But none that I thought offered new thinking on what are the critical properties of an environment for general purpose modeling or its libraries of kits.

[A decent survey of this work from the late 90s, early 2000s, is in a paper by Fahima Djelil et al entitled “Microworlds for Learning Object-oriented Programming: Considerations for Research to Practice.”]

Next, I am going to point to six different Smalltalk-based kits. I am summarizing these kits because they represent models explored in the pursuit of a Dynabook. And they inform the categories of questions with which I will end this talk.

Dance Kit. A very early kit we designed specifically for young kids was called the Dance Kit. It was documented in an article in *Byte Magazine in August 1981* as part of the special issue about Smalltalk that I organized. The Dance Kit idea was originally conceived by Bill Finzer to teach the BASIC programming language, but in the context of choreography. The Smalltalk-80 version was likely one of the earliest user interfaces to programming by manipulating *visual blocks*—here positions composed of steps, and bridges for iteration commands, collected into routines such as to kick and jump that the learner creates. Dance acts then make use of these routines.


Off Stage
On Stage

A GHOST bridge indicates that an image of the dancer should remain on the screen in the dancer's last position whenever the figure steps using the GO instruction. If replacement parts include:

for the right arm, and:

for the left arm, then a simple sequence to have the dancer wave each arm three times looks like:

REPEAT
3

Notice that the replacement parts can overlap. For example, the arm parts are large and overlap the head part so that it is possible to lift the arms above the head. Similarly, the leg parts allow overlap with arm parts. The bridge "covers" the steps to be requested, with a condition specified at the right girder. The figure can slowly dance stage right using:

The user is also able to define "dance routines" that enable certain fixed sequences to be named, stored, retrieved for further editing, and used as a "sub-routine." Suppose the first sequence of commands is stored as the Dance Routine known as WAVE. We can then use this routine in another one to get:

Figure 3: *A choreographer's programming language.*

Figure 4: *A screen view of DanceTalk. This image, which appears on the video display of a Smalltalk system running the Dance Kit program, gives the user a menu of options with which to animate the "dancer."*

Since the dancer object could take on any shape made up of parts, we could easily extend this Dance Kit into turtles, boxes, stick figures, and so on. It is not much of a leap to also represent the dance stage as a maze and direct the dances accordingly. Similar use of manipulating visual blocks representing language elements was adopted by Squeak, eToys, and Scratch.

ThingLab

The next kit to look at is ThingLab. ThingLab was created by Alan Borning, around 1978-1979, as his PhD dissertation for Stanford University. It was initially a Smalltalk-72 project.

A visual object-oriented simulation system, ThingLab was initially an exploration of designing a constraint-oriented system in which the user provides arbitrary inputs or outputs. Then the system solves for whatever is unknown.

The Thinglab browser references only those objects that are specifically in the modeling kit.

You can access a video on Youtube of a keynote talk by Alan Borning in 2016. It is ThingLab running originally in Smalltalk-76 on a Xerox Alto with 64K memory that was shared with the display. Note how large the ThingLab library is that represents only a few construction kit parts, and note Alan's explanation of how the visualization can be touched for debugging or understanding purposes.

What I worry about when seeing this video is that it shows a library that is a vast warehouse of parts. Scrolling to find parts is not likely a good approach for beginning learners. There needs to be some better filtering, visualizations, and search mechanisms based on pedagogical needs. I am reminded of the question I used to ask Unix gurus as to how they know what is available to reuse. Their answer was always, "you just know". Funny, but not going to be good enough given our target audience. Fortunately, Internet search systems tell us how we can shop for parts and get a filter onto an otherwise enormous library.

Using a system like Thinglab, a teacher could have an interactive animation to demonstrate ideas in the physical sciences. There are already computer-based education companies that make animations for this classroom purpose, but if teachers could make their own demonstrations, and students could modify the model, we can give teachers and students more input into their learning experience.

What also makes ThingLab so interesting is its view that constraints, whether equations to be solved or relations to be maintained, can express partial information, can be combined with other constraints, can be general purpose in the sense of their properties, are declarative, and can be searched and experimented with. So, these represent an important class of base objects we would need in a modeling environment.

