Algorithms for User Interfaces

Jaakko Järvi

Texas A&M University
Computer Science and Engineering
Parasol Lab

November 10, 2009
Outline

- Story of why algorithms matter in programming
Outline

- Story of why algorithms matter in programming
- or a promise of never having to write a GUI event handler again
Motivation

Why is software like this?

Jaakko Järvi (TAMU CSE) Algorithms for User Interfaces
November 10, 2009 3 / 28
Motivation

Why is software like this?
def ChangeCurrentHeightPx(self, event):
    self.LastUpdated = "Height"
    constrained = self.Controls["Constrain"].GetValue()
    # no matter what the percent & current stay bound together
    # get current height, and compute relative height and place new rel. ht
    height = float(self.Controls["AbsolutePx"]["Height"]).GetValue()
    pct = height / self.InitialSize["Height"]
    self.Controls["Relative%"]["Height"].SetValue(str(pct * 100))

    if constrained:
        # update width & width%
        width = pct * self.InitialSize["Width"]
        self.Controls["AbsolutePx"]["Width"].SetValue(str(round(width)))

    def ChangeCurrentHeightPct(self, event):
        self.LastUpdated = "Height"
        constrained = self.Controls["Constrain"].GetValue()
        # no matter what the percent & current stay bound together
        # get current rel. ht,
        # compute absolute height and place new abs. ht
        height = float(self.Controls["Relative%"]["Height"]).GetValue()
        cur = height * self.InitialSize["Height"] / 100
        self.Controls["AbsolutePx"]["Height"].SetValue(str(round(cur)))

        if constrained:
            # update width & width%
            width = height * self.InitialSize["Width"] / 100
            self.Controls["AbsolutePx"]["Width"].SetValue(str(round(width)))

    def ChangeConstrainState(self, event):
        constrained = self.Controls["Constrain"].GetValue()
        # If the ratio is constrained, determine which dimension
        # was last updated and update the OTHER dimension.
        # For example: If Height was last updated, use Height as
        # Width's new percent, and update Width's absolute value
        if constrained:
            if self.LastUpdated == "Height": # update width px & %
                pct = float(self.Controls["Relative%"]["Height"]).GetValue()
                self.Controls["Relative%"]["Width"].SetValue(str(pct))
                width = pct * self.InitialSize["Width"] / 100
                self.Controls["AbsolutePx"]["Width"].SetValue(str(round(width)))
            else: # update height px & %
                pct = float(self.Controls["Relative%"]["Height"]).GetValue()
                self.Controls["Relative%"]["Height"].SetValue(str(pct))
                height = pct * self.InitialSize["Height"] / 100
                self.Controls["AbsolutePx"]["Height"].SetValue(str(round(height)))

Jaakko Järvi (TAMU CSE)
Reuse is a proven and successful route to improve quality of software, and increase programmer productivity

- Vasts amounts of well tested and proven code routinely reused
  - GUI components, delivering events, rendering, capturing user’s actions
  - Example: a typical TextBox widget: 100 methods, recognizes > 200 events

- Compositions are not reusable
  ⇒ ad-hoc solutions, defects, inconsistent behavior, costly development

- Incidental data structures that arise from a network of objects
- Incidental algorithms that arise from the concert of localized actions

- Minimal requirement for reuse: understandable model
  - Not satisfied by incidental data structures and algorithms

Jaakko Järvi (TAMU CSE)
Algorithms for User Interfaces
November 10, 2009 5 / 28
**Motivation**

- Reuse is a proven and successful route to improve quality of software, and increase programmer productivity
- Vasts amounts of well tested and proven code routinely reused
  - GUI components, delivering events, rendering, capturing user’s actions
  - Example: a typical TextBox widget: 100 methods, recognizes > 200 events
- Compositions are not reusable
  -⇒ ad-hoc solutions, defects, inconsistent behavior, costly development
- Incidental data structures that arise from a network of objects
- Incidental algorithms that arise from the concert of localized actions
- Minimal requirement for reuse: understandable model
  - Not satisfied by incidental data structures and algorithms

---

<table>
<thead>
<tr>
<th>Initial Height</th>
<th>Absolute Height</th>
<th>Relative Height (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1500</td>
<td>1500.0</td>
<td>100.0</td>
</tr>
</tbody>
</table>

