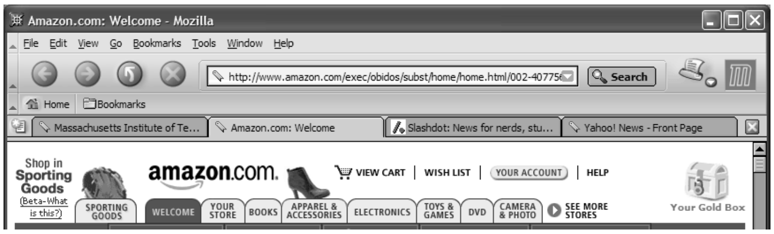
Lecture 8: Computer Prototyping

UI Hall of Fame or Shame?



Today’s candidate for the User Interface Hall of Fame is **tabbed browsing**, a feature found in almost all web browsers (Mozilla, Safari, Konqueror, Opera) except Internet Explorer. With tabbed browsing, multiple browser windows are grouped into a single top-level window and accessed by a row of tabs. You can open a hyperlink in a new tab by choosing that option from the right-click menu.

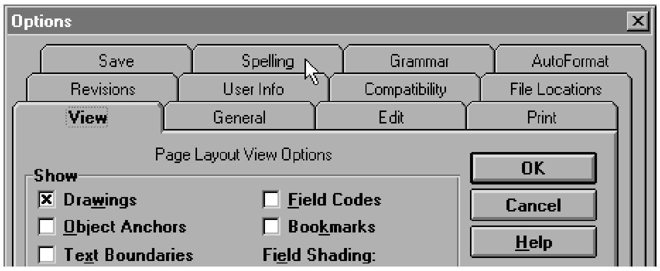
Tabbed browsing neatly solves a scaling problem in the Windows taskbar. If you accumulate several top-level Internet Explorer windows, they cease to be separately clickable buttons in the taskbar and merge together into a single Internet Explorer button with a popup menu. So your browser windows become **less visible** and **less efficient** to reach.

Tabbed browsing solves that by creating effectively a separate task bar specialized to the web browser. But it’s even better than that: you can open multiple top-level browser windows, each with its own set of tabs. Each browser window can then be dedicated to a particular task, e.g. apartment hunting, airfare searching, programming documentation, web surfing. It’s an easy and natural way for you to create task-specific groupings of your browser windows. That’s what the Windows task bar tries to do when it groups windows from the same application together into a single popup menu, but that simplistic approach doesn’t work at all because the Web is such a general-purpose platform. So tabbed browsing clearly wins on **task analysis**.

Another neat feature of tabbed browsing, at least in Mozilla, is that you can bookmark a set of tabs so you can recover them again later – a nice **shortcut** for task-oriented users.

What are the downsides of tabbed browsing? For one thing, you can’t compare the contents of one tab with another. External windows would let you do this by resizing and repositioning the windows. Another problem is that, at least in Mozilla, tab groups can’t be easily rearranged– moved to other windows, dragged out to start a new window.

Tabbing Doesn't Scale



Page Layout View Options

Show

☒ Drawings ☐ Field Codes

☐ Object Anchors ☐ Bookmarks

☐ Text Boundaries Field Shading:

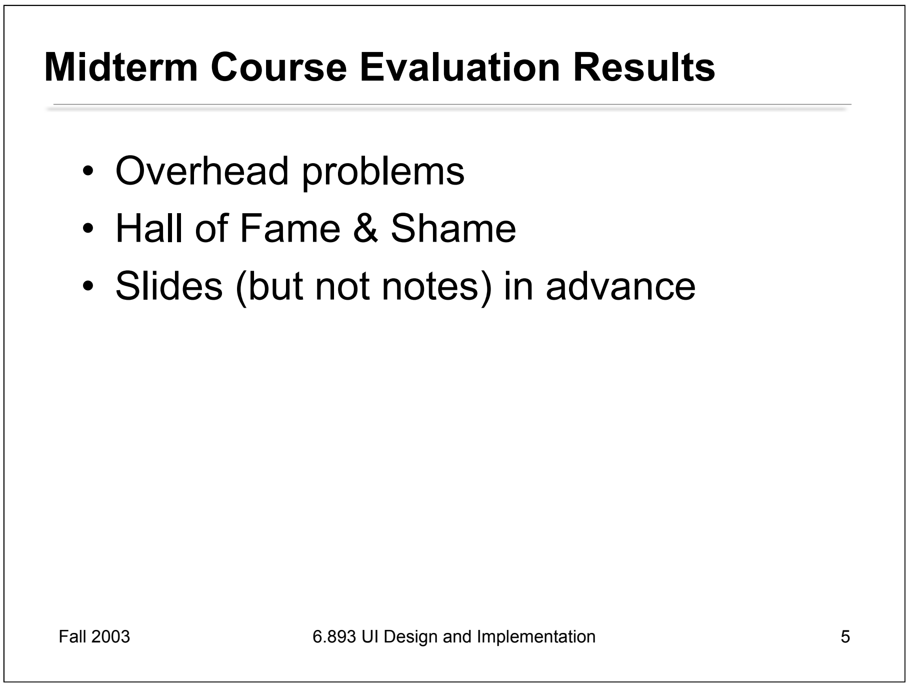
Another problem is that tabs don't really scale up either – you can't have more than 5-10 without shrinking their labels down to unreadability. Some designers have tried using **multiple rows of tabs**, but this turns out to be a horrible idea. Here's the Microsoft Word 6 option dialog. Clicking on a tab in a back row (like Spelling) has to move the whole row forward in order to maintain the tabbing metaphor. This is disorienting for two reasons: first, because the tab you clicked on has leaped out from under the mouse; and second, because other tabs you might have visited before are now in totally different places. Some plausible solutions to these problems were proposed in class – e.g., color-coding each row of tabs, or moving the front rows of tabs *below* the page. Animation might help too. All these ideas might reduce disorientation, but they involve tradeoffs like added visual complexity, greater demands on screen real estate, or having to move the page contents in addition to the tabs. And none of them prevent the tabs from jumping around, which is a basic problem with the approach.

As a rule of thumb, only one row of tabs really works, and the number of tabs you can fit in one row is constrained by the screen width and the tab label width. Most tabbing controls can scroll the tabs left to right, but scrolling tabs is definitely slower than picking from a popup menu.