Constraint programming is obviously not new nor unique to ThingLab. Indeed Alan Kay, in explaining the ideas of objects, constraint-based programming, and direct manipulation interfaces, always showed Ivan Sutherland's Sketchpad as an important system that solved complicated nonlinear systems of constraints. And, of course, Bertrand Meyer has long pushed the object-oriented technology community to understand the importance of being explicit about constraints, although the vocabulary of Eiffel speaks about "contracts among objects" in order to emphasize the need to build systems more reliably. As we consider how to lower the entry level for specifying models, we will need to make defining constraints a core capability of the modeling tools (whether in how objects communicate, or whether they can communicate at all).

Note that there are a number of other researchers that teach ways to declare constraints in an object-oriented language. Alan Borning references one such result—Babelsberg by Tim Felgentreff. Tim calls this *object constraint programming*.

Alternate Reality Kit

The next kit is the Alternate Reality Kit (ARK) written by Randy Smith around 1986. It was implemented in Smalltalk-80.

Ideas we glean from the ARK efforts are a class of constraints called Interactors, or interaction laws.

1. The ability for an object to interact in a particular way is specified by its relationship in the simulated world to a specific interactor (the law of motion is an example of an interactor as constraint, as is the law of gravity).
2. In a simulated world, interactors can be changed as a way to modify and explore how a law affects behavior.
3. Such objects need to be informed whether they can communicate with other objects.

Randy's initial work was more about exploring a user interface for accessing objects in an alternate reality. As I noted when talking about Stuart Card's ideas earlier, the object space or domain, and the domain's appropriate user interface for accessing the objects, are not necessarily independent pursuits. Moreover, it would be a mistake to assume that there is just one virtual world of simulation as generality often proves to make things harder for beginners.

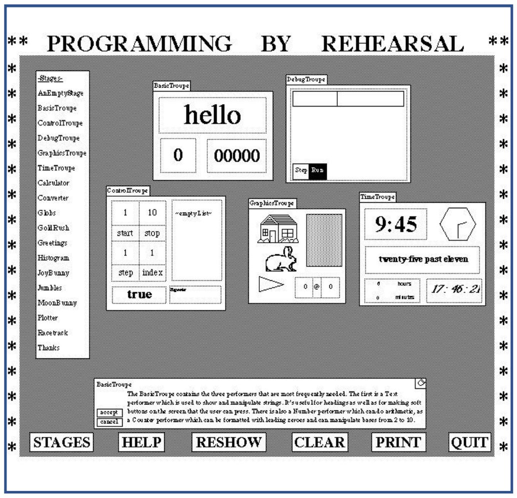
But each example begins to teach us what is the general nature of the modelling tools and libraries that will need to be supported in any successful solution that will span early school years. Just as in learning a natural language, depending on the age of the learner, the vocabulary and sentence structures grow in quantity and complexity, and span different domains. Preferably, we learn to constrain the vocabulary we use, depending on what we know about the others in the conversation.

Programming by Rehearsal

There were others who also sought a better understanding of how to support modeling. I name Laura Gould and Bill Finzer as two in our research laboratory who created a number of interesting educational tools. Bill's interest was data-centered, to look where data shows up in every subject studied. Bill's early interactive applications had the users literally point and change the data to ask "what if" questions. He would also embed games in the context of such applications to encourage the users to analyze the games they played in order to improve their strategies.

In the mid-80s, Bill and Laura collaborated on a Smalltalk-80 visual programming kit called Programming by Rehearsal, or the Rehearsal World, that was intended for creating educational software. Specifically, a nonprogrammer would be an educator authoring curriculum content.

There is a publication about Rehearsal World in the June, 1984 *Byte Magazine* special issue on computers and education. (https://archive.org/details/byte-magazine-1984-06/mode/2up, p187)



BasicTroupe
The BasicTroupe contains the three performers that are most frequently needed. The first is a Text performer which is used to show and manipulate strings. It's useful for headings as well as for making soft buttons on the screen that the user can press. There is also a Number performer which can do arithmetic, as a Counter performer which can be formatted with leading zeroes and can manipulate bases from 2 to 10.

To accommodate nonprogrammers, the metaphor of this kit is about performers on a stage (what is visible) and actors backstage supporting the performance (what is not visible). The stage is acting as a filter as to what the intended user can see.

The emphasis is on programming visually: only things that can be seen can be directly manipulated by the learner, while the author has access to the back stage. The design and programming process consists of moving "performers" around on "stages", and teaching them how to interact by sending "cues" to one another. The system relies almost completely on interactive graphics, and allows designers to react immediately to their emerging products by showing, at all stages of development, exactly what the potential users will see.