---

**Jaakko Järvi (TAMU CSE)**

Algorithms for User Interfaces

November 10, 2009 5 / 28
Motivation

- Reuse is a proven and successful route to improve quality of software, and increase programmer productivity
- Vasts amounts of well tested and proven code routinely reused
  - GUI components, delivering events, rendering, capturing user’s actions
  - Example: a typical TextBox widget: 100 methods, recognizes > 200 events
- Compositions are not reusable
  ⇒ ad-hoc solutions, defects, inconsistent behavior, costly development
- Incidental data structures that arise from a network of objects
- Incidental algorithms that arise from the concert of localized actions
- Minimal requirement for reuse: understandable model
  - Not satisfied by incidental data structures and algorithms
Reuse is a proven and successful route to improve quality of software, and increase programmer productivity.

Vasts amounts of well tested and proven code routinely reused
- GUI components, delivering events, rendering, capturing user’s actions
- Example: a typical TextBox widget: 100 methods, recognizes > 200 events

Compositions are not reusable
⇒ ad-hoc solutions, defects, inconsistent behavior, costly development

Incidental data structures that arise from a network of objects

Incidental algorithms that arise from the concert of localized actions

Minimal requirement for reuse: understandable model
- Not satisfied by incidental data structures and algorithms
Reuse is a proven and successful route to improve quality of software, and increase programmer productivity.

Vasts amounts of well tested and proven code routinely reused
- GUI components, delivering events, rendering, capturing user’s actions
- Example: a typical TextBox widget: 100 methods, recognizes > 200 events

Compositions are not reusable
⇒ ad-hoc solutions, defects, inconsistent behavior, costly development

Incidental data structures that arise from a network of objects

Incidental algorithms that arise from the concert of localized actions

Minimal requirement for reuse: understandable model
- Not satisfied by incidental data structures and algorithms
Software is forever doomed!

Given a sorted array \( A[0] \leq A[1] \leq \ldots \leq A[n-1] \), we want to determine if a given element \( T \) is in the array. Binary search solves the problem by keeping track of a range within the array in which \( T \) must be if it is anywhere in the array. Initially the range is the entire array. The range is shrunk by comparing its middle element to \( T \), and then discarding half the range. The process continues until \( T \) is found, or until the range in which it must lie is known to be empty. In an \( n \)-element table, the search uses roughly \( \log_2(n) \) comparisons.
Software is forever doomed!

```
int* binary_search(int* first, int* last, int x) {
    while (first != last) {
        int* middle = first + (last - first) / 2;
        if (*middle < x) first = middle + 1;
        else last = middle;
    }
    return first;
}
```
Cancel that, programming is not forever doomed after all

- The problem: UI related code is
  - bloated and buggy
    - for example, Adobe’s desktop applications, event handling is estimated to account for a third of the code and over half of the defects
  - full of incidental data structures and algorithms

- An approach for improving the status quo
  - To understand the commonalities that exist in event-handling code
  - To define a model that captures these commonalities
  - To apply
    - replace incidental data structures with explicit data structures
    - replace incidental algorithms with explicit reusable algorithm

- Result: substantial increase in reuse, programming productivity, software correctness and quality
Cancel that, programming is not forever doomed after all

- The problem: UI related code is
  - bloated and buggy
    - for example, Adobe’s desktop applications, event handling is estimated to account for a third of the code and over half of the defects
  - full of incidental data structures and algorithms

- An approach for improving the status quo
  - To understand the commonalities that exist in event-handling code
  - To define a model that captures these commonalities
  - To apply
    - replace incidental data structures with explicit data structures
    - replace incidental algorithms with explicit reusable algorithm

- Result: substantial increase in reuse, programming productivity, software correctness and quality
Outline

1. Motivation

2. Command Parameter Synthesis

3. Property Models as Multi-way Dataflow Constraint Systems

4. What was achieved

5. Experience and Conclusions
Dialogs serve to assist the user in selecting values for parameters to some command.
Dialogs serve to assist the user in selecting values for parameters to some command.

- Command interested in only a few values
Dialogs serve to assist the user in selecting values for parameters to some command.

Command interested in only a few values

- Dialog may provide more values than necessary for assistance.
Dialogs serve to assist the user in selecting values for parameters to some command.

- Command interested in only a few values
  - Dialog may provide more values than necessary for assistance
- After the user edits a value,
  - The dialog is inconsistent
Dialogs serve to assist the user in selecting values for parameters to some command.

