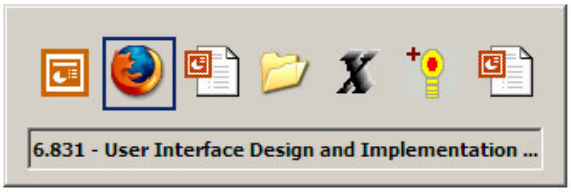
Lecture 10: Declarative UI

UI Hall of Fame or Shame?



Fall 2005 6.831 UI Design and Implementation 2

Today's candidate for the Hall of Fame & Shame is the **Alt-Tab** window switching interface in Microsoft Windows. This interface has been copied by a number of desktop systems, including KDE, Gnome, and even Mac OS X.

The first observation to make is that this interface is designed only for keyboard interaction. Alt-Tab is the only way to make it appear; pressing Tab (or Shift-Tab) is the only way to cycle through the choices. If you try to click on this window with the mouse, it vanishes. The interface is weak on affordances, and gives the user little help in remembering how to use it.

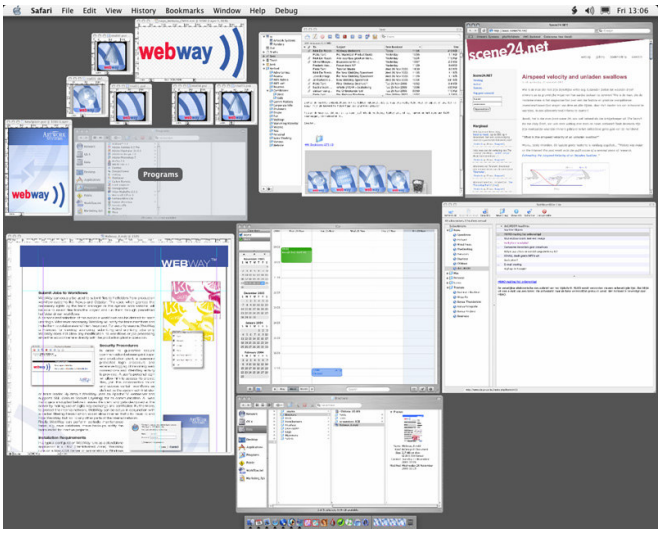
But that's OK, because the Windows taskbar is the primary interface for window switching, providing much better visibility and affordances. This Alt-Tab interface is designed as a **shortcut**, and we should evaluate it as such.

It's pleasantly **simple**, both in graphic design and in operation. Few graphical elements, good alignment, good balance. The 3D border around the window name could probably be omitted without any loss.

This interface is a **mode** (since pressing Tab is switching between windows rather than inserting tabs into text), but it's spring-loaded, happening only as long as the Alt button is held down. This spring-loading also provides good **dialog closure**.

Is it **efficient**? A common error, when you're tabbing quickly, is to overshoot your target window. You can fix that by cycling around again, but that's not as **reversible** as just moving backwards with a mouse. (You can also back up by holding down Shift when you press Tab, but that's not well-communicated by this interface, and it's tricky to negotiate while you're holding Alt down.)

UI Hall of Fame or Shame?



Fall 2005 6.831 UI Design and Implementation 3

For comparison, let's look at the Exposé feature in Mac OS X. When you push F9 on a Mac, it displays all the open windows – even hidden windows, or windows covered by other windows – shrinking them as necessary so that they don't overlap. Mousing over a window displays its title, and clicking on a window brings that window to the front and ends the Exposé mode, sending all the other windows back to their old sizes and locations.

Like Alt-Tab, Exposé is also a **mode**. Unlike Alt-Tab, however, it is not spring-loaded. It depends instead on dramatic visual differences as a mode indicator – with its shrunken, tiled windows, Exposé mode usually looks a lot different than the normal desktop.

To get out of Exposé mode *without* choosing a new window, you can press F9 again, or you can click the window you were using before. That's easier to discover and remember than Alt-Tab's mechanism – pressing Escape. When I use Alt-Tab, and then decide to abort it, I often find myself cycling through all the windows trying to find my original window again. Both interfaces support **user control and freedom**, but Exposé seems to make canceling more **efficient**.