So an important quality of this kit is that it is always working, always displaying what the model represents.

There are many other examples of these kinds of efforts to create end-user programming or programming by demonstration, or by example, or by interaction…Pygmalion by David Canfield Smith, and others are nicely documented by Alan Cypher in a book dated 1992.

There are probably a lot of reasons to be interested in these approaches to programming. One is that the approach is more *playful* given the instant feedback, and likely to encourage experimentation.

The Simulation Kit

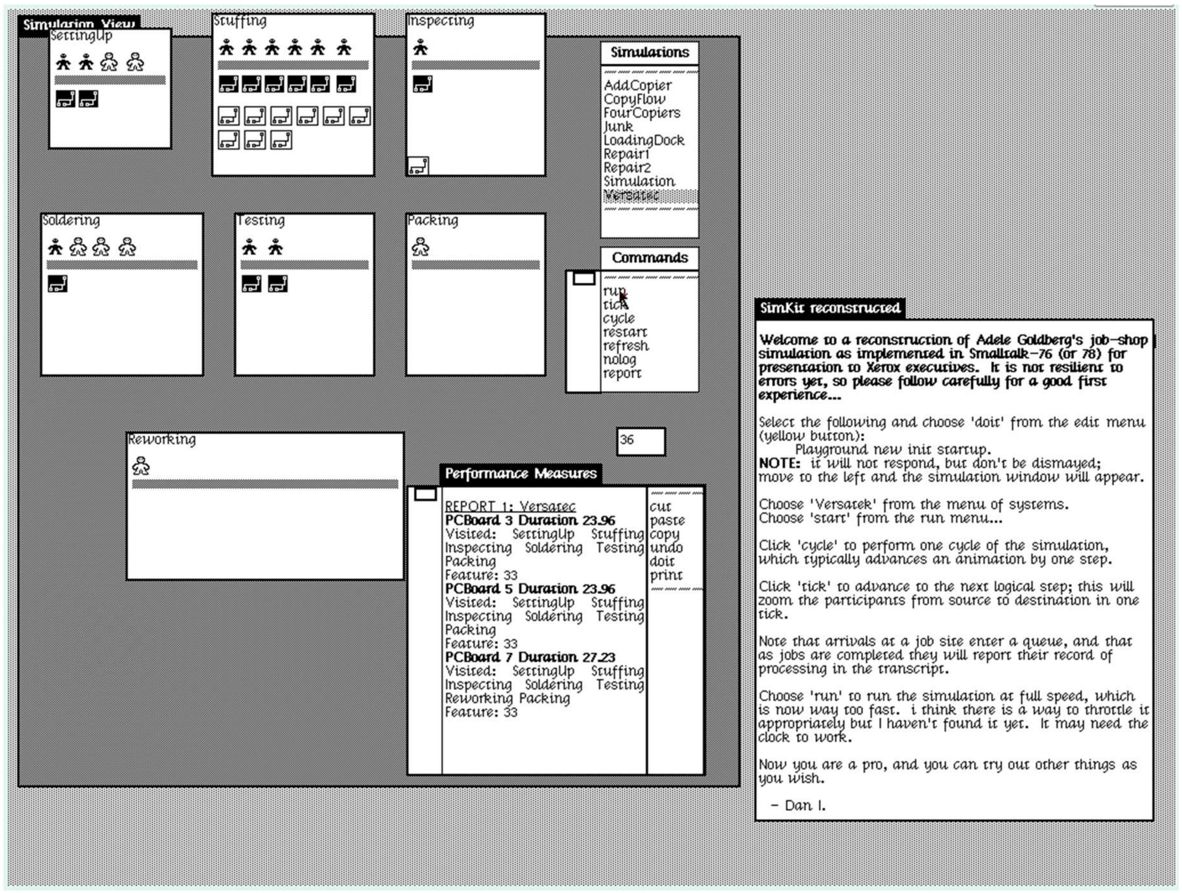
An important point made by our teaching experience is that, because we expect the learner to extend a model through programming new methods, not just by parameter modification, it is critical that some outcome that the learner mistakenly creates does not get explained using vocabulary outside of what has been already been introduced.

The obvious example from the Smalltalk system itself is its class browser and the debugger. Imagine you are creating a weather forecasting model and an error occurs in execution. But instead of seeing the error at the level of weather activity and the buttons and dials of the weather station, you see a bug in how text is displayed on the screen, thereby introducing an underlying part of the modeling environment that you have not yet learned about (and maybe do not want to learn about!). Certainly, dealing with an error outside of a kit is a distraction, potentially defocusing the learner's modeling intentions.