Command interested in only a few values
- Dialog may provide more values than necessary for assistance
- After the user edits a value,
  - The dialog is inconsistent
- Then it tries to restore consistency
Dialogs serve to assist the user in selecting values for parameters to some command. Command interested in only a few values:
- Dialog may provide more values than necessary for assistance
- After the user edits a value,
  - The dialog is inconsistent
- Then it tries to restore consistency
Dialogs serve to assist the user in selecting values for parameters to some command.

- Command interested in only a few values
  - Dialog may provide more values than necessary for assistance
- After the user edits a value,
  - The dialog is inconsistent
- Then it tries to restore consistency
Understanding UIs: Command Parameter Synthesis

- Dialogs serve to assist the user in selecting values for parameters to some command.

- Command interested in only a few values
  - Dialog may provide more values than necessary for assistance

- After the user edits a value,
  - The dialog is inconsistent

- Then it tries to restore consistency
Outline

1. Motivation
2. Command Parameter Synthesis
3. Property Models as Multi-way Dataflow Constraint Systems
4. What was achieved
5. Experience and Conclusions
Core of the Model: Multi-way Dataflow Constraint System

- Variables...
- Tied together by constraints...

- Height
- Absolute
- Initial \cdot (Height Relative 100)

Each of which can be satisfied by any of a number of methods:

- a: \[ \text{absolute height} = \text{initial height} \cdot \text{relative height} / 100; \]
- b: \[ \text{relative height} = (\text{absolute height} / \text{initial height}) \cdot 100; \]
Core of the Model: Multi-way Dataflow Constraint System

Variables ...

\[
\text{Absolute Height} = \text{Initial Height} \cdot \frac{\text{Relative Height}}{100};
\]

\[
\text{Relative Height} = \frac{\text{Absolute Height}}{\text{Initial Height}} \cdot 100;
\]
Core of the Model: Multi-way Dataflow Constraint System

Variables ...

... tied together by constraints ...

Absolute Height

Absolute Width

Initial Height

Initial Width

Relative Height (%)

Relative Width (%)
Variables ...

tied together by constraints ...

\[ \text{Height}_{\text{Absolute}} = \text{Height}_{\text{Initial}} \cdot \left( \frac{\text{Height}_{\text{Relative}}}{100} \right) \]
Variables ...

tied together by constraints ...

- \( \text{Height}_{\text{Absolute}} = \text{Height}_{\text{Initial}} \cdot \left( \frac{\text{Height}_{\text{Relative}}}{100} \right) \)

- each of which can be satisfied by any of a number of methods
Variables ...

tied together by constraints ...

- \( \text{Height}_{\text{Absolute}} = \text{Height}_{\text{Initial}} \cdot \left( \frac{\text{Height}_{\text{Relative}}}{100} \right) \)

each of which can be satisfied by any of a number of methods

- \( a: \text{absolute\_height} = \text{initial\_height} \times \text{relative\_height} / 100; \)
- \( b: \text{relative\_height} = (\text{absolute\_height} / \text{initial\_height}) \times 100; \)
Restoring consistency is now just solving the system.
Restoring consistency is now just solving the system

Solution defines a **dataflow**
Restoring consistency is now just solving the system.

Solution defines a dataflow:
- Selection of methods (in order) such that:
  - all constraints enforced
  - no two methods output to same variable
Restoring consistency is now just solving the system

Solution defines a dataflow
  - Selection of methods (in order) such that
    - all constraints enforced
    - no two methods output to same variable
Restoring consistency is now just solving the system.

Solution defines a dataflow:
- Selection of methods (in order) such that:
  - all constraints enforced
  - no two methods output to same variable
Restoring consistency is now just solving the system.

Solution defines a dataflow:
- Selection of methods (in order) such that:
  - all constraints enforced
  - no two methods output to same variable
Restoring consistency is now just solving the system

Solution defines a dataflow

- Selection of methods (in order) such that
  - all constraints enforced
  - no two methods output to same variable

  e.g. $a, e \rightarrow c$
Picking the “right” solution

- Programmer only defines relations and their methods, not which method to execute and when $\Rightarrow$ often multiple solutions
Picking the “right” solution

- Programmer only defines relations and their methods, not which method to execute and when ⇒ often multiple solutions
Picking the “right” solution

Programmer only defines relations and their methods, not which method to execute and when ⇒ often multiple solutions