The representation of windows is much richer in Exposé than in Alt-Tab. Rather than Alt-Tab's icons (many of which are identical, when you have several documents open in the same application), Exposé uses the **window itself** as its visual representation. That's much more in the spirit of direct manipulation. (A version of Alt-Tab included in Windows Power Toys shows images of the windows themselves – try it!)

Let's look at efficiency more deeply. Alt-Tab is a very linear interface – to pick an arbitrary window out of the n windows you have open, you have to press Tab O(n) times. Exposé, on the other hand, depends on pointing – so because of Fitts's Law, the cost is more like O(log n). (Of course, this analysis only considers motor movement, not visual search time; it assumes you already know where the window you want is in each interface. But Exposé probably wins on visual search, too, since the visual representation shows the window itself, rather than a frequently-ambiguous icon.)

Quiz on Monday

- Topics
  - L1: usability
  - L2: user-centered design, user & task analysis
  - L3: MVC, observer, view hierarchy
  - L4: component, stroke & pixel models, redraw, double-buffering
  - L5: perception, cognition, motor, memory, vision
  - L6: events, dispatch & propagation, finite state controllers, interactors
  - L7: interface styles, direct manipulation, affordances, mapping, visibility, feedback
  - L8: Nielsen's heuristics
  - L9: paper prototyping, fidelity, look/feel, depth/breadth, computer prototyping, Wizard of Oz
  - L10: automatic layout, layout propagation, constraints, model-based user interfaces
- Everything is fair game
  - Class discussion, lecture notes, readings, assignments
- Closed book exam, 80 minutes

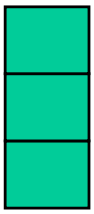
Today's Topics

- Automatic layout
- Constraints
- Model-based UI

Declarative vs. Procedural

- Declarative programming
  - Saying *what* you want
- Procedural programming
  - Saying *how* to achieve it

Declarative
A tower of 3 blocks.



Procedural
1. Put down block A.
2. Put block B on block A.
3. Put block C on block B.

Today we'll be talking about ways to implement user interfaces using higher-level, more abstract specifications – particularly, **declarative specifications**. The key advantage of declarative programming is that you just say **what** you want, and leave it to an automatic tool to figure out how to produce it. That contrasts with conventional procedural programming, where the programmer has to say, step-by-step, how to reach the desired state.

Example: Automatic Layout

- Layout = component positions & sizes
  - Sometimes called geometry
- Declarative layout
  - Declare the components
    - Java: component hierarchy
  - Declare their layout relationships
    - Java: layout managers
- Procedural layout
  - Write code to compute positions and sizes

Our first example of declarative user interface should already be somewhat familiar to you: automatic layout. 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 code that computes positions and sizes of graphical objects. You wrote a lot of this code in the checkerboard assignment, for example.

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

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 generally laid out using absolute coordinates in a GUI builder, refuse to be resized!)

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 Managers

- 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 form
- Disadvantages
  - Can only relate **siblings** in component hierarchy

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.