In fact, the Windows task bar actually scales better than tabbing does, because it doesn't have to struggle to maintain a metaphor. The Windows task bar is just a row of buttons. Expanding the task bar to show two rows of buttons puts no strain on its usability, since the buttons don't have to jump around. Alas, you couldn't simply replace tabs with buttons in the dialog box shown here. (Why not?) Tabbed browsing probably couldn't use buttons either, without some careful graphic design to distinguish them from bookmark buttons.

Today's Topics

- Course evaluation results
- Quiz preview
- Paper prototype post-mortem
- Computer prototyping



Midterm Course Evaluation Results

- Overhead problems
- Hall of Fame & Shame
- Slides (but not notes) in advance

Thanks to everybody who filled in a course evaluation form. Here were some changes we're making as a result:

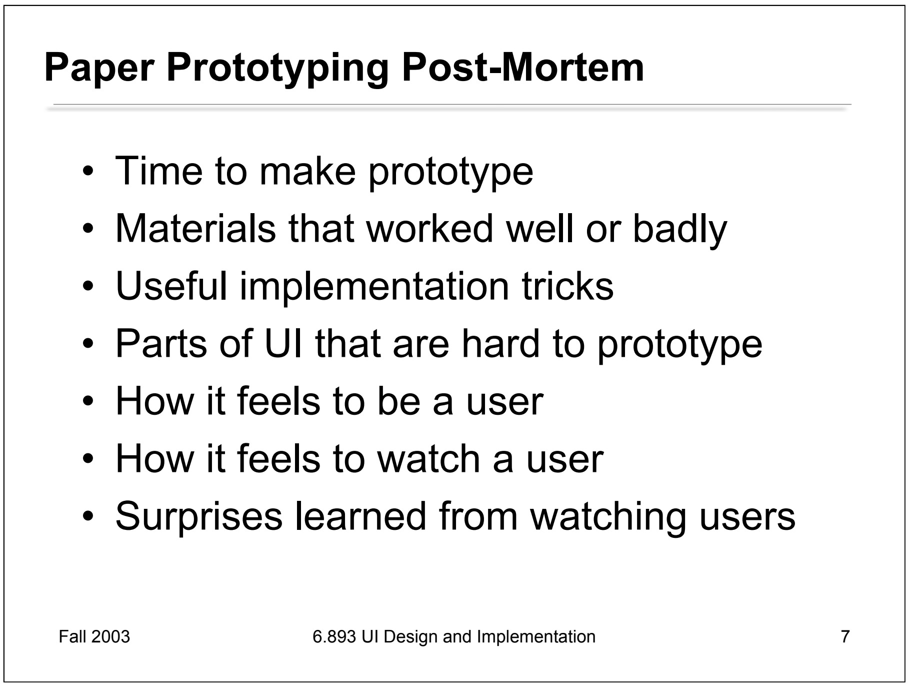
•The overhead projector is hard to read. We'll try to fix that by projecting on the front wall instead of the side.

•More Fame, less Shame. We'll also try to streamline the Hall of Fame & Shame, using only one example per class. That will help focus the discussion and keep it from eating away too much time for lecture content.

•Slides for each lecture will be available the night before or the morning of the lecture, for students who want to print them out as a note-taking aid. My own notes under each slide (what you're reading now) won't be included until after lecture, so if you don't plan to use these slides for note-taking, don't waste paper on them.

Quiz on Wednesday

- Topics
  - Usability
  - Iterative design
  - User & task analysis
  - Model human processor
  - Color
  - Conceptual models & metaphor
  - Affordance, constraint, visibility, feedback
  - Errors
  - Nielsen's heuristics
  - Heuristic evaluation
  - Prototyping
  - Graphic design principles
- Everything is fair game
  - Class discussion, lecture notes, readings, assignments
- Closed book exam, 80 minutes



Paper Prototyping Post-Mortem

- Time to make prototype
- Materials that worked well or badly
- Useful implementation tricks
- Parts of UI that are hard to prototype
- How it feels to be a user
- How it feels to watch a user
- Surprises learned from watching users

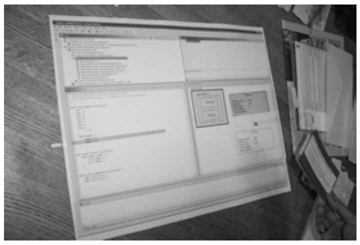
We discussed what we learned from paper prototyping. Some of the highlights:

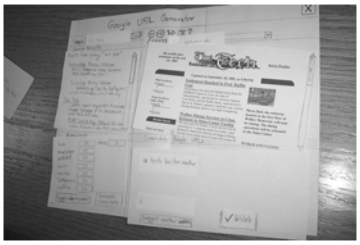
•**Time.** Most people took 4-12 hours to build their paper prototypes. Steve mentioned that his group did some iteration while building the prototype, which adds to the time, but the time like that is well spent. pays off. Some people commented that since the prototype was being graded, they spent more time making it look good than they might have otherwise.

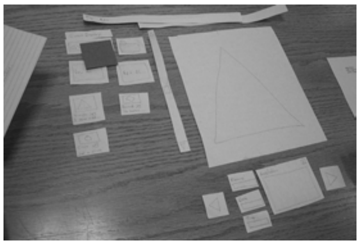
•**Materials.** Ellie mentioned that her group felt the most free to explore with plain old printer paper, because its cost is zero. Posterboard or index cards felt like a scarcer resource, which inhibited putting ideas down on it. Joe mentioned that Post-it glue is cool, but some little pieces got so sticky that they were hard to pick up from the prototype. So don't use too much, and leave a tab unsticky so you can pull the piece off.

•**Hard to prototype:** mouse-over feedback, since users didn't want to wave their fingers over the prototype (Matt); text editing, since there are so many things users might do (Min). Janet pointed out that drag & drop behaves differently in paper prototypes; users prefer to point to pick up and point to put down, rather than dragging a finger across the surface.

Hand-Drawn or Not?







