

Lecture 16: Constraints & Layout

Spring 2008

6.831 User Interface Design and Implementation

1

UI Hall of Fame or Shame?



Spring 2008

6.831 User Interface Design and Implementation

2

This example was suggested by Dennis Ramdass. Doodle is a web site for conducting small polls, which is most commonly used to schedule an event by asking a group of people to mark their availability.

The interface for entering your choices is shown on the left – you enter your name in the textbox, click on the checkboxes to mark your availability, and then click Participate.

The editing interface is shown on the right. You click the Edit An Entry or Delete An Entry link (shown at the top), and then the page displays Edit and Delete links next to individual entries (shown at the bottom).

Let's discuss this interface from the perspective of:

- learnability
- efficiency
- graphic design
- user control & freedom
- error prevention

Today's Topics

- Automatic layout
- Layout managers
- Constraints

Spring 2008

6.831 User Interface Design and Implementation

3

Today's lecture is about **automatic layout** – determining the positions and sizes of UI components. Automatic layout is a good example of declarative user interface specification. The programmer specifies what kind of layout is desired by attaching properties or layout managers to the view hierarchy, and then an automatic algorithm (layout propagation) actually computes the layout.

We'll also talk about **constraints**, which is a rather low-level, but also declarative, technique for specifying layout. Constraints are useful for more than just layout; unfortunately most GUI toolkits don't have a general-purpose constraint solver built in. But constraints are nevertheless a useful way to think about relationships in a user interface declaratively, even if you have to translate them to procedural code yourself.

In Java, 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 component hierarchy. Then you specify how they should be related to each other, by attaching a layout manager to each container.

You can contrast this to a procedural approach to layout, in which you actually write Java or Javascript code that computes positions and sizes of graphical objects. (You probably wrote a lot of this kind of code in the checkerboard output assignment, for example, to compute where each checker should appear on the screen.)

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

Automatic Layout

- **Layout** determines the sizes and positions of components on the screen
 - Also called geometry in some toolkits
- Declarative layout
 - Java: layout managers
 - CSS: layout styles
- Procedural layout
 - Write code to compute positions and sizes

Spring 2008

6.831 User Interface Design and Implementation

4

Reasons to Do Automatic Layout

- Higher level programming
 - Shorter, simpler code
- Adapts to change
 - Window size
 - Font size
 - Widget set (or theme or skin)
 - Labels (internationalization)
 - Adding or removing components

Spring 2008

6.831 User Interface Design and Implementation

5

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 components -- 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.

Layout Manager Approach

- Layout manager performs automatic layout of a container's children
 - 1D (BoxLayout, FlowLayout, BorderLayout)
 - 2D (GridLayout, GridBagLayout, TableLayout)
- Advantages
 - Captures most common kinds of layout relationships in reusable, declarative form
- Disadvantages
 - Can only relate **siblings** in component hierarchy

Spring 2008

6.831 User Interface Design and Implementation

6

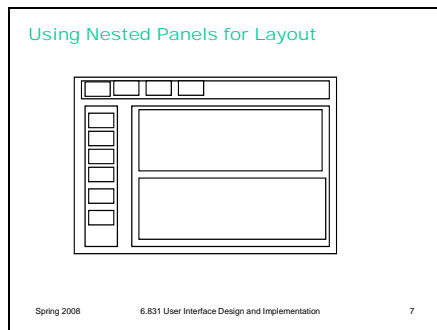
Let's talk specifically about the layout-manager approach used in Java, which evolved from earlier UI toolkits like Motif and Tcl/Tk. A **layout manager** is attached to a container, and it computes the positions and sizes of that container's children. There are two basic kinds of layout managers: one-dimensional and two-dimensional.

One-dimensional layouts enforce only one direction of alignment between the components; for example, `BoxLayout` aligns components along a line either horizontally or vertically. `BorderLayout` is also one-dimensional: it can align components along any edge of the container, but the components on different edges aren't aligned with each other at all.

Two-dimensional layouts can enforce alignment in two directions, so that components are lined up in rows and columns. 2D layouts are generally more complicated to specify (totally `GridBag`!), but we'll see in the Graphic Design lecture that they're really essential for many dialog box layouts, in which you want to align captions and fields both horizontally

and vertically at the same time.

Layout managers are a great tool because they capture the most common kinds of layout relationships as reusable objects. But a single layout manager can make only **local decisions**: that is, it computes the layout of **only one** container's children, based on the space available to the container. So they can only enforce relationships between **siblings** in the component hierarchy. For example, if you want all the buttons in your layout to be the same size, a layout manager can only enforce that if the buttons all belong to the same parent. That's a difference from the more general constraint system approach to layout that we'll see later in this lecture. Constraints can be global, cutting across the component hierarchy to relate different components at different levels.