Layout Propagation

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

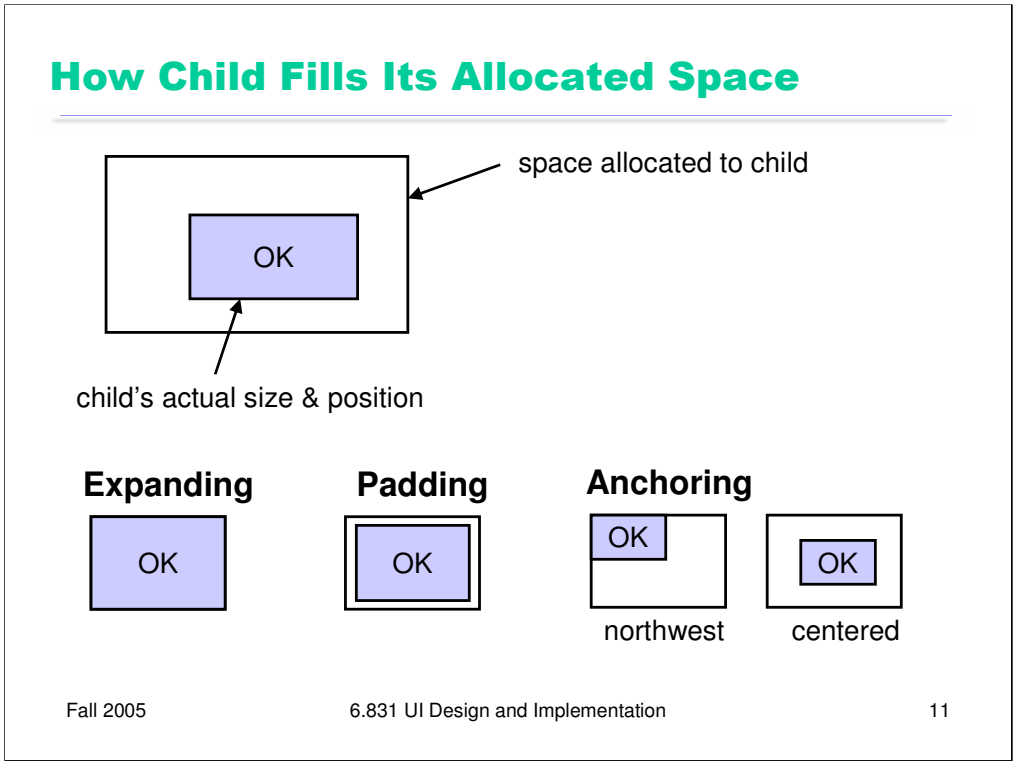
layout(Container parent) requires: parent's size already set
  apply layout constraints to allocate space for each child
    child.(width,height) = (parent.width / #children, parent.height)
  set positions of children
    child[i].(x,y) = (child[i-1].x+child[i-1].width, 0)
  for each child in parent,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

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 parts.

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.