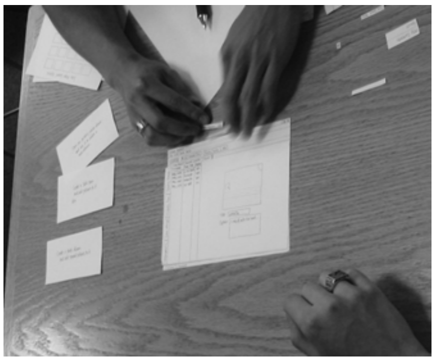
Fall 2003 6.893 UI Design and Implementation 8

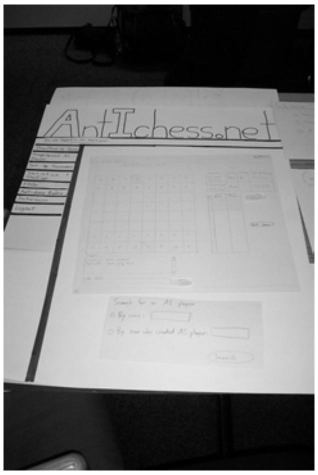
Here are some of the prototypes made by the class. Should a paper prototype be hand-sketched or computer-drawn? Generally hand-sketching is better in early design, but sometimes realistic images can be constructive additions. Top left is a prototype for an interface that will be integrated into an existing program (IBM Eclipse), so the prototype is mostly constructed of modified Eclipse screenshots. The result is very clean and crisp, but also tiny – it's hard to read from a distance. It may also be harder for a test user to focus on commenting about the new parts of the interface, since the new features look just like Eclipse. A hybrid hand-sketched/screenshot interface might work even better.

The top right prototype shows such a hybrid – a interface designed to integrate into a web browser. Actual screenshots of web pages are used, mainly as props, to make the prototype more concrete and help the user visualize the interface better. Since web page layout isn't the problem the interface is trying to solve, there's no reason to hand-sketch a web page.

The bottom photo shows a pure hand-sketched interface that might have benefited from such props -- a photo organizer could use real photographs to help the user think about what kinds of things they need to do with photographs. This prototype could also use a **window frame** – a big posterboard to serve as a static background.

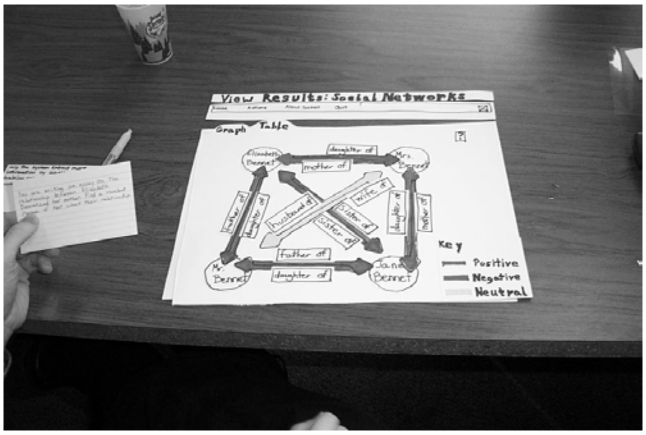
Size Matters





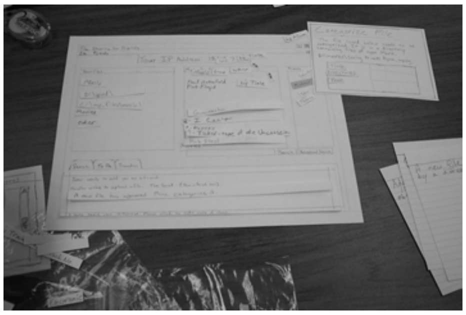
Both of these prototypes have good window frames, but the big one on the right is easier to read and manipulate.

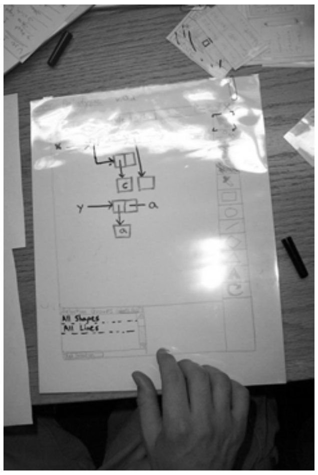
The Importance of Writing Big and Dark



This prototype is even easier to read. Markers are better than pencil. (Whiteout and correction tape can fix mistakes as well as erasers can!) Color is also neat, but don't bother unless color is a design decision that needs to be tested, as it is in this prototype. If color doesn't really matter, monochromatic prototypes work just as well.

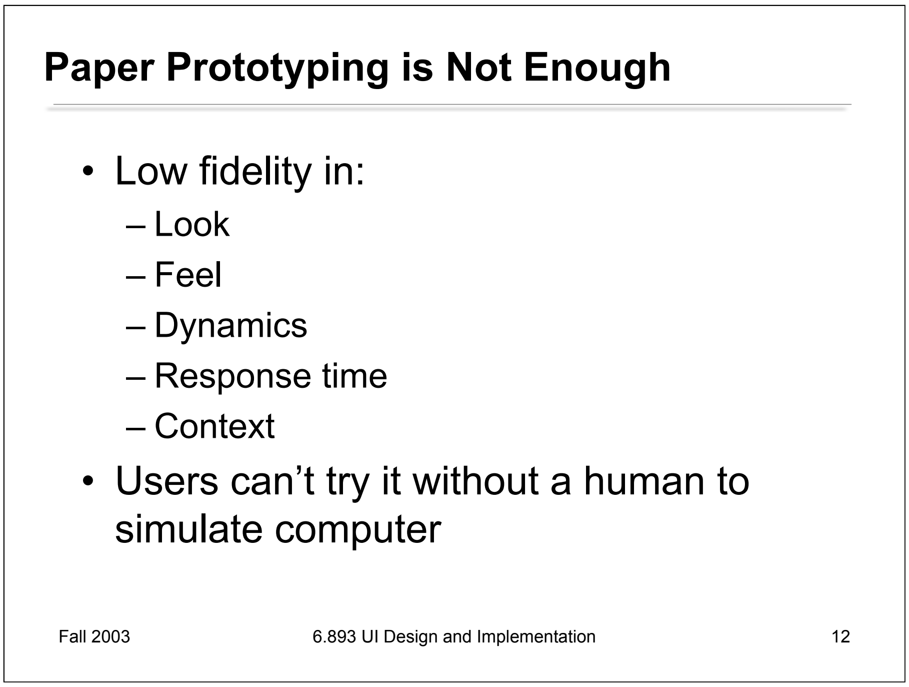
Post-it Glue and Transparencies are Good





The prototype on the left has lots of little pieces that have trouble staying put. Post-it glue can help with that.