Another common trick in layout is to introduce new containers (divs in HTML, JPanels in Java) in the component hierarchy, just for the sake of layout. This makes it possible to use one-dimensional layout managers more heavily in your layout. Suppose this example is Swing. A `BorderLayout` might be used at the top level to arrange the three topmost panels (toolbar at top, palette along the left side, and main panel in the center), with `BoxLayout`s to layout each of those panels in the appropriate direction.

This doesn't eliminate the need for two-dimensional layout managers, of course. Because a layout manager can only relate one container's children, you wouldn't be able to enforce simultaneous alignments between captions and fields, for example. Using nested panels with one-dimensional layouts would force you to put them into separate containers.

Basic Layout Propagation Algorithm

```
computePreferredSize(Container parent)
for each child in parent,
  computePreferredSize(child)
compute parent's preferred size from children
  e.g., horizontal layout,
  (prefwidth,pretheight) = (sum(children prefwidth),
                           max(children pretheight))

layout(Container parent) requires: parent's size already set
apply layout constraints to allocate space for each child
  child.(width,height) = (parent.width / #children, parent.height)
set positions of children
  child[j].(x,y) = (child[j-1].x+child[j-1].width, 0)
for each child in parent,
  layout(child)
```

Spring 2008

6.831 User Interface Design and Implementation

8

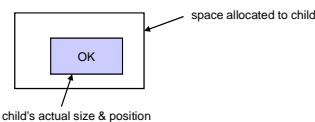
Since the component hierarchy usually has multiple layout managers in it (one for each container), these managers interact by a **layout propagation** algorithm to determine the overall layout of the hierarchy.

Layout propagation has two phases.

First, the size requirements (**preferred sizes**) of each container are calculated by a **bottom-up** pass over the component hierarchy. The leaves of the hierarchy – like labels, buttons, and textboxes – determine their preferred sizes first, by calculating how large a rectangle they need to display to display their text label and surrounding whitespace or decorations. Then each container's layout manager computes its size requirement by combining the desired sizes of its children. The preferred sizes of components are used for two things: (1) to determine an initial size for the entire window, which is what Java's pack() method does; and (2) to allow some components to be fixed to their natural size, rather than trying to expand them or shrink them, and adjust other parts of the layout accordingly.

Once the size of the entire window has been established (either by computing its preferred size, or when the user manually sets it by resizing), the actual layout process occurs **top-down**. For each container in the hierarchy, the layout manager takes the container's assigned size (as dictated by its own parent's layout manager), applies the layout rules to allocate space for each child, and sets the positions and sizes of the children appropriately. Then it recursively tells each child to compute its layout.

How Child Fills Its Allocated Space



Expanding



Padding



Anchoring



Spring 2008

6.831 User Interface Design and Implementation

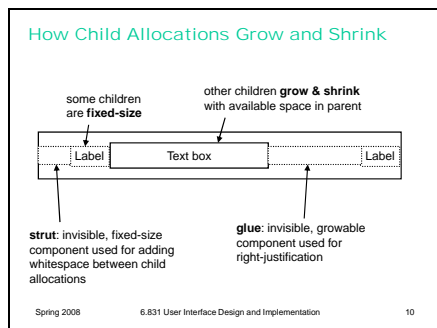
9

Let's talk about a few key concepts in layout managers. First, depending on the layout manager, the **space allocated** to a child by its container's layout manager is not always the same as the **size of the child**. For example, in GridBagLayout, you have to explicitly say that a component should **fill** its space allocation, in either the x or y direction or both (also called **expanding** in other layout managers).

Some layout managers allow some of the space allocation to be used for a margin around the component, which is usually called **padding**. The margin is added to the child's preferred size during the bottom-up size requirements pass, but then subtracted from the available space allocation during the top-down layout pass.

When a child doesn't fill its allocated space, most layout managers let you decide how you want the component to be **anchored** (or aligned) in the space – along a boundary, in a corner, or centering in one or both directions. In a sense, expanding is just anchoring to all four corners of the available space.

Since the boundaries aren't always visible – the button shown here has a clear border around it, but text labels usually don't – you might find this distinction between the space allocation and the component confusing. For example, suppose you want to left-justify a text label within the allocated space. You can do it two ways: (1) by telling the label itself to display left-justified with respect to its own rectangle, or (2) by telling the layout manager to anchor the label to the left side of its space allocation. But method #1 works only if the label is expanded to fill its space allocation, and method #2 works only if the label is **not** expanded. So subtle bugs can result.



