Damage Management & Layout
Damage management

- Need to keep track of parts of the screen that need update
  - interactor has changed appearance, moved, appeared, disappeared, etc.
  - done by “declaring damage”
    - each object responsible for telling system when part of its appearance needs update
Damage management

- Example: in Swing done via a call to repaint()
  - takes a rectangle parameter
  - Adds the specified region to the RepaintManager’s dirty list
    - list of regions that need to be redrawn
    - RepaintManager schedules repaints for later, can collapse multiple dirty regions into a few larger ones to optimize
  - When scheduled repaint comes up, RepaintManager calls component’s paintImmediately() method, which calls paintComponent(), paintChildren(), paintBorders()
    - You generally never want to call this yourself
    - Generally, seldom need to work with RepaintManager directly
Damage Management

- Can optimize somewhat
  - Multiple rectangles of damage
  - Knowing about opaque objects
- But typically not worth the effort
Damage Management in Swing

JComponent
- repaint()
- paintImmediately()
- paintComponent()
- paintBorder()
- paintChildren()

RepaintManager
- addDirtyRegion()

Event Dispatch Queue
Typical overall “processing cycle”

loop forever

wait for event then dispatch it

causes actions to be invoked and/or update interactor state

typically causes damage

if (damaged_somewhere)

layout
Layout

- Deciding size and placement of every object
  - easiest version: static layout
    - objects don’t move or change size
    - easy but very limiting
      - hard to do dynamic content
  - only good enough for simplest cases
Dynamic layout

- Change layout on the fly to reflect the current situation
- Need to do layout before redraw
  - Can’t be done e.g., in paintComponent()
  - Why?
Dynamic layout

- Change layout on the fly to reflect the current situation
- Need to do layout before redraw
  - Can’t be done e.g., in paintComponent()
  - Because you have to draw in strict order, but layout (esp. position) may depend on size/position of things not in order (drawn after you!)
Layout in Swing

- invalidate() method
  - Called on a container to indicate that its children need to be laid out
  - Called on a component to indicate that something about it has changed that may change the overall layout (change in size, for example)

- validate() method
  - Starts the process that makes an invalid layout valid--recomputes sizes and positions to get correct layout
“Issues” with Swing validation

- `invalidate()` is often called automatically
  - e.g., in response to changes to components’ state
- ... but not always
  - e.g., if a JButton’s font or label changes, no automatic call to `invalidate()`
  - Mark the button as changed by calling `invalidate()` on it
  - Tell the container to redo layout by calling `validate()` on it
- In older versions of Swing you had to do this by hand
- Newer versions (post 1.2) add a shortcut: `revalidate()`
  - Invalidates the component you call it on
  - Begins the process of validating the layout, starting from the appropriate parent container
- Validation *also* uses the RepaintManager
Layout Validation in Swing

JComponent

revalidate()

validate()

Container

RepaintManager

addInvalidComponent()

Event Dispatch Queue
Layout with containers

- Containers (parent components) can control size/position of children
  - example: rows & columns
  - Two basic strategies
    - Top-down (AKA outside-in)
    - Bottom-up (AKA inside-out)
Top-down or outside-in layout

- Parent determines layout of children
  - Typically used for position, but sometimes size
  - Example?
Top-down or outside-in layout

- Parent determines layout of children
  - **Typically used for position**, but sometimes size
  - Dialog box OK / Cancel buttons
    - stays at lower left
Bottom-up or inside-out layout

- Children determine layout of parent
  - Typically just size
  - Example?
Bottom-up or inside-out layout

- Children determine layout of parent
  - Typically just size
  - Shrink-wrap container
    - parent just big enough to hold all children
    - e.g., pack() method on JWindow and JFrame
      - Resizes container to just big enough to accommodate contents’ preferredSizes
Which one is better?
Neither one is sufficient

- Need both
- May even need both in same object
  - horizontal vs. vertical
  - size vs. position (these interact!)
- Need more general strategies
Layout Policies in Swing

- Swing layout policies are (generally) customizable
- Some containers come with a “built-in” layout policy
  - JSplitPane, JScrollPane, JTabbedPane
- Others support “pluggable” policies through `LayoutManager`
  - LayoutManagers installed in Containers via `setLayout()`
  - Two interfaces (from AWT): `LayoutManager` and `LayoutManager2`
  - Determines position and size of each component within a container
  - Looks at components inside container:
    - Uses `getMinimumSize()`, `getPreferredSize()`, `getMaximumSize()`
    - ... but is free to ignore these
- Example LayoutManagers:
  - FlowLayout, BorderLayout, GridLayout, BoxLayout, ...
Layout Policies in Swing

