L17: Layout
Today's Hall of Fame and Shame is a comparison of two generations of Google Advanced Search. This is the old interface.
And this is the new interface.

Let’s compare and contrast these two interfaces in terms of:

- visibility (specifically self-disclosure)
- graphic design
- task analysis
- efficiency
Today’s Topics

- Layout principles
  - Grouping & hierarchy
  - Whitespace
  - Balance & symmetry
  - Alignment & grids
- Layout implementation
  - Box, flow, grid layouts
  - Margins & padding
  - Space-filling & alignment rules
The power of white space for grouping derives from the Gestalt principle of proximity. These principles, discovered in the 1920’s by the Gestalt school of psychologists, describe how early visual processing groups elements in the visual field into larger wholes. Here are the six principles identified by the Gestalt psychologists:

**Proximity.** Elements that are closer to each other are more likely to be grouped together. You see four vertical columns of circles, because the circles are closer vertically than they are horizontally.

**Similarity.** Elements with similar attributes are more likely to be grouped. You see four rows of circles in the Similarity example, because the circles are more alike horizontally than they are vertically.

**Continuity.** The eye expects to see a contour as a continuous object. You primarily perceive the Continuity example above as two crossing lines, rather than as four lines meeting at a point, or two right angles sharing a vertex.

**Closure.** The eye tends to perceive complete, closed figures, even when lines are missing. We see a triangle in the center of the Closure example, even though its edges aren’t complete.

**Area.** When two elements overlap, the smaller one will be interpreted as a figure in front of the larger ground. So we tend to perceive the Area example as a small square in front of a large square, rather than a large square with a hole cut in it.

**Symmetry.** The eye prefers explanations with greater symmetry. So the Symmetry example is perceived as two overlapping squares, rather than three separate polygons.

Grouping & Hierarchy

- Group together related items
- Make a hierarchy of importance among items
White space plays an essential role in composition. Screen real estate is at a premium in many graphical user interfaces, so it’s a constant struggle to balance the need for white space against a desire to pack information and controls into a display. But insufficient white space can have serious side-effects, making a display more painful to look at and much slower to scan.

Put **margins** around all your content. Labels and controls that pack tightly against the edge of a window are much slower to scan. When an object is surrounded by white space, keep a sense of proportion between the object (the **figure**) and its surroundings (**ground**). Don’t crowd controls together, even if you’re grouping the controls. Crowding inhibits scanning, and produces distracting effects when two lines (such as the edges of text fields) are too close. Many UI toolkits unfortunately encourage this crowding by packing controls tightly together by default, but Java Swing (at least) lets you add empty margins to your controls that give them a chance to breathe.

**White Space**

- Use white space for grouping, instead of lines
- Use margins to draw eye around design
- Integrate figure and ground
  - Object should be scaled proportionally to its background
- Don’t crowd controls together
  - Crowding creates spatial tension and inhibits scanning
Here’s an example of an overcrowded dialog. The dialog has no margins around the edges; the controls are tightly packed together; and lines are used for grouping where white space would be more appropriate. Screen real estate isn’t terribly precious in a transient dialog box.

The crowding leads to some bad perceptual effects. Lines appearing too close together – such as the bottom of the Spacing text field and the group line that surround it – blend together into a thicker, darker line, making a wart in the design. A few pixels of white space between the lines would completely eliminate this problem.
A particularly effective use of white space is to put labels in the left margin, where the white space sets off and highlights them. In dialog box (a), you can’t scan the labels and group names separately; they interfere with each other, as do the grouping lines. In the redesigned dialog (b), the labels are now alone on the left, making them much easier to scan.

For the same reason, you should put labels to the left of controls, rather than above.
Here’s an interesting idea from Tufte: get rid of the grid rules on a standard bar chart, and use whitespace to show where the grid lines would cross the bars. It’s much less noisy. (But alas, impossible to do automatically in Excel.)
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.

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 (a kind of contrast) actually helps emphasize these links slightly.
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 among the controls in the last row: the checkbox “Use custom editor”, the text field, and the Browse button.
A **grid** is one effective way to achieve both alignment and balance, nearly automatically. A grid means that you divide the user interface into equal-width columns (separated by gaps, and with margins on both sides of the window), and align content and controls on the column boundaries. Some elements may span multiple columns, but they align (start or end at) column boundaries.