Now let's look at how space allocations typically interact. During the top-down phase of the layout process, the container's size is passed down from above, so the layout manager has to do the best it can with the space provided to it. This space may be larger or smaller than the layout's preferred size. So layout managers usually let you specify which of the children are allowed to grow or shrink in response, and which should be fixed at their preferred size. If more than one child is allowed to take up the slack, the layout manager has rules (either built in or user-specified) for what fraction of the excess space should be given to each resizable child.

In Java, growing and shrinking is constrained by two other properties of components: minimum size and maximum size. So one way to keep a component from growing or shrinking is to ensure that its minimum size and maximum size are always identical to its preferred size. But layout managers often have a way to specify it explicitly, as well.

Struts and glue are two handy idioms for inserting whitespace (empty space) into an automatic layout. A **strut** is a fixed-size invisible component; it's used for margins and gaps between components. **Glue** is

an invisible component that can grow and shrink with available space. It's often used to push components over to the right (or bottom) of a layout.


Sometimes the layout manager itself allows you to specify the whitespace directly in its rules, making struts and glue unnecessary. For example, `TableLayout` lets you have empty rows or columns of fixed or varying size. But `BoxLayout` doesn't, so you have to use struts and glue.

Java has factory methods for struts and glue in the `Box` class, but even if struts or glue weren't available in the toolkit, you could create them easily. Just make a component that draws nothing and set its sizes (minimum, preferred, maximum) appropriately.

HTML and CSS Layout

- Left-to-right, wrapping flow is the default
Words in a paragraph (like Swing's `FlowLayout`)
- Absolute positioning in parent coordinate system

```
#B {  
  position: absolute;  
  left: 20px;  
  width: 50%;  
  bottom: 5em;  
}
```



- 2D table layout
`<table>`, `<tr>`, `<td>` (like ClearThought's `TableLayout`)

Spring 2008 6.831 User Interface Design and Implementation 11

CSS layout offers three main layout strategies. The first is the default, left-to-right, wrapping flow typical of web pages. This is what Swing's `FlowLayout` also does.

More useful for UI design is **absolute** positioning, which allows a component's coordinates to be specified either explicitly (in pixels relative to the parent's coordinate system) or relatively (as percentages of the parent). For example, setting a component's left to 50% would put it halfway across its parent's bounding box. Absolute positioning can constrain any two of the coordinates of a component: left, right, and width. (If it specifies all three, then CSS ignores one of them.)

Finally, HTML offers a table layout, which is flexible enough to handle most 2D alignments you'd want. The easiest way to use it is to use the `<table>` element and its family of related elements (`<tr>` for rows, and `<td>` for cells within a row).

Constraints

- **Constraint** is a relationship among variables that is automatically maintained by system
 - **Constraint propagation**: When a variable changes, other variables are automatically changed to satisfy constraint

Spring 2008

6.831 User Interface Design and Implementation

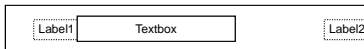
12

Since layout managers have limitations, let's look at a more general form of declarative UI, that can be used not only for layout but for other purposes as well: **constraints**.

A constraint is a relationship among variables. The programmer specifies the relationship, and then the system tries to automatically satisfy it. Whenever one variable in the constraint changes, the system tries to adjust variables so that the constraint continues to be true. Constraints are rarely used in isolation; instead, the system has a collection of constraints that it's trying to satisfy, and a **constraint propagation** algorithm satisfies the constraints when a variable changes.

In a sense, layout managers are a limited form of constraint system. Each layout manager represents a set of relationships among the positions and sizes of the children of a single container; and layout propagation finds a solution that satisfies these relationships.

Using Constraints for Layout



```
label1.left = 5
label1.width = textwidth(label1.text, label1.font)
label1.right = textbox.left
label1.left + label1.width = label1.right

textbox.width >= parent.width / 2
textbox.right <= label2.left

label2.right = parent.width
```

Spring 2008

6.831 User Interface Design and Implementation

13

Here's an example of some constraint equations for layout. This is same layout we showed a couple of slides ago, but notice that we didn't need struts or glue here; constraint equations can do the job instead.

This simple example reveals some of the important issues about constraint systems. One issue is whether the constraint system is **one-way** or **multiway**. One-way constraint systems are like spreadsheets – you can think of every variable like a spreadsheet cell with a formula in it calculating its value in terms of other variables. One-way constraints must be written in the form $X=f(X_1,X_2,X_3,\dots)$. Whenever one of the X_i 's changes, the value of X is recalculated. (In practice, this is often done **lazily** – i.e., the value of X isn't recalculated until it's actually needed.)

