Lecture 12: Computer Prototyping

UI Hall of Fame or Shame?

To see this image, go to
http://images.google.com/images?q=color+easycd2.gif

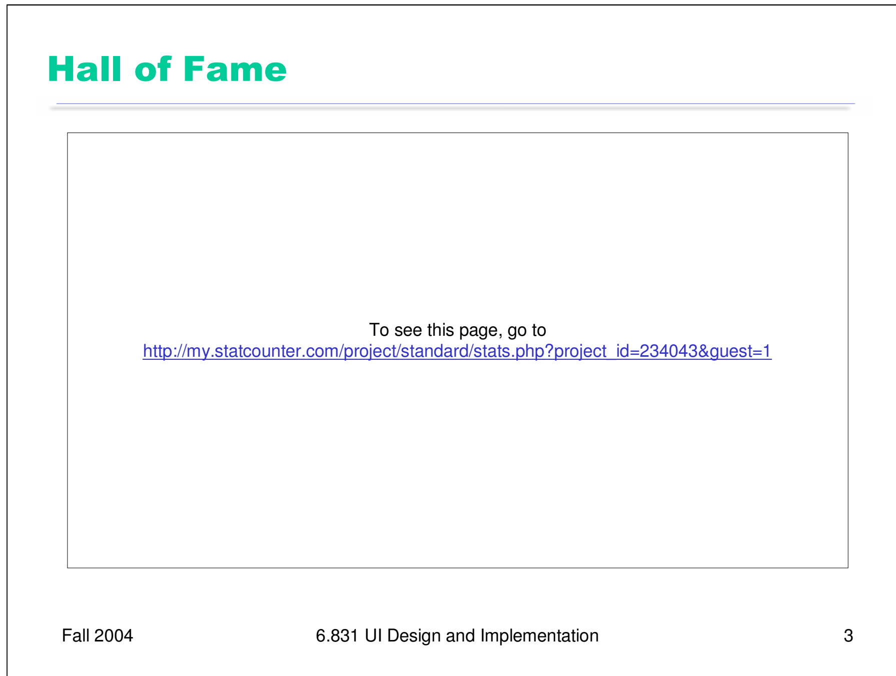
To see this image, go to
http://images.google.com/images?q=color+easycd1.gif

Our Hall of Shame candidate for the day is this dialog box from Adaptec Easy CD Creator, which appears at the end of burning a CD. The top image shows the dialog when the CD was burned successfully; the bottom image shows what it looks like when there was an error.

The key problem is the **lack of contrast** between these two states. Success or failure of CD burning is important enough to the user that it should be obvious at a glance. But these two dialogs look identical at a glance. How can we tell? Use the **squint test**, which we talked about in the graphic design lecture. When you're squinting, you see some labels, a big filled progress bar, a roundish icon with a blob of red, and three buttons. All the details, particularly the text of the messages and the exact shapes of the red blob, are fuzzed out. This simulates what a user would see at a quick glance, and it shows that the graphic design doesn't convey the contrast.

One improvement would change the check mark to another color, say green or black. Using red for OK seems **inconsistent** with the real world, anyway. But designs that differ only in red and green wouldn't pass the squint test for color-blind users.

Another improvement might remove the completed progress bar from the error dialog, perhaps replacing it with a big white text box containing a more detailed description of the problem. That would clearly pass the squint test, and make errors much more noticeable.



Here's our hall of fame example. StatCounter is a web site that tracks usage statistics of your web site, using a hit counter. This is a sample of its statistics page.

The first thing to note is the **simplicity** of the design. Only a few colors are used: gray, black, and a few shades of blue, predominately. This simplicity allows the few different colors – like the red TIP – to really stand out. The design also omits unnecessary labels – for example, the date entry boxes on the bottom don't need labels like "From:" and "To:", because they're self-describing.

It's interesting to look at how **visual variables** were used to encode the information. Position is used to represent date (horizontally) and number of hits (vertically). The kind of statistic is encoded in the value of the graph line, ranging from dark blue (returning visitors) to light blue (page loads). The statistics are actually related in a hierarchy: since every returning visitor is a unique visitor, and every unique visitor causes at least one page load, it is always the case that page loads > unique visitors > returning visitors. This hierarchy is emphasized both by position (the curves are always in the same order vertically) and by value. Position and value were good choices for emphasizing the ordering, because both variables are ordered. An unordered visual variable, like the shape of the data point, would not have been as effective.