- Each LayoutManager is free to do what it wants when layout out components
  - Can ignore components’ min/preferred/max sizes
  - Can ignore (not display) components at all
- Generally, most will look at children’s requests and then:
  - Size the parent component appropriately
  - Position the children within that component
- So, top-down with input from child components
More general layout strategies

- Boxes and glue model
- Springs and struts model
- Constraints
Boxes and glue layout model

- Comes from the TeX document processing system
  - Brought to UI work in Interviews toolkit (C++ under X-windows)
  - See “Composing User Interfaces with Interviews”
- Tiled composition (no overlap)
  - toolkit has other mechanisms for handling overlap
  - glue between components (boxes)
Boxes and glue layout model

- 2 kinds of boxes: hbox & vbox
  - do horiz and vert layout separately
    - at separate levels of hierarchy
- Each component has
  - natural size
  - min size
  - max size
Box sizes

- Natural size
  - the size the object would normally like to be
    - e.g., button: title string + border
- Min size
  - minimum size that makes sense
    - e.g. button may be same as natural
- Max size ...
Boxes and glue layout model

- Each piece of glue has:
  - natural size
  - min size (always 0)
  - max size (often “infinite”)
  - stretchability factor (0 or “infinite” ok)
- Stretchability factor controls how much this glue stretches compared with other glue
Example (Paper: p13, fig 4&5)

- Two level composition
  - vbox
    - middle glue twice as stretchable as top and bottom
  - hbox at top
    - right glue is infinitely stretchable
  - hbox at bottom
    - left is infinitely stretchable
How boxes and glue works

- Boxes (components) try to stay at natural size
  - expand or shrink glue first
  - if we can’t fit just changing glue, only then expand or shrink boxes
- Glue stretches / shrinks in proportion to stretchability factor
Computing boxes and glue layout

- Two passes:
  - bottom up then top down
- Bottom up pass:
  - compute natural, min, and max sizes of parent from natural, min, and max of children
  - natural = sum of children’s natural
  - min = sum of children’s min
  - max = sum of children’s max
Computing boxes and glue layout

- Top down pass:
  - window size fixed at top
  - at each level in tree determine space overrun (shortfall)
  - make up this overrun (shortfall) by shrinking (stretching)
    - glue shrunk (stretched) first
    - if reaches min (max) only then shrink (stretch components)
Top down pass (cont)

- Glue is changed proportionally to stretchability factor
  - example: 30 units to stretch
    - glue_1 has factor 100
    - glue_2 has factor 200
  - stretch glue_1 by 10
  - stretch glue_2 by 20
- Boxes changed evenly (within min, max)
What if it doesn’t fit?

- Layout breaks
  - negative glue
  - leads to overlap
Springs and struts model

- Developed independently, but can be seen a simplification of boxes and glue model
  - more intuitive (has physical model)
- Has struts, springs, and boxes
  - struts are 0 stretchable glue
  - springs are infinitely stretchable glue
Springs and struts model

- **Struts**
  - specify a fixed offset
- **Springs**
  - specify area that is to take up slack
  - equal stretchability
- **Components (boxes)**
  - not stretchable (min = natural = max)
Constraints

- A more general approach
- General mechanism for establishing and maintaining relationships between things
  - layout is one use
  - several other uses in UI
    - deriving appearance from data
    - multiple view of same data
    - automated semantic feedback
General form: declare relationships

- Declare “what” should hold
  - this should be centered in that
  - this should be 12 pixels to the right of that
  - parent should be 5 pixels larger than its children

- System automatically maintains relationships under change
  - system provides the “how”
You say what
System figures out how

- A very good deal
- But sounds too good to be true
You say what
System figures out how

- A very good deal
- But sounds too good to be true
  - It is: can’t do this for arbitrary things (unsolvable problem)
- Good news: this can be done if you limit form of constraints
  - limits are reasonable
  - can be done very efficiently
Form of constraints

- For UI work, typically express in form of equations
  - $\text{this.x} = \text{that.x} + \text{that.w} + 5$
    - 5 pixels to the right
  - $\text{this.x} = \text{that.x} + \text{that.w}/2 - \text{this.w}/2$
    - centered
  - $\text{this.w} = 10 + \max \text{child[i].x} + \text{child[i].w}$
    - 10 larger than children
The Power of Constraints

- this.x = that.x + that.w/2 - this.w/2
  - What’s so cool about this?
- Power comes from *dynamic computation of result*
  - Value isn’t just computed immediately
  - Instead, saves references to objects involved in calculation
  - When any operand changes, result value is automatically recomputed
- Express relationships declaratively
- Systems updates as necessary to preserve the constraints you’ve specified
How would you express this?