We had two opportunities to explore the problem of constraining content and context:

> The 1978-1981 work on the *Simulation Kit*, and
> *LearningWorks*, from the early 1990s

A version of the Simulation Kit underlying model was the large example provided in the original Smalltalk-80 book on *The Language and its Implementation.* It was motivated by a similar example used to explain Simula-68. However, the actual development environment interface to the underlying statistics and event-driven simulation model, was not published in that book.



```
REPORT 1: Versatec
PCBoard 3 Duration 23.96
Visited:  SettingUp  Stuffing
Inspecting  Soldering  Testing
Packing
Feature: 33
PCBoard 5 Duration 23.96
Visited:  SettingUp  Stuffing
Inspecting  Soldering  Testing
Packing
Feature: 33
PCBoard 7 Duration 27.23
Visited:  SettingUp  Stuffing
Inspecting  Soldering  Testing
Reworking Packing
Feature: 33
```

Welcome to a reconstruction of Adele Goldberg's job-shop simulation as implemented in Smalltalk-76 (or 78) for presentation to Xerox executives. It is not resilient to errors yet, so please follow carefully for a good first experience...

Select the following and choose 'doit' from the edit menu (yellow button):
    Playground new init startup.
NOTE: it will not respond, but don't be dismayed; move to the left and the simulation window will appear.

Choose 'Versatek' from the menu of systems.
Choose 'start' from the run menu...

Click 'cycle' to perform one cycle of the simulation, which typically advances an animation by one step.

Click 'tick' to advance to the next logical step; this will zoom the participants from source to destination in one tick.

Note that arrivals at a job site enter a queue, and that as jobs are completed they will report their record of processing in the transcript.

Choose 'run' to run the simulation at full speed, which is now way too fast. i think there is a way to throttle it appropriately but I haven't found it yet. It may need the clock to work.

Now you are a pro, and you can try out other things as you wish.

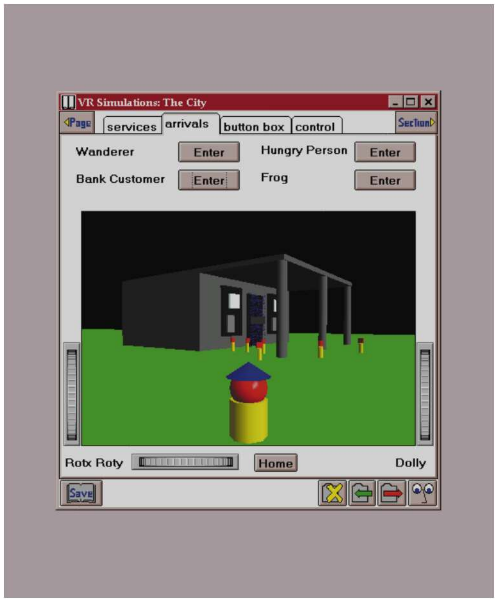
– Dan I.

The Simulation Kit provided the basics for implementing discrete, event-driven simulations. Actions to happen in parallel are synched by a clock. The browser for the Simulation Kit was called a Playground. It was constrained to show only the classes in the kit library: Simulation Objects such as Worker, Customer, Job, and Station; as well as all of the classes that define a statistical package. An instance of class Simulation manages the layout, arrival scheduling, reporting schedule, and a clock to register elapsed time. Neither Clock nor the Reporting window are modifiable by the programmer except to provide implementations to a fixed interface. Within a Playground, the user could create subclasses of those simulation classes, as well as schedule the arrival and work processes of workers and customers. In addition, the sub-views in the browser invoked their own editors; for example, browsing to a view of a Worker invoked a Bit Editor. There was, unfortunately, no semantic testing as to whether the learner writes into these methods complied with expectations.

The debugger for the Simulation Kit was also constrained so that only the implementation of classes specific to the kit would be made visible in the execution stack if an error occurs. Traversing deeper into the system, for example, using ctrl-C to interrupt the running system to see how text display was handled, was not accessible.

Deployment was to the 10 top executives of the Xerox Corporation, who, in early 1978, were at the research center for a 2-day seminar on components-based software. Thus, the curriculum consisted of models of Xerox businesses, such as a copier/duplicator customer service center, a machine repair center, a company central reproduction center, office printing center, and a manufacturing plant. Looking around the room, each laboratory participant could see that, although everyone started with the same modeling kit, everyone was deploying a different simulation.