- Need a way to order them
Picking the “right” solution

- Programmer only defines relations and their methods, not which method to execute and when ⇒ often multiple solutions
  - Need a way to order them
- In general, want to prefer methods that change older values
Picking the “right” solution

- Programmer only defines relations and their methods, not which method to execute and when ⇒ often multiple solutions
  - Need a way to order them
- In general, want to prefer methods that change older values
- Priorities

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial Height</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initial Width</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Relative Height</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Absolute Height</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Relative Width</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Absolute Width</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Picking the “right” solution

Programmer only defines relations and their methods, not which method to execute and when ⇒ often multiple solutions

- Need a way to order them

- In general, want to prefer methods that change older values

- Priorities = Hierarchical Stay Constraints
Picking the “right” solution

- Programmer only defines relations and their methods, not which method to execute and when \( \Rightarrow \) often multiple solutions
  - Need a way to order them
- In general, want to prefer methods that change older values
- **Priorities** = Hierarchical Stay Constraints
  - Stay constraint = does nothing, so its variable *stays* the same
Picking the “right” solution

- Programmer only defines relations and their methods, not which method to execute and when => often multiple solutions
  - Need a way to order them
- In general, want to prefer methods that change older values
- Priorities = Hierarchical Stay Constraints
  - Stay constraint = does nothing, so its variable stays the same
  - Hierarchy = groups of constraints with certain strength
Explicit Algorithm for Command Parameter Synthesis

- Each UI element has a variable in a constraint system
- Event handling code becomes auto-generated boilerplate
  - Value modification generates a request to the constraint system to modify one variable and its priority, and solve
  - At all times, the UI element shows the value of the variable in the constraint system
Outline

1. Motivation
2. Command Parameter Synthesis
3. Property Models as Multi-way Dataflow Constraint Systems
4. What was achieved
5. Experience and Conclusions
Incidental Data Structure $\rightarrow$ Explicit Model
Incidental Data Structure $\rightarrow$ Explicit Model
Incidental Data Structure → Explicit Model
def ChangeCurrentHeightPx(self, event):
    self.LastUpdated = "Height"
    constrained = self.Controls["Constrain"].GetValue()
    # no matter what the percent & current stay bound together
    # get current height, and compute relative height and place new rel. ht
    height = float(self.Controls["AbsolutePx"]["Height"].GetValue())
    pct = height / self.InitialSize["Height"]
    self.Controls["Relative%"]["Height"].SetValue(str(pct * 100))
    if constrained: # update width & width%
        width = float(self.Controls["AbsolutePx"]["Width"].GetValue())
        self.Controls["Relative%"]["Width"]["Height"][str(round(width))]
    self.Controls["AbsolutePx"]["Height"][str(round(height))]
    def ChangeCurrentWidthPx(self, event):
        self.LastUpdated = "Width"
        constrained = self.Controls["Constrain"].GetValue()
        # no matter what the percent & current stay bound together
        # get current width, and compute relative width and place new rel. wd
        width = float(self.Controls["AbsolutePx"]["Width"]["Height"][str(round(width))]
        pct = width / self.InitialSize["Width"]
        self.Controls["Relative%"]["Width"]["Height"][str(round(pct * 100))]
        if constrained: # update height & height%
            height = float(self.Controls["AbsolutePx"]["Height"][str(round(pct))]
            self.Controls["Relative%"]["Height"][str(round(height))]
    def ChangeConstrainState(self, event):
        constrained = self.Controls["Constrain"]["Height"][str(round(pct))]
        # If the ratio is constrained, determine which dimension
        # was last updated and update the OTHER dimension.
        # For example: If Height was last updated, use Height as
        # Width’s new percent, and update Width’s absolute value
        if constrained:
            if self.LastUpdated == "Height": # update width px & %
                width = float(self.Controls["AbsolutePx"]["Height"][str(round(width))]
                self.Controls["Relative%"]["Height"][str(round(pct))]
            else: # update width px & %
                width = float(self.Controls["AbsolutePx"]["Width"][str(round(width))]
                self.Controls["Relative%"]["Height"][str(round(pct))]
        else: # update width px & %
            if self.LastUpdated == "Height": # update width px & %
                width = float(self.Controls["AbsolutePx"]["Height"][str(round(width))]
                self.Controls["Relative%"]["Height"][str(round(pct))]
            else: # update width px & %
                width = float(self.Controls["AbsolutePx"]["Width"][str(round(width))]
                self.Controls["Relative%"]["Height"][str(round(pct))]