- $this.x = that.x + that.w/2 - this.w/2$
- Remember, not programming language expression!

- Parsable strings
  - $c = \text{new Constraint(“this.x = that.x + that.w/2 - this.w/2”})$

- Nested function calls
  - $c = \text{new Constraint(Equals(this.x, Add(this.x, Sub(Div(that.w, 2), Div(this.w, 2)))))}$

- Operator overloading
  - If your language supports, it can make it look very like the example above
  - Requires defining constraint objects, overloading common arithmetic operators
Example: doing springs and struts with constraints

Parent

St1 → Obj1 → Sp1 → Obj2 → Sp2 → Obj3 → St2
Example: doing springs and struts with constraints

First, what does this do?

- Obj1 and obj3 stay fixed distance from left and right edges
- Obj2 centered between them
Example: doing springs and struts with constraints

- Compute how much space is left
  
  \[ \text{parent.slack} = \text{parent.w} - (\text{obj1.w} + \text{obj2.w} + \text{obj3.w} + \text{st1.w} + \text{st2.w}) \]
Example: doing springs and struts with constraints

- Space for each spring
  
  `parent.sp_len = parent.slack / 2`
Example: doing springs and struts with constraints

- A little better version
  
  ```python
  parent.num_sp = 2
  if parent.num_sp == 0
    parent.sp_len = 0
  else
    parent.sp_len = parent.slack / parent.num_sp
  ```
Example: doing springs and struts with constraints

- Now assign spring sizes
  
  \[
  \text{sp1.w = parent.sp_len} \\
  \text{sp2.w = parent.sp_len}
  \]
Example: doing springs and struts with constraints

- Now do positions left to right
  \[ \text{st1.x} = 0 \]
  \[ \text{obj1.x} = \text{st1.x} + \text{st1.w} \]
  \[ \text{sp1.x} = \text{obj1.x} + \text{obj1.w} \]
  ...

![Diagram of objects and constraints](image.png)
Power of constraints

- If size of some component changes, system can determine new sizes for springs, etc.
  - automatically
  - just change the size that has to change, the rest “just happens”
  - very nice property
Bigger example

- Suppose we didn’t want to fix number of children, etc. in advance
  - don’t want to write new constraints for every layout
  - instead put constraints in object classes (has to be a more general)
  - in terms of siblings & first/last child
Bigger (generalized) example

- First compute slack across arbitrary children
- Each strut, spring, and object:
  - “before” means before considering this object
  - “after” means after considering this object
  - prev_sibling is a name that dynamically refers to the object before obj at the same level in the tree
    
    if prev_sibling = null
    
    obj.sl_before = parent.w

    else

    obj.sl_before = prev_sibling.sl_after
Bigger (generalized) example

- For struts and objects:
  - Roll forward, subtracting out object sizes from slack
    \[ \text{obj.sl\_after} = \text{obj.sl\_before} - \text{obj.w} \]

- For springs:
  - Because they take up no space unless necessary, springs don’t detract from the slack
    \[ \text{spr.sl\_after} = \text{spr.sl\_before} \]
Example of a “chained” computation

- Compute my value based on previous value
  - Special case at beginning
  - This now works for any number of children
    - adding a new child dynamically not a problem
- Very common pattern
Now compute number of springs

- For springs use:
  if prev_sibling == null
    spr.num_sp = 1
  else
    spr.num_sp = prev_sibling.num_sp + 1

- For struts and objects use:
  if prev_sibling == null
    obj.num_sp = 0
  else
    obj.num_sp = prev_sibling.num_sp
Carry values to parent

- Propagate values computed in children up to the parent
- `last_child` is a dynamic reference that refers to the last child in the parent.

```python
parent.num_sp = last_child.num_sp
parent.slack = last_child.sl_after
```

- Again, don’t need to know how many children
  - Correct value always at last one
Compute spring lengths

- Figure up the length we’ll use for each spring:
  
  ```python
  if parent.num_sp == 0
      parent.sp_len = 0
  else
      parent.sp_len = parent.slack / parent.num_sp
  ```
Set sizes of springs & do positions

- For springs use:
  \[ \text{spr.w} = \text{parent.sp_len} \]

- For all use:
  ```java
  if prev_sibling == null
    obj.x = 0
  else
    obj.x = prev_sibling.x + prev_sibling.w
  ```
More complex, but...