On the right is a prototype that's completely covered with a transparency. Users can write on it directly with dry-erase marker, which just wipes off – a much better approach than water-soluble transparency markers. With multiple layers of transparency, you can let the user write on the top layer, while you use a lower layer for computer messages, selection highlighting, and other effects.



Paper Prototyping is Not Enough

- Low fidelity in:
  - Look
  - Feel
  - Dynamics
  - Response time
  - Context
- Users can’t try it without a human to simulate computer

Paper prototyping is neat, but it’s not enough. We discussed some of these drawbacks in the paper prototyping lecture.

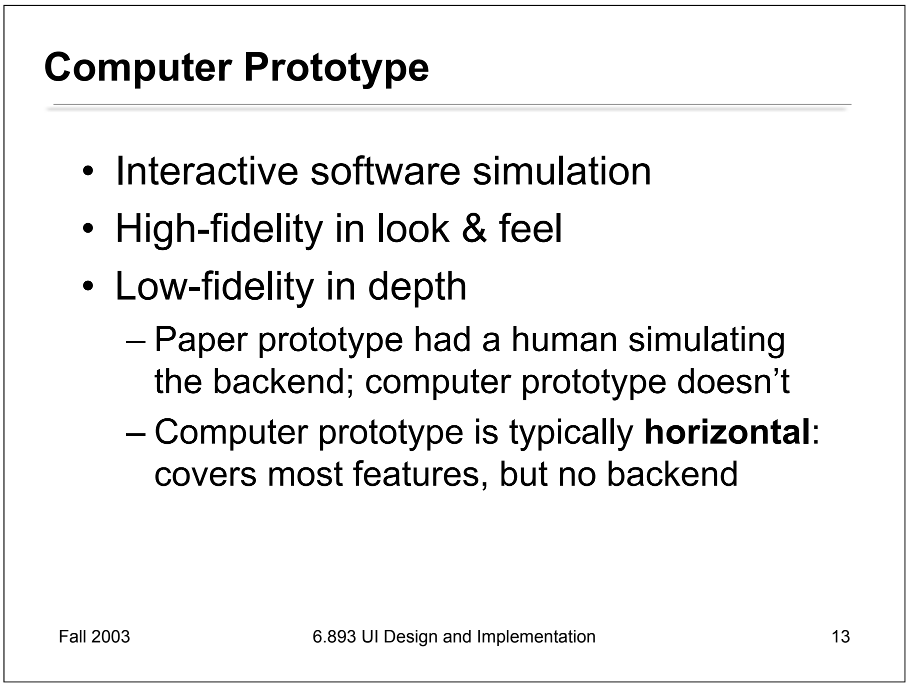
First, paper prototypes are low-fi in **look**. It’s sometimes hard for users to recognize widgets that you’ve hand-drawn, or labels that you’ve hastily scribbled. A paper prototype won’t tell you what will actually fit on the screen, since your handwritten font and larger-than-life posterboard aren’t realistic simulations. A paper prototype can’t easily simulate some important characteristics of your interface’s look – for example, does the selection highlighting have enough **contrast**, using visual variables that are **selective**, does it float above the rest of the design in its own layer? You have to make decisions about graphic design at some point, and you want to iterate those decisions just like other aspects of usability. So we need another prototyping method.

Paper prototypes are also low-fi in **feel**. Finger & pen doesn’t behave like mouse & keyboard (e.g., the drag & drop issue mentioned earlier). Even pen-based computers don’t really feel like pen & paper. You can’t test Fitts’s Law issues, like whether a button is big enough to click on.

Paper is low-fi in **dynamic feedback**. You don’t get any animation, like spinning globes, moving progress bars, etc. You can’t test mouse-over feedback, as we mentioned earlier.

It’s also low-fi in **response time**, because the human computer is far slower than the real computer. So you can’t measure how long it takes users to do a task with your interface.

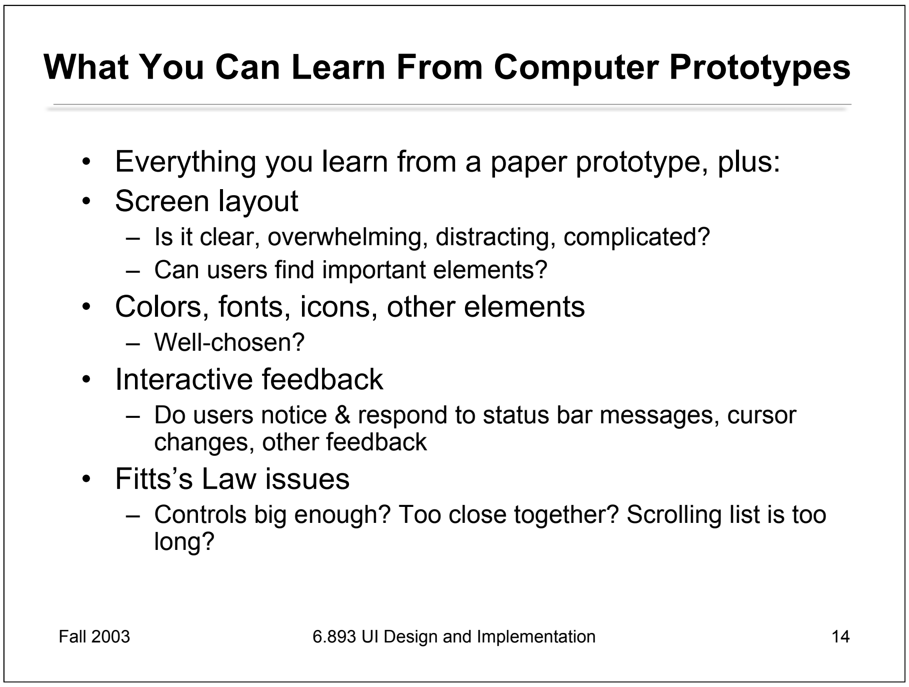
Finally, paper is low-fi with respect to **context of use**. A paper prototype can only really be used on a tabletop, in office-like environment. Users can’t take it to grocery store, subway, aircraft carrier deck, or wherever the target interface will actually be used. If it’s a handheld application, the ergonomics are all wrong; you aren’t holding it in your hand, and it’s not the right weight.