Jaakko Järvi (TAMU CSE)
Code of Incidental Algorithm → Model Declaration

def ChangeCurrentHeightPx(self, event):
    self.LastUpdated = "Height"
    constrained = self.Controls["Constrain"].GetValue()
    # no matter what the percent & current stay bound together
    # get current height, and compute relative height and place new rel. ht
    height = float(self.Controls["AbsolutePx"]["Height"].GetValue())
    pct = height / self.InitialSize[self.Height]
    self.Controls["Relative%"]["Height"].SetValue(str(pct * 100))

    if constrained:
        # update width & width%
        width = height * self.InitialSize[self.Width] / 100
        self.Controls["AbsolutePx"]["Width"].SetValue(str(round(width)))

    def ChangeCurrentWidthPx(self, event):
        self.LastUpdated = "Width"
        constrained = self.Controls["Constrain"].GetValue()
        # no matter what the percent & current stay bound together
        # get current width, and compute relative width and place new abs. wd
        height = float(self.Controls["AbsolutePx"]["Width"].GetValue())
        pct = height / self.InitialSize[self.Width]
        self.Controls["Relative%"]["Width"].SetValue(str(pct * 100))

        if constrained:
            # update height & height%
            height = width * self.InitialSize[self.Height] / 100
            self.Controls["AbsolutePx"]["Height"].SetValue(str(round(height)))

        def ChangeCurrentHeightPct(self, event):
            self.LastUpdated = "Height"
            constrained = self.Controls["Constrain"].GetValue()
            # no matter what the percent & current stay bound together
            # get current rel. ht,
            # and compute absolute height and place new abs. ht
            height = float(self.Controls["Relative%"]["Height"].GetValue())
            cur = height * self.InitialSize[self.Height] / 100
            self.Controls["AbsolutePx"]["Height"].SetValue(str(round(cur)))

            if constrained:
                # update width & width%
                width = height * self.InitialSize[self.Width] / 100
                self.Controls["AbsolutePx"]["Width"].SetValue(str(round(width)))

        def ChangeCurrentWidthPct(self, event):
            self.LastUpdated = "Width"
            constrained = self.Controls["Constrain"].GetValue()
            # no matter what the percent & current stay bound together
            # get current rel. wd,
            # and compute absolute width and place new abs. wd
            width = float(self.Controls["Relative%"]["Width"].GetValue())
            cur = width * self.InitialSize[self.Width] / 100
            self.Controls["AbsolutePx"]["Width"].SetValue(str(round(cur)))

            if constrained:
                # update height & height%
                height = width * self.InitialSize[self.Height] / 100
                self.Controls["AbsolutePx"]["Height"].SetValue(str(round(height)))

        def ChangeConstrainState(self, event):
            self.LastUpdated = "Height"
            constrained = self.Controls["Constrain"].GetValue()
            # If the ratio is constrained, determine which dimension
            # was last updated and update the OTHER dimension.
            # For example:
            # if Height was last updated, use Height as
            # Width's new percent, and update Width's absolute value
            if constrained:
                if self.LastUpdated == "Height": # update width px & %
                    pct = float(self.Controls["Relative%"]["Height"].GetValue())
                    self.Controls["AbsolutePx"]["Width"].SetValue(str(pct * 100))
                    width = pct * self.InitialSize[self.Width] / 100
                    self.Controls["AbsolutePx"]["Height"].SetValue(str(round(width)))
                else: # update width px & %
                    pct = float(self.Controls["Relative%"]["Width"].GetValue())
                    self.Controls["AbsolutePx"]["Height"].SetValue(str(pct))
                    height = pct * self.InitialSize[self.Height] / 100
                    self.Controls["AbsolutePx"]["Height"].SetValue(str(round(height)))
Declarative Specification of Command Parameter Synthesis

sheet image_resize {
    input:
        initial_width : 5 * 300;
        initial_height : 7 * 300;
    interface:
        preserve_ratio : true;
        absolute_width : initial_width;
        absolute_height : initial_height;
        relative_width; relative_height;
    logic:
        relate {
            absolute_height <= relative_height * initial_height / 100;
            relative_height <= absolute_height * 100 / initial_height;
        }
        relate {
            absolute_width <= relative_width * initial_width / 100;
            relative_width <= absolute_width * 100 / initial_width;
        }
        when (preserve_ratio) relate {
            relative_width <= relative_height;
            relative_height <= relative_width;
        }
}
Declarative Specification of Command Parameter Synthesis