This page does have some problems. One is the use of two different terms, "range" and "period", which basically mean the same thing (**internal consistency**). The Set Period interface is in fact a list of common **shortcuts**, like "the last 30 days", which is a good thing; but the shortcuts should be presented more prominently. There's no reason why the year field (2004) should be a text box, rather than a drop-down with choices appropriate to the actual range of data available (**error prevention**). And the hyphen between the start date and the end date is too small to have good **contrast** with the controls around it; it disappears.

Today's Topics

- Quiz preview
- Balance
- Alignment
- Color guidelines
- Computer prototyping

Quiz on Wednesday

- Topics
  - L1: usability
  - L2: user-centered design, user & task analysis
  - L3: MVC, observer, view hierarchy
  - L4: perception, cognition, motor, memory, attention, vision
  - L5: component, stroke & pixel models, redraw, double-buffering
  - L6: interface styles, direct manipulation, affordances natural mapping, visibility, feedback
  - L8: Nielsen's heuristics
  - L9: paper prototyping, fidelity, look/feel, depth/breadth, Wizard of Oz
- Everything is fair game
  - Class discussion, lecture notes, readings, assignments
- Closed book exam, 80 minutes

Balance & Symmetry

- Choose an axis (usually vertical)
- Distribute elements equally around the axis
  - Equalize both mass and extent

Balance and symmetry are valuable tools in a designer's toolkit. In graphic design, symmetry rarely means exact, mirror-image equivalence. Instead, what we mean by symmetry is more like balance: is there the same amount of stuff on each side of the axis of symmetry. We measure "stuff" by both mass (quantity of nonwhite pixels) and extent (area covered by those pixels); both mass and extent should be balanced.

Symmetry Example

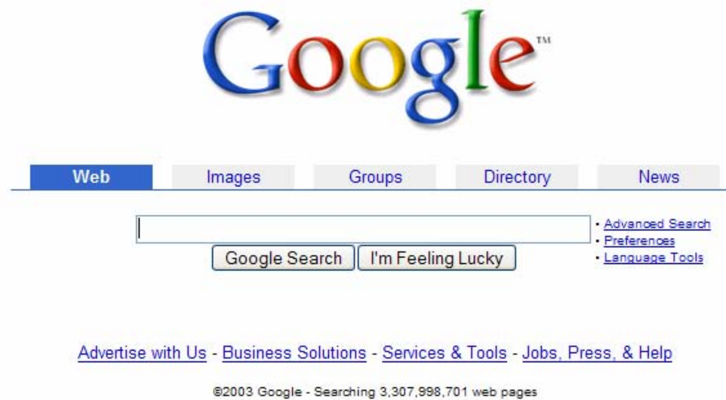


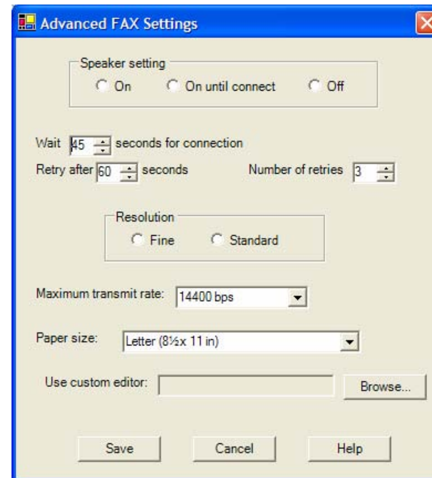
Image courtesy of Google. Used with permission.

An easy way to achieve balance is to simply center the elements of your display. That automatically achieves balance around a vertical axis. If you look at Google's home page, you'll see this kind of approach in action. In fact, only one element of the Google home page breaks this symmetry: the stack of links for Advanced Search, Preferences, and Language Tools on the right. This slight irregularity actually helps emphasize these links slightly.

Symmetry is a kind of **simplicity**; asymmetry creates a **contrast**. Use that contrast wisely.

Alignment

- Align labels on left or right
- Align controls on left *and* right
  - Expand as needed
- Align text baselines



Finally, simplify your designs by aligning elements horizontally and vertically. Alignment contributes to the simplicity of a design. Fewer alignment positions means a simpler design. The dialog box shown has totally haphazard alignment, which makes it seem more complicated than it really is.