Computer Prototype

- Interactive software simulation
- High-fidelity in look & feel
- Low-fidelity in depth
  - Paper prototype had a human simulating the backend; computer prototype doesn't
  - Computer prototype is typically **horizontal**: covers most features, but no backend

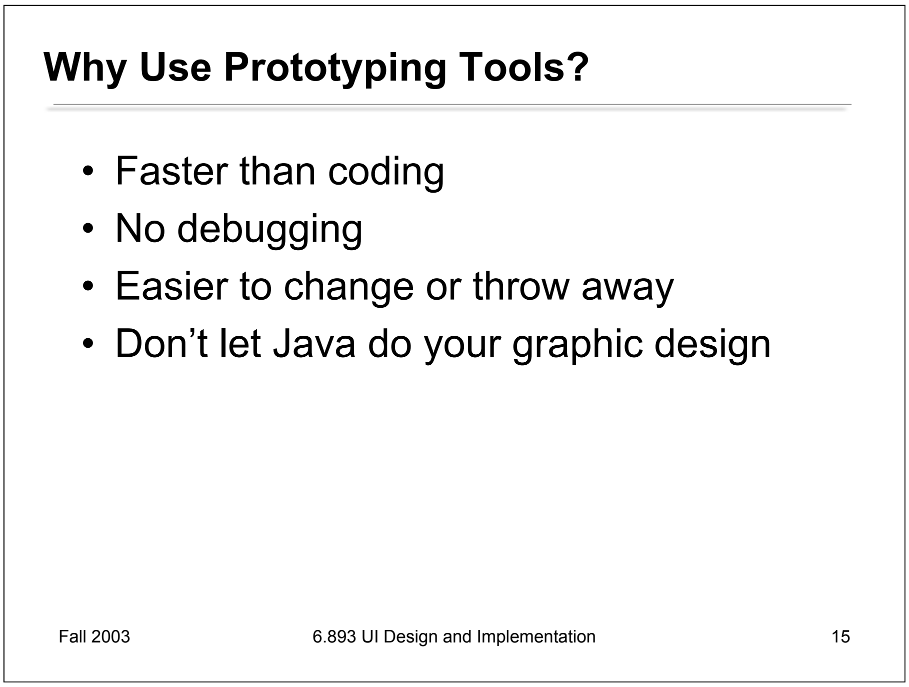
So at some point we have to depart from paper and move our prototypes into software. A typical computer prototype is a **horizontal** prototype. It's high-fi in look and feel, but low-fi in depth – there's no backend behind it. Where a human being simulating a paper prototype can generate new content on the fly in response to unexpected user actions, a computer prototype cannot.



What You Can Learn From Computer Prototypes

- Everything you learn from a paper prototype, plus:
- Screen layout
  - Is it clear, overwhelming, distracting, complicated?
  - Can users find important elements?
- Colors, fonts, icons, other elements
  - Well-chosen?
- Interactive feedback
  - Do users notice & respond to status bar messages, cursor changes, other feedback
- Fitts's Law issues
  - Controls big enough? Too close together? Scrolling list is too long?

Computer prototypes help us get a handle on the graphic design and dynamic feedback of the interface.



Why Use Prototyping Tools?

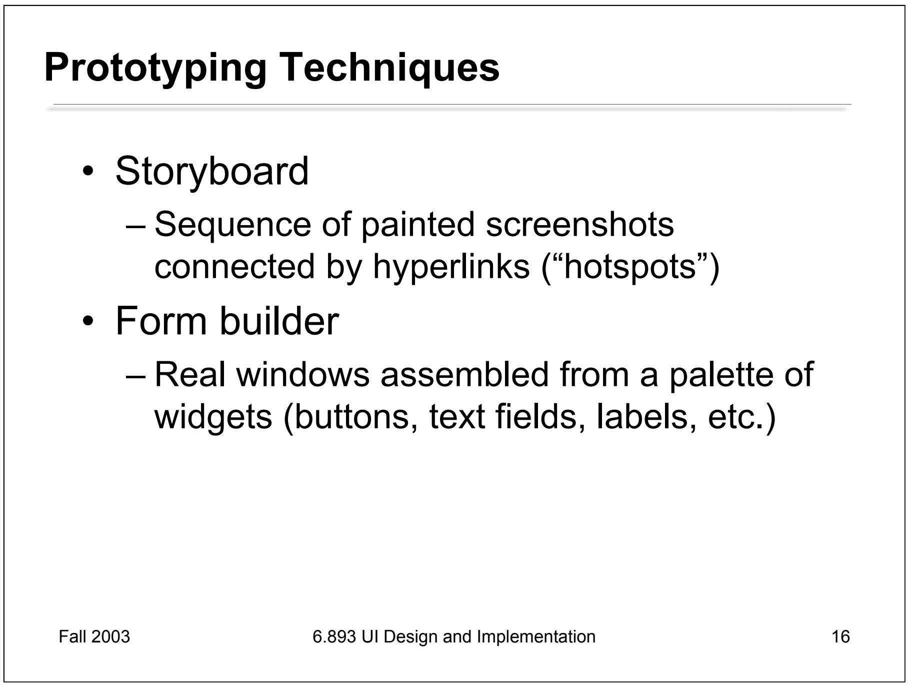
- Faster than coding
- No debugging
- Easier to change or throw away
- Don’t let Java do your graphic design

One way to build a computer prototype is just to program it directly in an implementation language, like Java or C++, using a user interface toolkit, like Swing or MFC. If you don’t hook in a backend, or use stubs instead of your real backend, then you’ve got a horizontal prototype.

But it’s often better to use a **prototyping tool** instead. Building an interface with a tool is usually faster than direct coding, and there’s no code to debug. It’s easier to change it, or even throw it away if your design turns out to be wrong. Recall Cooper’s concerns about prototyping: your computer prototype may become so elaborate and precious that it *becomes* your final implementation, even though (from a software engineering point of view) it might be sloppily designed and unmaintainable.