Newspapers are famous for designing with grids, but if you look carefully at magazines, posters, and many other print designs, you’ll often see a grid guiding the design.
Here’s a grid in a user interface. 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 of 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.
LAYOUT IMPLEMENTATION
In HTML/CSS, automatic layout is a declarative process. First you specify the graphical objects that should appear in the window, which you do by creating instances of various objects and assembling them into a view tree. We’ve seen how HTML does this. Then you specify how they should be laid out by attaching styles.

You can contrast this to a procedural approach to layout, in which you write Javascript code that computes positions and sizes of objects in the view tree.
Here are the two key reasons why we like automatic layout – and these two reasons generalize to other forms of declarative UI as well.

First, it makes programming easier. The code that sets up layout managers is usually much simpler than procedural code that does the same thing.

Second, the resulting layout can respond to change more readily. Because it is generated automatically, it can be regenerated any time changes occur that might affect it. One obvious example of this kind of change is resizing the window, which increases or decreases the space available to the layout. You could handle window resizing with procedural code as well, of course, but the difficulty of writing this code means that programmers generally don’t. (That’s why many Windows dialog boxes, which are often laid out using absolute coordinates in a GUI builder, refuse to be resized! A serious restriction of user control and freedom, particularly if the dialog box contains a list or file chooser that would be easier to use if it were larger.)

Automatic layout can also automatically adapt to font size changes, different widget sets (e.g., buttons of different size, shape, or decoration), and different labels (which often occur when you translate an interface to another language, e.g. English to German). These kinds of changes tend to happen as the application is moved from one platform to another, rather than dynamically while the program is running; but it’s helpful if the programmer doesn’t have to worry about them.

Another dynamic change that automatic layout can deal with is the appearance or disappearance of nodes from the view tree-- if the user is allowed to add or remove buttons from a toolbar, for example, or if new textboxes can be added or removed from a search query.
Flow Layout

- Left-to-right, automatically-wrapping
- CSS calls this “inline” layout
display: inline
- Many elements use inline layout by default

<button>People</button>
<button>Places</button>
<button>Things</button>
<button>New</button>
<button>Save</button>
<button>Print</button>
...
Box Layout

- Blocks are laid out vertically
  - display: block
  - divs default to block layout
- Inline blocks are laid out in flow
  - display: inline-block

```html
<div>
  <button>People</button>
  <button>Places</button>
  <button>Things</button>
</div>

<div>
  <button>New</button>
  <button>Save</button>
  <button>Print</button>
</div>
```
Float Layout

- Float pushes a block to left or right edge

```html
<style>
.navbar { float: left; }
.navbar button { display: block; }
</style>

<div class="navbar">
<button>People</button>
<button>Places</button>
<button>Things</button>
</div>

<div>
<button>New</button>
<button>Save</button>
<button>Print</button>
</div>
```
**Grid Layout**

- Blocks & floats are typically not enough to enforce all the alignments you want in a UI
- CSS grid layout is coming but not quite here
- For now, use tables for 2D alignment instead

```html
<table>
<tr><td>Name:</td> <td><input type="text" /></td></tr>
<tr><td>Groups:</td> <td><textarea></textarea></td></tr>
</table>
```
Absolute Positioning

- Setting position & size explicitly
  - in coordinate system of entire window, or of node's parent
  - CSS has several units: px, em, ex, pt
  - mostly useful for popups

```html
<style>
  button { position: absolute;
           left: 5px;
           top: 5px; }
</style>
```

Print the user interface goes here
Margins, Borders, & Padding

Margin
Border
Padding
Content
Space-Filling & Alignment

- `width: 100%`, `height: 100%` consumes all of the space available in the parent
- `vertical-align` moves a node up and down in its parent’s box
  - baseline is good for lining up labels with textboxes
  - top and bottom are useful for other purposes
- `Centering`
  - `margin: auto` for boxes
  - `text-align: center` for inlines
Summary

- Layout should establish grouping of items
- Use whitespace & alignment to preserve simplicity
- Automatic layout adapts to changes in UI