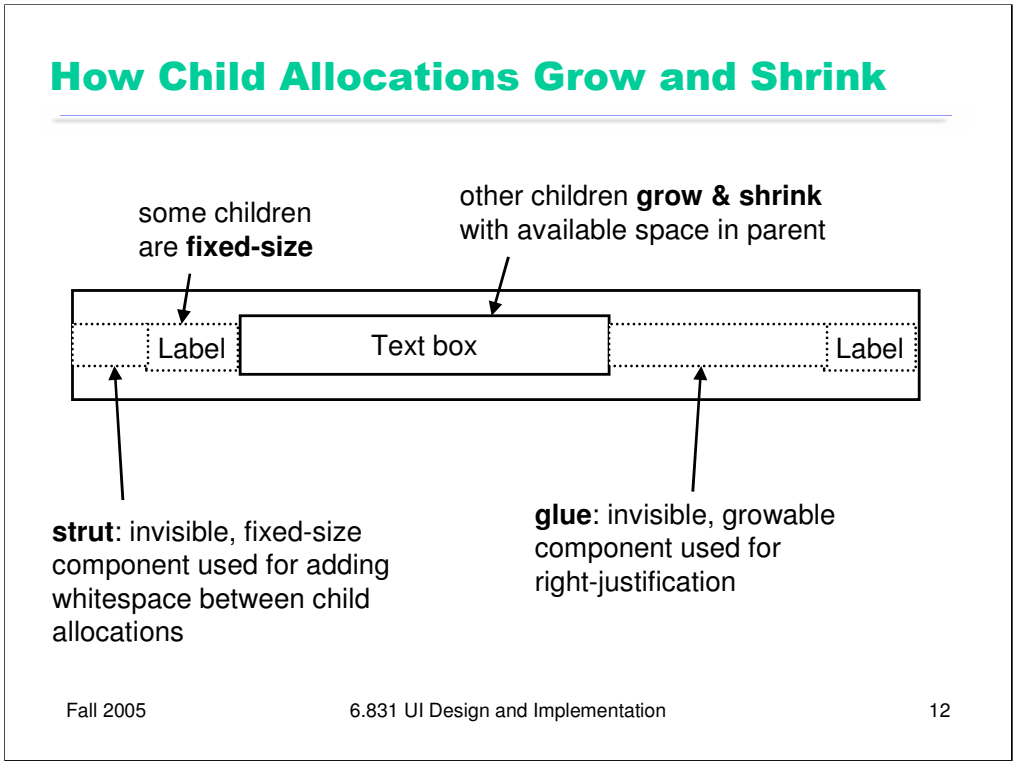


Let's talk about a few key concepts in layout managers. First, depending on the layout manager, the **space allocated** to a child by its container's layout manager is not always the same as the **size of the child**. For example, in GridBagLayout, you have to explicitly say that a component should **fill** 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.

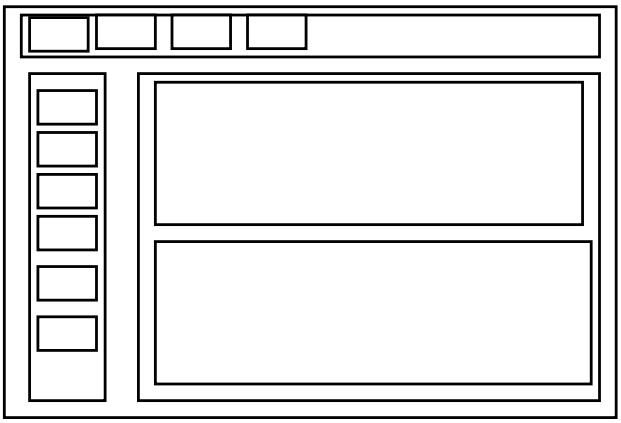
In Java, growing and shrinking is constrained by two other properties of components: minimum size and maximum size. So one way to keep a component from growing or shrinking is to ensure that its minimum size and maximum size are always identical to its preferred size. But layout managers often have a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

Using Nested Panels for Layout



Another common trick in layout managers is to introduce new containers (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. In this example, 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 BoxLayouts 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 enforce simultaneous alignments between captions and fields, for example, because using nested panels with one-dimensional layouts would force you to put them into separate containers.

Constraints

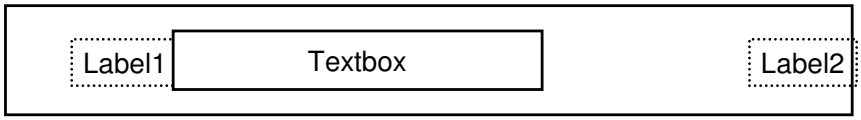
- Constraint = relationship among variables
  - Automatically maintained by system
    - **Constraint propagation:** When a variable changes, other variables are automatically changed to satisfy constraint

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

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(X1,X2,X3,…). Whenever one of the Xi'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(X1,X2,X3,…) = 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

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

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.

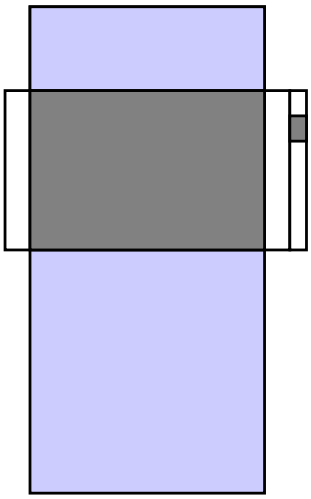
Constraints can be very useful for keeping user interface components consistent with each other. For example, a Delete toolbar button and a Delete command on the Edit menu should only be enabled if something is actually selected. Constraints can make this easy to state.

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 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 checker 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.

Constraints Are Declarative UI

$$\frac{\text{scrollbar.thumb.y}}{\text{scrollbar.track.height} - \text{scrollbar.thumb.height}} = \frac{\text{-scrollpane.child.y}}{\text{scrollpane.child.height} - \text{scrollpane.height}}$$



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 see how moving the thumb should affect the pane, and how moving the pane (e.g. by scrolling with arrow keys or jumping to a bookmark) should affect the thumb, so that 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, Artkit, 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.