Multiway constraints are more like systems of equations -- you could write each one as $f(X_1,X_2,X_3,\dots)=0$. The programmer doesn't identify one variable as the output of the constraint – instead, the system can adjust any variable (or more than one variable) in the equation to make the constraint

become true. Multiway constraint systems offer more declarative power than one-way systems, but the constraint propagation algorithms are far more complex to implement.

One-way constraint systems must worry about **cycles**: if variable X is computed from variable Y, but variable Y must be computed from variable X, how do you compute it? Some systems simply disallow cycles (spreadsheets consider them errors, for example). Others break the cycle by reusing the old (or default) value for one of the variables; so you'll compute variable Y using X's old value, then compute a new value for X using Y.

Conflicting constraints are another problem – causing the constraint system to have no solution. Conflicts can be resolved by **constraint hierarchies**, in which each constraint equation belongs to a certain priority level. Constraints on higher priority levels take precedence over lower ones.

Inequalities (such as `textbox.right <= label2.left`) are often useful in specifying layout constraints, but require more expensive constraint satisfaction algorithms.

Constraints can be used for more general purposes than just layout. Here are a few.

Some forms of **input** can be handled by constraints, if you represent the state of the input device as variables in constraint equations. For example, to drag a checker around on a checkerboard, you constrain its position to the position of the mouse pointer.

Constraints can be very useful for keeping user interface components consistent with each other. For example, a Delete toolbar button and a Delete command on the Edit menu should only be enabled if something is actually selected. Constraints can make this easy to state.

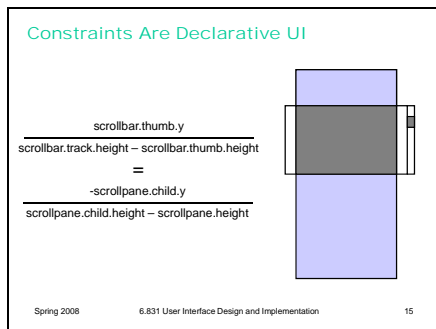
The connection between a view and a model is often easy to describe with constraints, too. (But notice the conflicting constraints in this example! `checker.x` is defined both by the dragging constraint and by the model constraint. Either you have to mix both constraints in the same expression – e.g., if dragging

Using Constraints for Behavior

- Input
 - `checker.(x,y) = mouse.(x,y)`
if `mouse.button1 && mouse.(x,y) in checker`
- Output
 - `checker.dropShadow.visible = mouse.button1 && mouse.(x,y) in checker`
- Interactions between components
 - `deleteButton.enabled = (textbox.selection != null)`
- Connecting view to model
 - `checker.x = board.find(checker).column * 50`

then use the dragging constraint, else use the model constraint – or you have to specify priorities to tell the system which constraint should win.)

The alternative to using constraints in all these cases is writing **procedural code** – typically an event handler that fires when one of the dependent variables changes (like `mouseMoved` for the mouse position, or `selectionChanged` for the textbox selection, or `pieceMoved` for the checkbox position), and then computes the output variable correctly in response. The idea of constraints is to make this code **declarative** instead, so that the system takes care of listening for changes and computing the response.



This example shows how powerful constraint specification can be. It shows how a scrollbar's thumb position is related to the position of the pane that it's scrolling. (The pane's position is relative to the coordinate system of the scroll window, which is why it's *negative*.) Not only is it far more compact than procedural code would be, but it's **multiway**. You can solve this equation for different variables, to compute the position of the scrollpane as a function of the thumb position (in order to respond to the user dragging the thumb), or to compute the thumb position as a function of the pane position (e.g. if the user scrolls the pane with arrow keys or jumps directly to a bookmark). So both remain consistent.

Alas, constraint-based user interfaces are still an area of research, not much practice. Some research UI toolkits have incorporated constraints (Amulet, Arkit, Subarctic, among others), and a few research constraint solvers exist that you can plug in to existing toolkits (e.g., Cassowary). But you won't find constraint systems in most commercial user interface toolkits, except in limited ways. The `SpringLayout` layout manager is the closest thing to a constraint system you can find in standard Java (it suffers from the limitations of all layout managers).

But you can still *think* about your user interface in terms of constraints, and *document your code* that way. You'll find it's easier to generate procedural

code once you've clearly stated **what** you want (declaratively). If you state a constraint equation, then you know which events you have to listen for (any changes to the variables in your equation), and you know what those event handlers should do (solve for the other variables in the equation). Writing procedural code for the scrollpane is much easier if you've already written the constraint relationship.

Summary

- Automatic layout adapts to different platforms and UI changes
- Layout managers are 1D or 2D
- Layout propagation algorithm
- Constraints are useful for more than just layout