LearningWorks



The basic idea behind LearningWorks was to create a software implementation of a kind of smart book that we called a LearningBook. "Book" was a metaphor for packaging a set of class definitions to find, read, and use—the capabilities that had been more typically seen visually in a Smalltalk Browser or Debugger—along with the user interface accessing-constraints we identified in doing the Simulation Kit. LearningWorks was first reported on at OOPSLA in 1990 while I was still involved at ParcPlace Systems. It was then part of a contract to Neometron by the Mitsubishi Electric Corporation carried out by myself, David Liebs, Wlodek Kubalski, Tami Lee, Jasen Minton, and Steven Abell.

LearningWorks was comprised of a LearningBook framework, seen as windows on the display screen that emulate the structure of a book with sections and pages. The pages contain application activities with which the user can interact, and the objects needed to realize these applications. LearningBooks reference other LearningBooks as a way to specify which object definitions are to be included, and which should be visible to the user. An author uses these packaging features to build a set of coordinated exercises that can limit access to objects in the larger system. Such constraint can be done at the book and also the book section levels. There are three kinds of books:

- Activity books contain the context in which the learner explores curriculum topics and carries out the curriculum tasks, including tools for defining object methods, documenting, and testing. As you would expect, there is an Inspector book to examine the structure of objects and applications, and a Debugger book to examine interrupted execution, both constrained by the specification of the Activity Book that was running when the interruption occurred.
- Authoring a course consists of creating a Course Binder that contains the overall curriculum plan, and one or more LearningBooks that carry out the plan.
- A book supporting a team communications framework is included so students can form teams and work together on the Internet to do a project. They can share their work by sharing LearningBooks containing the work of each subteam. The pages of each subteam's

book contain code that fulfills the subteam's system requirements, test suite, documentation, and example uses.

The peer-to-peer communications framework that supports real-time sharing of LearningBooks was based on SPLINE, developed at the Mitsubishi Electric Research Laboratory to enable social virtual reality in 3D.

The image above shows a view of a simple LearningBook, a city simulation executing in 3D space.

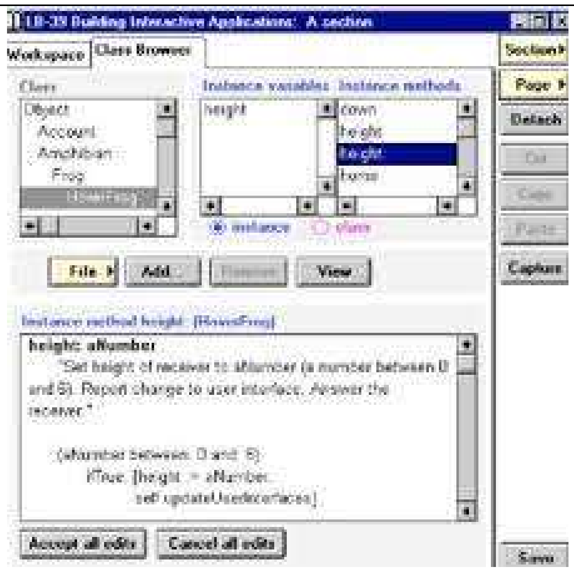
You can see the influence of the prior studies that I described earlier. But LearningWorks required a different approach to a graphical user interface editor because of the special microworld elements: dynamic creatures that can be animated, places or "cells" in which creatures live, and events that trigger creature and cell behavior. There is also a bit of complexity to enable what is constructed to be seen and what is in the background, somewhat reminiscent of the Rehearsal World.

Now let's turn to the LearningWorks modifications done at the Open University in the UK

This last example-project targeted education of software engineers. While working on LearningWorks, we were approached by Mark Woodman from the Open University in the UK. He was starting the creation of a new Open University first course in Computer Science. The instructional format for the University was to use the postal system to deploy assignments, including any materials (even a small chemistry lab), although in the timeframe of our work with Mark, distribution would transition from the post office to the Internet.