Also, when you go directly from paper prototype to code, there’s a tendency to let your UI toolkit handle all the graphic design for you. That’s a mistake. For example, Java has layout managers that automatically arrange the components of an interface. Layout managers are powerful tools, but they produce horrible interfaces when casually or lazily used. A prototyping tool will help you envision your interface and get its graphic design right first, so that later when you move to code, you know what you’re trying to persuade the layout manager to produce.

Even with a prototyping tool, computer prototypes can still be a tremendous amount of work. When drag & drop was being considered for Microsoft Excel, a couple of Microsoft summer interns were assigned to develop a prototype of the feature using Visual Basic. They found that they had to implement a substantial amount of basic spreadsheet functionality just to test drag & drop. It took two interns their entire summer to build the prototype that proved that drag & drop was useful. Actually adding the feature to Excel took a staff programmer only a week. This isn’t a fair comparison, of course – maybe six intern-months was a cost worth paying to mitigate the risk of one fulltimer-week, and the interns certainly learned a lot. But building a computer prototype can be a slippery slope, so don’t let it suck you in too deeply. Focus on what you want to test, i.e., the design risk you need to mitigate, and only prototype that.



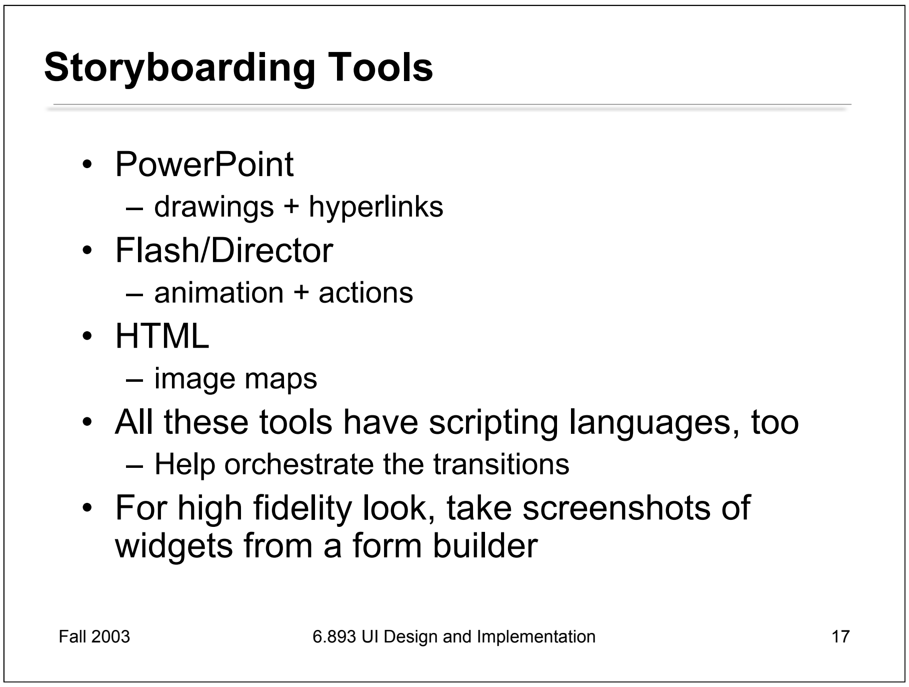
Prototyping Techniques

- Storyboard
  - Sequence of painted screenshots connected by hyperlinks (“hotspots”)
- Form builder
  - Real windows assembled from a palette of widgets (buttons, text fields, labels, etc.)

There are two major techniques for building a computer prototype.

A **storyboard** is a sequence (a graph, really) of fixed screens. Each screen has one or more **hotspots** that you can click on to jump to another screen. Sometimes the transitions between screens also involve some animation in order to show a dynamic effect, like mouse-over feedback or drag-drop feedback.

A **form builder** is a tool for drawing real, working interfaces by dragging widgets from a palette and positioning them on a window.



Storyboarding Tools

- PowerPoint
  - drawings + hyperlinks
- Flash/Director
  - animation + actions
- HTML
  - image maps
- All these tools have scripting languages, too
  - Help orchestrate the transitions
- For high fidelity look, take screenshots of widgets from a form builder

Here are some tools commonly used for storyboarding.

A **PowerPoint** presentation is just a slide show. Each slide shows a fixed screenshot, which you can draw in a paint program and import, or which you can draw directly in PowerPoint. A PowerPoint storyboard doesn't have to be linear slide show. You can create hyperlinks that jump to any slide in the presentation.

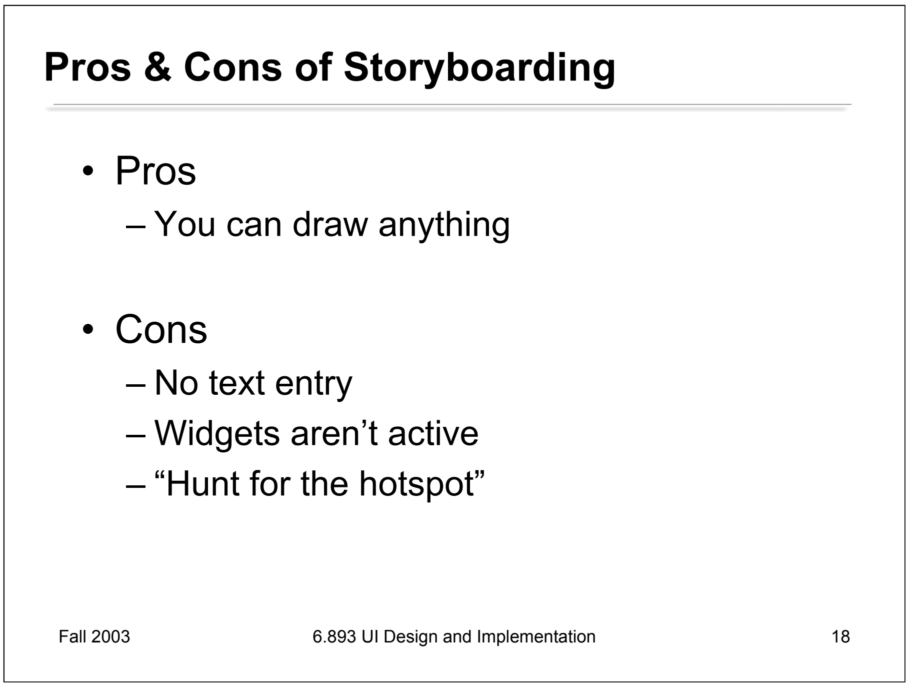
Macromedia **Flash** (formerly Director) is a tool for constructing multimedia interfaces. It's particularly useful for prototyping interfaces with rich animated feedback.