Model-Based User Interfaces

- Programmer writes logical model of UI
  - State variables (bool, int, string, list)
  - Commands
- System generates actual presentation
  - Grouping into windows, tabs, panels
  - Widget selection
  - Layout
- Same motivation as other declarative UI
  - Higher-level programming
  - Adapting to change: particularly for devices and users
    - Screen size (watch, phone, PDA, laptop, desktop, wall)
    - Widgets available (phone vs. desktop)
    - Input style (mouse vs. arrow buttons; speech, finger, pen)
    - Output style (speech vs. display)
    - User behavior (uses some components more)

Let's discuss one more example of declarative UI, which is even farther-out than constraint systems. In **model-based user interface**, the goal is to describe the *entire user interface* declaratively, not just one aspect of it like layout or input handling.

The programmer provides a high-level description of the user interface, often called a model, which consists of a set of data variables and commands. A model for a login dialog box, for example, might state that are two string variables (a username and password) and one command (login).

This description is then used to generate a presentation. Aspects of the presentation that must be generated include its **grouping** (how variables and commands are organized); the particular **widgets** selected for each model element (e.g., a string variable might be represented by a textfield, a text area, or a combobox; a command might be a button, menu item, keyboard shortcut, or all three). **Layout** is part of presentation, of course; so are labels for the widgets, font and color choices.

The presentation can't be generated completely automatically, of course – UI design isn't that easy to automate (yet). Generally, the programmer has to provide some presentation specification as well – perhaps completely specified (as in the UIML system we read about) or merely as hints or constraints on legal or desirable presentations (as in the SUPPLE system we read about).

Model-based user interface is driven by the same motivations as other declarative UI: simpler programming, and adapting to change. But the kinds of change that model-based UI can adapt to is broader. Ideally, a model-based UI should be able to generate presentations for a broad variety of **devices** and I/O styles: not just a desktop with mouse and keyboard, but a cellphone with a tiny screen, buttons, and voice I/O; or a large wall screen with touch-sensitive finger input. Another kind of adaptation envisioned by model-based UI is **user adaptation**; an interface might change depending on how the user interacts with it.

UIML Approach

- Programmer writes XML spec for both model and view
  - Model: <description>
  - Grouping: <structure>
  - Labels: <data>
  - Widget selection & layout: <style>
  - Behavior: <events>
- Separation of concerns allows managing families of interfaces
  - Reuse application parts for multiple devices
  - Reuse device parts for multiple applications

The readings for today's lecture covered two approaches to model-based UI. With UIML, the programmer writes an XML specification that includes not only the model but also explicit rules for transforming the model into a presentation. This puts the programmer in control, but still enables the same UI model to be retargeted to different devices or platforms.

SUPPLE Approach

- Application model
  - Elements: state variables and commands
  - Tree structure: grouping
  - Labels for each element
- Device description
  - Widget set
  - Navigation costs (switch, enter, leave)
  - Manipulation costs (changing value)
- User data
  - Trace of actions by a user
- System automatically searches for a presentation
  - Assignment of widgets to model elements that minimizes cost of user trace

SUPPLE is interesting because it's much more automatic. Given a model and an underspecified presentation, SUPPLE can search for a presentation that optimizes a certain cost function. As described in the paper, SUPPLE optimizes for **efficiency**, given some information about the kinds of actions that a particular user does. The cost of actions includes both navigation – moving from one widget to another – and manipulation – changing the value of a control. If two widgets are on different tab panes, for example, the cost of navigating between them is higher than if they were colocated. The cost of manipulating a spinner widget (with only up/down arrows, hence O(n) time to change a distance of n) is higher than that of a slider widget (which uses Fitts's Law, hence O(log n) time to change).

The natural question to ask about model-based user interfaces is, how much control do you really have over the usability of the final presentation? For example, SUPPLE optimizes for efficiency – what if other dimensions of usability are more important, like learnability or minimizing errors? Also, what about internal consistency – if an interface appears different on different devices, or if it changes over time to optimize your actions, then the user may have to relearn the interface as it changes. Model-based user interfaces are still an active area of research.