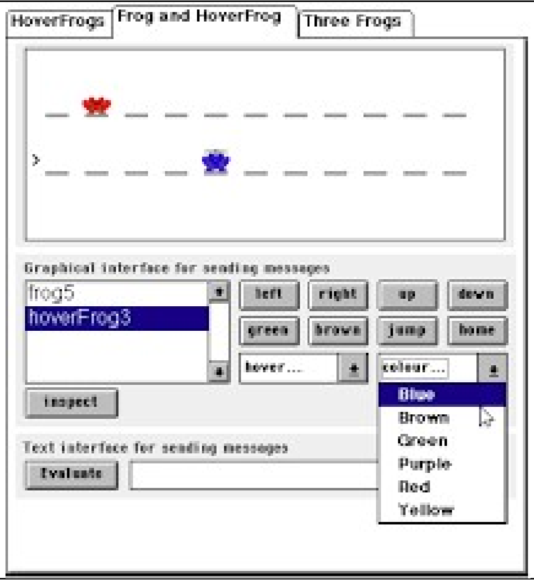
The problem posed by Mark was that he be able to unfold his new CS curriculum, delivering the curriculum content to his remote students in a form in which they could minimize the perceived confusion when a student, working alone, is given access to a very rich programming environment.

In 1994, when the OU was planning its new first course in CS, the decision to "embrace an objects-first approach and to choose Smalltalk as the primary teaching vehicle", was seen "as a radical and controversial step." Nonetheless, the project we started together in 1994, when released at the time of its 1999 publication, had already attracted over ten thousand (10,000) students in the UK and Europe, soon to go to Singapore, and then the U.S.—all countries in which the Open University was incorporated. As Mark has said, this course was the largest such course in the world. The average age of a student was 37 years old, as was fitting given the OU model to support people pursuing their education regardless of age and situation, to learn at their own pace, and to earn undergraduate as well as graduate level degrees.



```
height: aNumber
    "Set height of receiver to aNumber (a number between 0
and 6). Report change to user interface. Answer the
receiver."

    (aNumber between: 0 and: 6)
        ifTrue: [height := aNumber.
                self updateUserInterfaces]
```



What LearningWorks offered was a way to progressively disclose details and thereby control disclosure of an otherwise complex programming environment. Such disclosure meant that the students would gradually learn to use a complex commercially available programming system. Note that in the same timeframe as Mark's work, this particular issue—how to support learners of a sophisticated programming environment working remotely—is dealt with in an interesting paper by Thomas Kuhne in about the same timeframe, where Kuhne was comparing Smalltalk to other choices, like Eiffel and Java. Kuhne added some diagramming visualizations to Smalltalk to handle some student confusions, much as the OU LearningWorks did.

OU terminology was that each LearningBook contained a *microworld* to be explored and modified. What OU LearningWorks added were:

- Some simple changes to the user interface to constrain the size and numbers of windows, and to pull out LearningBook pages into a separate window so that instructions and inspections could be done simultaneously, thereby treating the book more like a notebook or, as Mark would say, a ProjectBook.
- The OU team added new tools: an HTML browser, special visualization tools for arguments and parsing expressions, workspaces for controlling microworld behavior textually in addition to the existing button clicking controls, and a class reporter that showed a complete view of a class in a single read-only textual view.
- Students could create their own user interface to a microworld, made possible by the careful separation of the microworld modeling from its interaction interface. The GUI tool checks to make sure it is complete and consistent, i.e., fulfills the expected protocol for interacting with the microworld objects.
- The OU team created a curriculum as a set of LearningBooks to teach software engineering concepts. Their air traffic control simulation was especially interesting as it was used to teach about professional ethics.
- And they added the restriction that only one LearningBook would be loaded at a time so that the unfolding curriculum could include modifications or replacements to the objects defined in a prior LearningBook.

This last addition raises a flag for any follow-up work to LearningWorks.

There needs to be a better way to share teacher-defined or student-created models without restricting how the new objects in the model are named—sort of similar to allowing for the composition of system modules into an executable system by a team that did not agree on naming conventions, or could not simply rely on automatically added prefixes. Student LearningBooks need to serve as a way to package an individual student's assignment work, protecting the student's private name space. But curriculum shared across students as a form of Curriculum LearningBooks containing a library of named objects also pose a different problem. The nature of progressive unfolding meant that each stage of the curriculum either needed a new book into which the student could load work from an existing book, or a way to introduce objects into an existing book to expand the availability of models to explore without creating conflicts in the name space.