HTML is also useful for storyboarding. Each screen is an imagemap. Macromedia Dreamweaver makes it easy to build HTML imagemaps.

All these tools have scripting languages – PowerPoint has Basic, Flash/Director has a language called Lingo, and HTML has Javascript – so you can write some code to orchestrate transitions, if need be.

If your storyboards need standard widgets like buttons or text boxes, you can create some widgets in a form builder and take static screenshots of them for your storyboard.

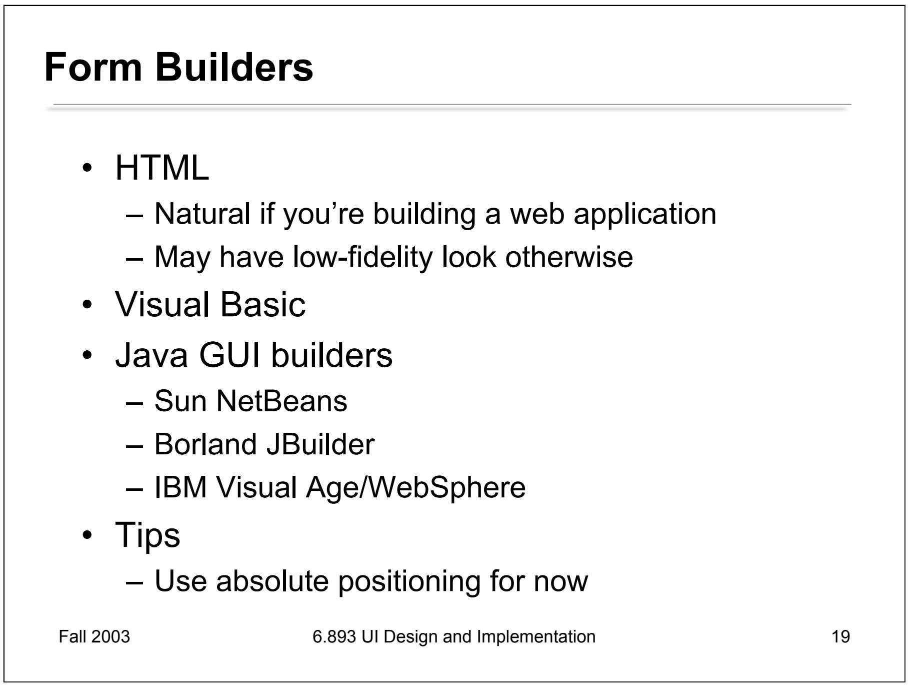
You can find Flash, Director, and Dreamweaver installed in MIT's New Media Center (search for it in Google), a cluster of Macs on the first floor of building 26. The room is sometimes used for classes during the day, but is open to the MIT community at other times.


Pros & Cons of Storyboarding

- Pros
  - You can draw anything
- Cons
  - No text entry
  - Widgets aren't active
  - "Hunt for the hotspot"

The big advantage of storyboarding is similar to the advantage of paper: you can draw anything on a storyboard. That frees your creativity in ways that a form builder can't, with its fixed palette of widgets.

The disadvantages come from the storyboard's static nature. All you can do is click, not enter text. You can still have text boxes, but clicking on a text box might make its content magically appear, without the user needing to type anything. Similarly, scrollbars, list boxes, and buttons are just pictures, not active widgets. Watching a real user in front of a storyboard often devolves into a game of **"hunt for the hotspot",** like children's software where the only point is to find things on the screen to click on and see what they do. The hunt-for-the-hotspot effect means that storyboards are largely useless for user testing, unlike paper prototypes. In general, horizontal computer prototypes are better evaluated with other techniques, like heuristic evaluation.



Form Builders

- HTML
  - Natural if you're building a web application
  - May have low-fidelity look otherwise
- Visual Basic
- Java GUI builders
  - Sun NetBeans
  - Borland JBuilder
  - IBM Visual Age/WebSphere
- Tips
  - Use absolute positioning for now

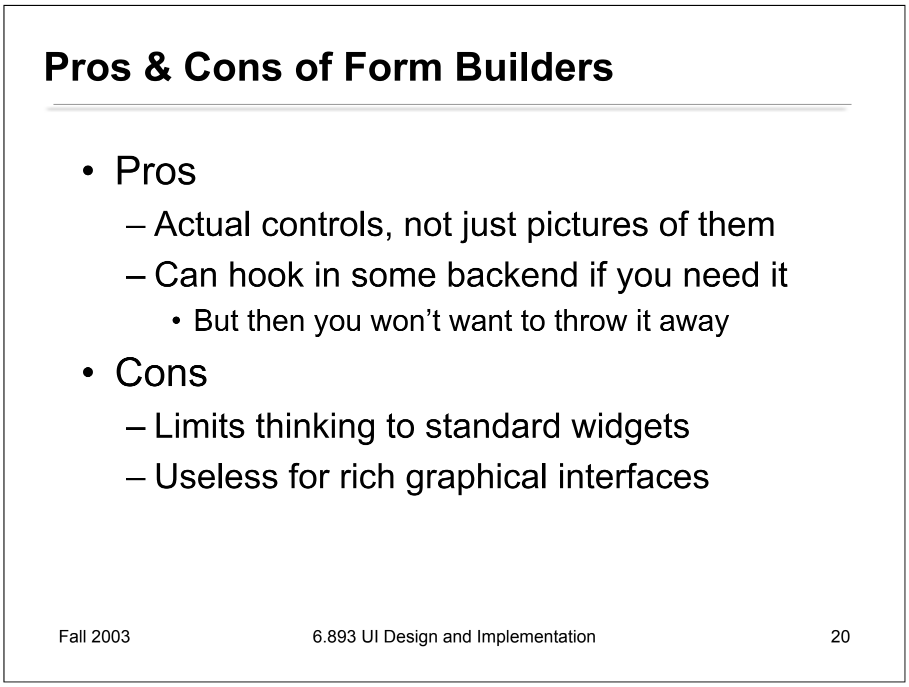
Fall 2003 6.893 UI Design and Implementation 19

Here are some form builder tools.