sheet image_resize {
  input:
  initial_width : 5 * 300;
  initial_height : 7 * 300;
  interface:
  preserve_ratio : true;
  absolute_width : initial_width;
  absolute_height : initial_height;
  relative_width; relative_height;
  logic:
  relate {
    absolute_height <= relative_height * initial_height / 100;
    relative_height <= absolute_height * 100 / initial_height;
  }
  relate {
    absolute_width <= relative_width * initial_width / 100;
    relative_width <= absolute_width * 100 / initial_width;
  }
  when (preserve_ratio) relate {
    relative_width <= relative_height;
    relative_height <= relative_width;
  }
}
Declarative Specification of Command Parameter Synthesis

```javascript
sheet image_resize {
  input:
  initial_width : 5 * 300;
  initial_height : 7 * 300;
  interface:
  preserve_ratio : true;
  absolute_width : initial_width;
  absolute_height : initial_height;
  relative_width; relative_height;
  logic:
  relate {
    absolute_height <= relative_height * initial_height / 100; // a
    relative_height <= absolute_height * 100 / initial_height; // b
  }
  relate {
    absolute_width <= relative_width * initial_width / 100; // c
    relative_width <= absolute_width * 100 / initial_width; // d
  }
  when (preserve_ratio) relate {
    relative_width <= relative_height; // e
    relative_height <= relative_width; // f
  }
}
```
Algorithms for User Interfaces

- Before, every new feature required more spaghetti (incidental) code, specific to each dialog
- Now, each new feature can be defined as a reusable algorithm in a library
Scripting

- A script is a recorded sequence of commands
  - e.g. remove red-eye, skin blemishes, extra weight
- What do we record from our model as part of the script?
- Remember that probably not every value is useful
  - Some are provided by the document
  - Some are provided by the user
- Only want to capture what the user intended
Scripting

- A script is a recorded sequence of commands
  - e.g. remove red-eye, skin blemishes, extra weight
- What do we record from our model as part of the script?
  - Remember that probably not every value is useful
    - Some are provided by the document
    - Some are provided by the user
- Only want to capture what the user intended
Scripting

- A script is a recorded sequence of commands
  - e.g. remove red-eye, skin blemishes, extra weight

- What do we record from our model as part of the script?
- Remember that probably not every value is useful
  - Some are provided by the document
  - Some are provided by the user

- Only want to capture what the user intended
Capturing the User’s Intent

- Command looks at Absolute Height, Absolute Width,
- but what we wanted to change is Relative Height
Capturing the User’s Intent

- Command looks at Absolute Height, Absolute Width,
- but what we wanted to change is Relative Height
Capturing the User’s Intent

- Command looks at Absolute Height, Absolute Width,
- but what we wanted to change is Relative Height
Outline

1. Motivation
2. Command Parameter Synthesis
3. Property Models as Multi-way Dataflow Constraint Systems
4. What was achieved
5. Experience and Conclusions
Experiences

- Early experience deploying our approach for command parameter synthesis at Adobe
  - Code reductions of a factor of 8 to 10
  - Fewer defects
  - Consistency among user interfaces
**Experiment**

- Rewriting user interface code for a major desktop application
- Four teams of roughly three engineers each,
- each tasked with rewriting a large number of dialogs and palettes
- Three teams (AE1–AE3) used the declarative approach, fourth team (TF) a modern vendor-supplied object-oriented UI framework
Results: Productivity

- AE1–AE3 teams combined completed roughly 75 dialogs and palettes, with 50 more underway
- TF team completed fewer than 10 altogether
Experience and Conclusions

Results: Defect Count

![Graph showing defect count over reporting weeks]

- AE1
- AE2
- AE3
- TF

Jaakko Järvi (TAMU CSE)
Algorithms for User Interfaces
November 10, 2009
Future Directions

- Opportunities for user interfaces using property models
  - Recently worked on algorithms for enabling/disabling
  - Presets and defaults will follow
  - Perfecting the model for command parameter synthesis

- Incidental structures present in many areas of software
  - Want to know how the approach generalizes
  - Currently developing ideas about applying the declarative approach/constraint systems to other kinds of document modeling