Given the experimental nature of their work, the OU needed a way to update the underlying LearningWorks itself in a way that new and remote learners could handle. To that end, the system was shipped in two parts: (1) an unchangeable and safe image containing the underlying VisualWorks, the definitions for the LearningWorks framework, and the core changes to LearningWorks created by the OU; and (2) the LearningBooks themselves that comprised the unfolding curriculum. The unchangeable image offered a problem in how to patch any bugs given a large-scale distribution worldwide, which was eventually resolved by including the patches in an individual LearningBook, effectively modifying the underlying Smalltalk when a LearningBook was executing.

So, there were lots of things to think about changing in LearningWorks to do a better job of supporting the pedagogical plan of an ambitious course. Both the development of LearningWorks and the OU version overlapped in timing, so to some extent we had a typical coordination issue. I have to say that one of the fun parts of preparing this talk was rereading Mark and his team's publications. They are definitely an interesting presentation of what about the underlying Smalltalk and the LearningWorks systems needed enhancement or caused pedagogical issues in their design of a first course focused on teaching good software engineering practices. They smartly figured out how to program around most issues with new tools and effective use of the LearningWorks filtering support for ***progressive disclosure.***

What are the themes and questions raised from these projects?

I have brought up several themes and related questions around those themes. Quickly, let's review them.

First, what is the role of text, especially where the youngest learners are involved. Is text always actionable? For example, clicking on a single token…does that raise an explanation or a "show me" reaction? Consider what happens when you click on a word in an eBook in Kindle and a dictionary explanation appears.

Do buttons with icons properly replace text for the youngest learners?

What precisely do we mean by a *pattern of behavior* or a *pattern of relations*? Is identifying patterns and understanding what raises this behavior a critical aspect of coming up with an abstract representation of a set of models?

Third, we explored the idea of learning to write by reading. Is doing so the requirement for effective reuse?

Help, and Objects and messages paradigm

Fourth, where do users get help? Is help necessarily about building intelligent tutors, or is the current climate of building Internet community something to be leveraged instead? If we start by teaching users how to effectively use debuggers, do the users have the tool they need to get help?

Fifth, it seems clear that the objects and message paradigm serves the educational inquiry-based mission quite well. Is there a limit? Do we have to specifically teach object-oriented programming as how we talk to a computer in order to use the modeling environment, and, if so, at what point?

Often time, the way we support *pairing* user interfaces—views and interactions—with underlying models can get complicated. Certainly, an initial encounter with VisualWorks, which was designed for professional developers, might testify to this concern. For pedagogical purposes, should some simplifications to progressively move towards the full system capability be considered?

Role of Constraints

Is the notion of constraints just another class of objects, or some new first-class category of base class definitions?

There are many ways in which constraints likely need first class support in the modeling environment as gleaned from past projects. I list some here, but probably missed a few.

- Maintain how specific properties of one object constrains the behavior of another
- Maintain constrained relationships between classes of objects or between specific instances
- Declare whether particular objects are visible to other objects, can communicate with those objects, and are restricted in behavior by the properties of other objects
- Hide user access to implementation details outside that which pertains to the elements of the kit itself when creating, looking to reuse, or to debug

Reuse, and the role of constraints

Much of the Smalltalk system was about supporting reuse and programming by refinement. Will the approaches that work for professionals suffice or need to be supplanted for younger learners? Do the common Internet search capabilities offer new ways to teach how to find existing models and views that can pair with those models?

Modeling kits for delivering curriculum

I brought up two curriculum design principles. Does the modeling environment provide a curriculum deployment framework for adopting the principles of constrained disclosure and progressive disclosure?

Conclusion

Frankly, we would only know for sure if we have designed the Dynabook modeling environment if we build a *multi-year curriculum*, that is, modelling projects for children of all ages, and see what is needed to support the kind of modeling that should be a part of the way we educate our children.

Clearly the key to future success is by taking on this curriculum challenge, iteratively building the modeling environment until we have something worth criticizing.

But I do want to throw out some things that come from modern computing's existing ability to use sensors and remote controls. Again, for pedagogical purposes, creating models that drive physical devices like robots can be a lot of fun. And perhaps, to beg the obvious, we need to remember that information used by these models will most likely come from direct access to Internet information stores.

It appears that most things needed for a good *first* modeling environment have been experimented with and also broadly distributed. Maybe it is just the packaging that is missing. The challenge is that this packaging involves coming up with a user interface that can progressively unfold from something simple and manipulable enough for young kids, yet evolve gracefully for more experienced modelers. And it does seem that ways to connect a model to Internet data-gathering still seems something to-be-determined.

And of course, success means solving the NoteTaker modeling challenge.