• Only have to write it once
  • put it in various superclasses
  • this is basically all we have to do for springs and struts layout (if we have constraints)
  • can also do boxes and glue (slightly more complex, but not unreasonable)
  • can write other kinds of layout and mix and match using constraints
Springs ‘n’ Struts in Swing

- Swing provides a basic constraint-based Springs’n’struts LayoutManager
  - javax.swing.SpringLayout
- Allows simple arithmetic computation of constraints
Dependency graphs

- Useful to look at a system of constraints as a “dependency graph”
  - graph showing what depends on what
  - two kinds of nodes (bipartite graph)
    - variables (values to be constrained)
    - constraints (equations that relate)
Dependency graphs

- Example: $A = f(B, C, D)$

- Edges are dependencies
Dependency graphs

- Dependency graphs chain together: $X = g(A, Y)$
Kinds of constraint systems

- Actually lots of kinds, but 2 major varieties used in UI work
  - reflect kinds of limitations imposed
- One-Way constraints
  - must have a single variable on LHS
  - information only flows to that variable
    - can change B,C,D system will find A
    - can’t do reverse (change A …)
One-Way constraints

- Results in a directed dependency graph:
  - \( A = f(B,C,D) \)

- Normally require dependency graph to be acyclic
  - cyclic graph means cyclic definition
One-Way constraints

- Problem with one-way: introduces an asymmetry
  \[ \text{this}.x = \text{that}.x + \text{that}.w + 5 \]
  - can move (change x) “that”, but not “this”
Multi-way constraints

- Don’t require info flow only to the left in equation
  - can change A and have system find B, C, D
- Not as hard as it might seem
  - most systems require you to explicitly factor the equations for them
    - provide $B = g(A, C, D)$, etc.
Multi-way constraints

- Modeled as an undirected dependency graph
- No longer have asymmetry
Multi-way constraints

- But all is not rosy
  - most efficient algorithms require that dependency graph be a tree (acyclic undirected graph)

\[
\begin{align*}
X & \rightarrow g & A & \rightarrow f \\
Y & \downarrow & & \\
B & & C & \rightarrow D
\end{align*}
\]
Multi-way constraints

- But: $A = f(B, C, D) \& X = h(D, A)$

**Not OK** because it has a cycle (not a tree)
Another important issue

- A set of constraints can be:
  - Over-constrained
    - No valid solution that meets all constraints
  - Under-constrained
    - More than one solution
      - sometimes infinite numbers
Over- and under-constrained

- Over-constrained systems
  - solver will fail
  - isn’t nice to do this in interactive systems
  - typically need to avoid this
    - need at least a “fallback” solution
Over- and under-constrained

- Under-constrained
  - many solutions
  - system has to pick one
  - may not be the one you expect
  - example: constraint: point stays at midpoint of line segment
    - move end point, then?
Over- and under-constrained

- Under-constrained
  - example: constraint: point stays at midpoint of line segment
    - move end point, then?
  - Lots of valid solutions
    - move other end point
    - collapse to one point
    - etc.
Over- and under-constrained

- Good news is that one-way is never over- or under-constrained (assuming acyclic)
  - system makes no arbitrary choices
  - pretty easy to understand
Over- and under-constrained

- Multi-way can be either over- or under-constrained
  - have to pay for extra power somewhere
  - typical approach is to over-constrain, but have a mechanism for breaking / loosening constraints in priority order
    - one way: “constraint hierarchies”
Over- and under-constrained

- Multi-way can be either over- or under-constrained
  - unfortunately system still has to make arbitrary choices
  - generally harder to understand and control
Implementing constraints

- Simple algorithm for one-way
  - Need bookkeeping for variables
  - For each keep:
    - value - the value of the var
    - eqn - code to eval constraint
    - dep - list of vars we depend on
    - done - boolean “mark” for alg
Simple algorithm for one-way

- After any change:

```java
// reset all the marks
for each variable V do
    V.done = false;

// make each var up-to-date
for each variable V do
    evaluate(V);
```
Simple algorithm for one-way

\[
evaluate(V): \\
\text{if} \ (\! V.\text{done}) \ \\
\quad V.\text{done} = \text{true}; \\
\quad \text{Parms} = \text{empty}; \\
\quad \text{for each DepVar in V.dep do} \\
\quad \quad \text{Parms} += \evaluate(\text{DepVar}) \\
\quad V.\text{value} = V.\text{eqn(Parms)} \\
\text{return V.value}
\]
Approach for multi-way implementation

- Use a “planner” algorithm to assign a direction to each undirected edge of dependency graph
- Now have a one-way problem
Better algorithms

• “Incremental” algorithms exist for both one-way and multi-way
  • don’t recompute every variable after every (small) change
  • (small) partial changes require (small) partial updates