Labels (e.g. “Wait” and “Retry after”). There are two schools of thought about label alignment: one school says that the left edges of labels should be aligned, and the other school says that their right edges (i.e., the colon following each label) should be aligned. Both approaches work, and experimental studies haven’t found any significant differences between them. Both approaches also fail when long labels and short labels are used in the same display. You’ll get best results if you can make all your labels about the same size, or else break long labels into multiple lines.

Controls (e.g., text fields, combo boxes, checkboxes). A column of controls should be aligned on both the left and the right. Sometimes this seems unreasonable -- should a short date field be expanded to the same length as a filename? It doesn’t hurt the date to be larger than necessary, except perhaps for reducing its perceived affordance for receiving a date. You can also solve these kinds of problems by rearranging the display, moving the date elsewhere, although be careful of disrupting your design’s functional grouping or the expectations of your user.

So far we’ve only discussed left-to-right alignment. Vertically, you should ensure that labels and controls on the same row share the same **text baseline**. Java Swing components are designed so that text baselines are aligned if the components are centered vertically with respect to each other, but not if the components’ tops or bottoms are aligned. Java AWT components are virtually impossible to align on their baselines. The dialog shown here has baseline alignment problems, particularly between the combo boxes and their captions.

Grids Are Effective

Image found in:

K. Mullet and D. Sano, "Designing visual interfaces: Communication-oriented techniques,“ Sunsoft Press, Prentice Hall (1995), p. 165.

A **grid** is one effective way to achieve both alignment and balance, nearly automatically. Notice the four-column grid used in this dialog box (excluding the labels on the left). The only deviation from the grid is the row of three command buttons at the bottom which are nevertheless still balanced. In fact, their deviation from the grid helps to set them off, despite the minimal white space separating them from the rest of the display.

One criticism of this dialog is false grouping. The controls for Size, All Caps, and Superscript tend to adhere because their proximity, and likewise for the next two rows of the display. This might be fixed by pushing the toggle buttons further to the right, to occupy columns 3 and 4 instead of 2 and 3, but at the cost of some balance.

Color Guidelines

- Remember limitations of human vision
  - Color blindness, red-on-blue, small blue details
- Use few colors
- Avoid saturated colors
- Be consistent and match expectations

Fall 2004 6.831 UI Design and Implementation 10

We've already talked a lot about the limits of human color vision. In general, colors should be used sparingly. An interface with many colors appears more complex, more cluttered, and more distracting. Use only a handful of colors.

Background colors should establish a good contrast with the foreground. White is a good choice, since it provides the most contrast; but it also produces bright displays, since our computer displays emit light rather than reflecting it. Pale (desaturated) yellow and very light gray are also good background colors.

In general, avoid strongly saturated colors – i.e., the colors around the top edge of the HSV cone. Saturated colors can cause visual fatigue because the eye must keep refocusing on different wavelengths. They also tend to saturate the viewer's receptors (hence the name). One study found that air traffic controllers who viewed strongly saturated green text on their ATC interfaces for many hours had trouble seeing pink or red (the other end of the red/green color channel) for up to 15 minutes after their shift was over.

Use less saturated versions instead, pushing them towards gray.

To sharpen contrasts, you can use opponent colors: red/green, blue/yellow. But keep color blind users in mind; hue should not be the *only* way you establish the contrast. Both color-blind and color-normal users will see the contrast better if you vary both hue *and* value.

Finally, match expectations. One of the problems with the Adaptec dialogs at the beginning of this lecture was the use of red for OK. Red generally means stop, warning, error, or hot. Green conventionally means go, or OK. Yellow means caution, or slow.

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

We now turn to prototyping. 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

Fall 2004 6.831 UI Design and Implementation 14

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

- HTML
  - image maps
- Flash/Director
  - animation + actions
- PowerPoint
  - images + links + animation
- 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"

Fall 2004 6.831 UI Design and Implementation 17

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 pages and forms
  - Natural if you're building a web application
  - May have low-fidelity look otherwise
- Java GUI builders
  - Sun NetBeans
  - Eclipse Visual Editor
  - Borland JBuilder
- Other GUI builders
  - Visual Basic, .NET Windows Forms
  - Mac Interface Builder
  - Qt Designer
- Tips
  - Use absolute positioning for now

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. 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

Fall 2004 6.831 UI Design and Implementation 19

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.