HTML is a natural tool to use if you're building a web application. You can compose static HTML pages simulating the dynamic responses of your web interface. Although the responses are canned, your prototype is still better than a storyboard, because its screens are more active than mere screenshots: the user can actually type into form fields, scroll through long displays, and see mouse-over feedback. Even if you're building a desktop or handheld application, HTML may still be useful. For example, you can mix static screenshots of some parts of your UI with HTML form widgets (buttons, list boxes, etc) representing the widget parts. It may be hard to persuade HTML to render a desktop interface in a high-fidelity way, however.
Visual Basic is the classic form builder. Many custom commercial applications are built entirely with Visual Basic.

There are several form builders for Java. Sun NetBeans and Borland JBuilder (Personal Edition) can be downloaded free.

Be careful when you're using a form builder for prototyping to avoid layout managers when you're doing your initial graphic designs. Instead, use **absolute positioning**, so you can put each component where you want it to go. Visual Basic's only option is absolute positioning (which is fine for prototyping, but a serious drawback for building real interfaces that need resizable windows). Java GUI builders may need to be told not to use a layout manager.



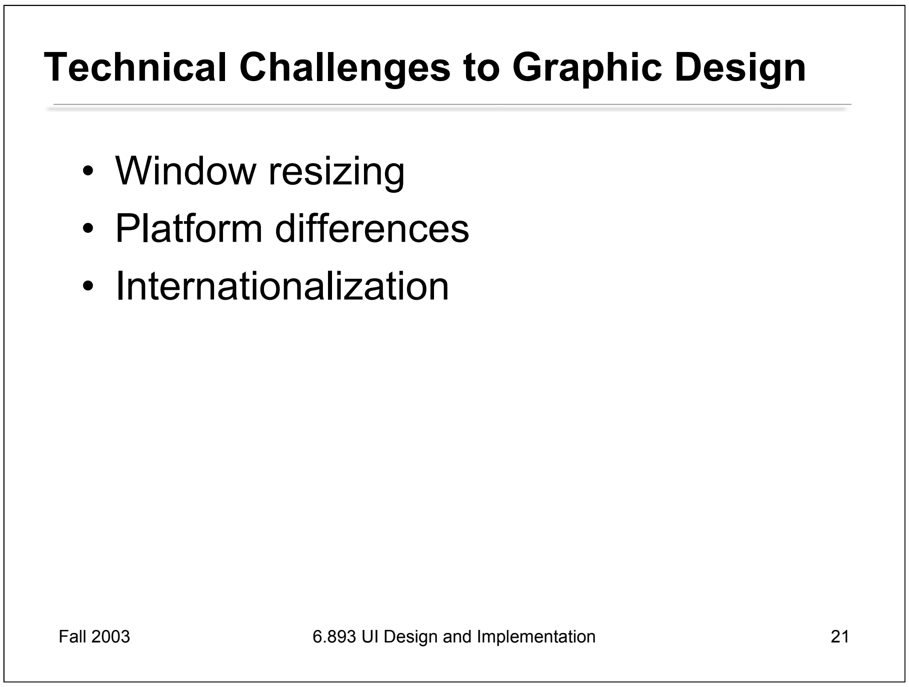
Pros & Cons of Form Builders

- Pros
  - – Actual controls, not just pictures of them
  - – Can hook in some backend if you need it
    - But then you won’t want to throw it away
- Cons
  - – Limits thinking to standard widgets
  - – Useless for rich graphical interfaces

Unlike storyboards, form builders use actual working widgets, not just static pictures. So the widgets look the same as they will in the final implementation (assuming you’re using a compatible form builder – a prototype in Visual Basic may not look like a final implementation in Java).

Also, since form builders usually have an implementation language underneath them – which may even be the same implementation language that you’ll eventually use for your final interface -- you can also hook in as much or as little backend as you want.

On the down side, form builders give you a fixed palette of standard widgets, which limits your creativity as a designer, and which makes form builders largely useless for prototyping rich graphical interfaces, e.g., a circuit-drawing editor. Form builders are great for the menus and widgets that surround a graphical interface, but can’t simulate the “insides” of the application window.



Technical Challenges to Graphic Design

- Window resizing
- Platform differences
- Internationalization

We didn’t have a chance to discuss these issues in the graphic design lecture, so now is as good a place as any to mention them. Graphic design in the print medium doesn’t have these problems; they’re consequences of the dynamic, portable nature of computer software.

On a computer screen, windows can be **resized** – in fact, for good usability, windows *should* be resizable. Some web sites don’t seem to realize this fact, and design for exactly one size. Some UI development tools, like Visual Basic, actually encourage this behavior: Visual Basic windows are unresizable by default, and a VB programmer has to do a tremendous amount of work to make the window resizable.

Unless you plan for it, window resizing can wreak havoc on a careful graphic design, by adding or removing white space, forcing lines to reflow, changing alignment, disrupting balance and symmetry. Fortunately, automatic layout managers can make it possible to provide *reasonable* layout even in the face of window resizing. But your interface should look its best at its default size, since many users won’t resize it at all.

So there are two lessons here: (1) design your interface well at its default size; then (2) make it resizable, even if the default look suffers a bit. Many programmers do (2) without thinking about (1), and many designers just do (1) without bothering with (2).

Platform differences, such as widget appearance, available fonts, screen resolution, and screen size, have similar effects on graphic design, and a similar lesson can be drawn: design for one platform first, then make it look as good as possible on all platforms.

Internationalization also causes problems, particularly translating labels into different languages. Components that fit easily into a screen layout in English may not fit in a language with longer words, like German. Other languages don’t read left-to-right. Internationalization must be planned for in advance, not left until the end, since any message that the user might see must be abstracted out of the source code into a resource file that can be easily substituted for different internationalized versions.