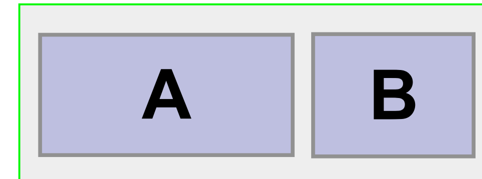
Problem



Designers working on a GUI:

- "Button A should be twice as wide as button B" $(width_A = 2\ width_B)$
- "Button B should be 100 pixels wide"
- "The GUI should be no more than 200 pixels wide"

Infeasible!

Linear Programming

Input:

- Set of linear constraints C on variables

$$C \subset \{ \; a_0x_0 + \ldots + a_mx_m + b_0y_0 + \ldots + b_ny_n \; OP \; c \;\; | \;\; a_0, \ldots, a_m, b_0, \ldots, b_n, c \in \mathbb{R} \wedge OP \in \{\leq, =, \geq\}\}$$

- Linear objective function to minimize

$$c_1x_1 + c_2x_2 + \ldots + c_nx_n$$

Output: variable values so that…

- All constraints are satisfied
- The value of the objective function is minimal

Soft Constraints

$width_A = 2\ width_B$ is a **hard** constraint

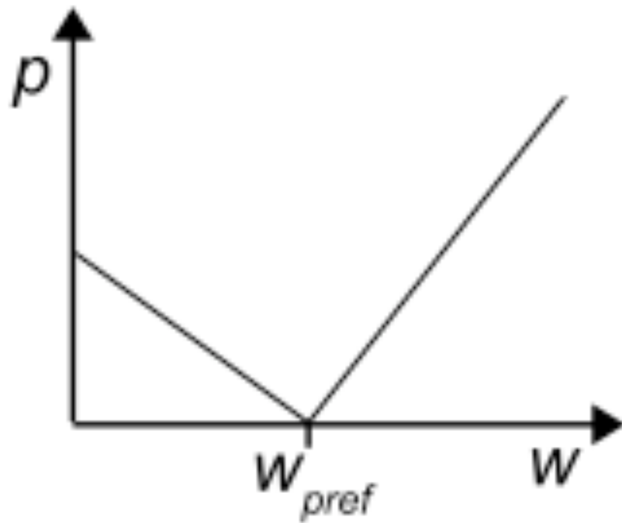
1. Allow $width_A$ to be smaller or bigger

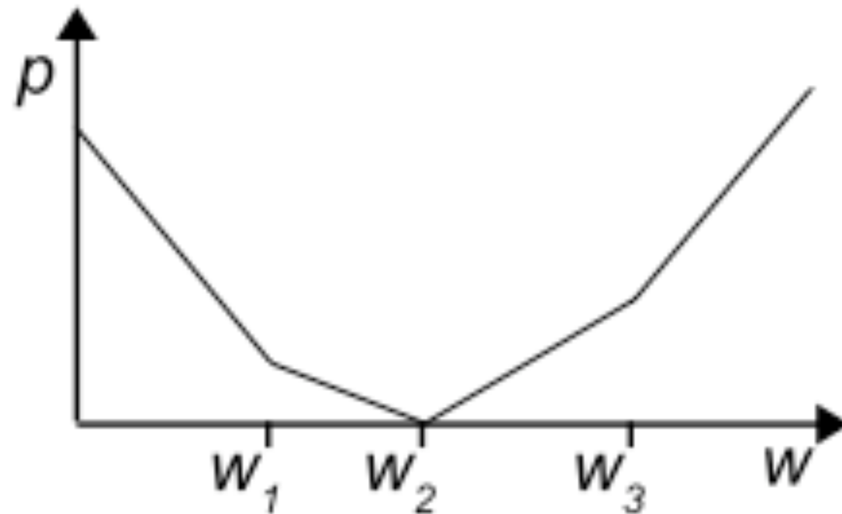
$$width_A + d_1 - d_2 = 2\ width_B \qquad d_1 \geq 0,\ d_2 \geq 0$$

2. $width_A$ should not be much smaller/bigger

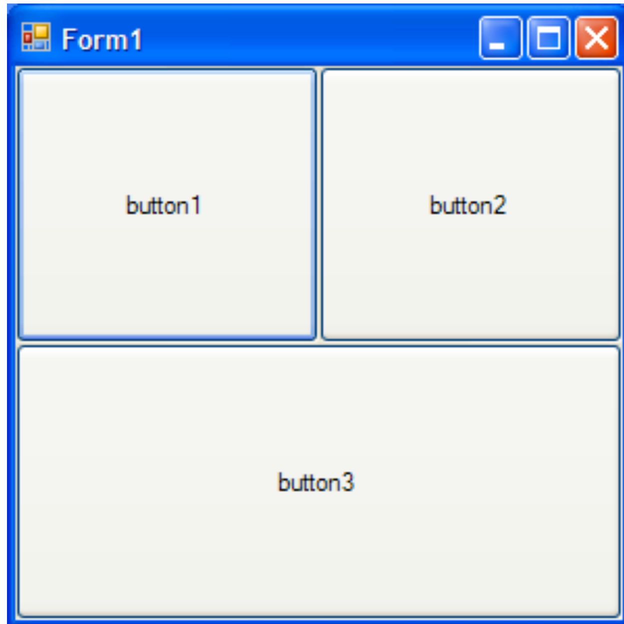
$$objective\ function = d_1 + d_2$$

Example functions of penalty over length

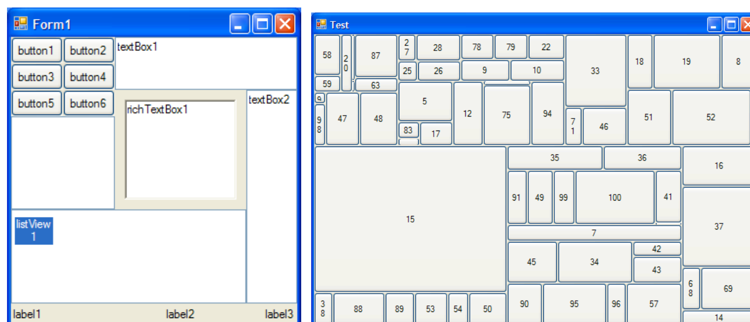




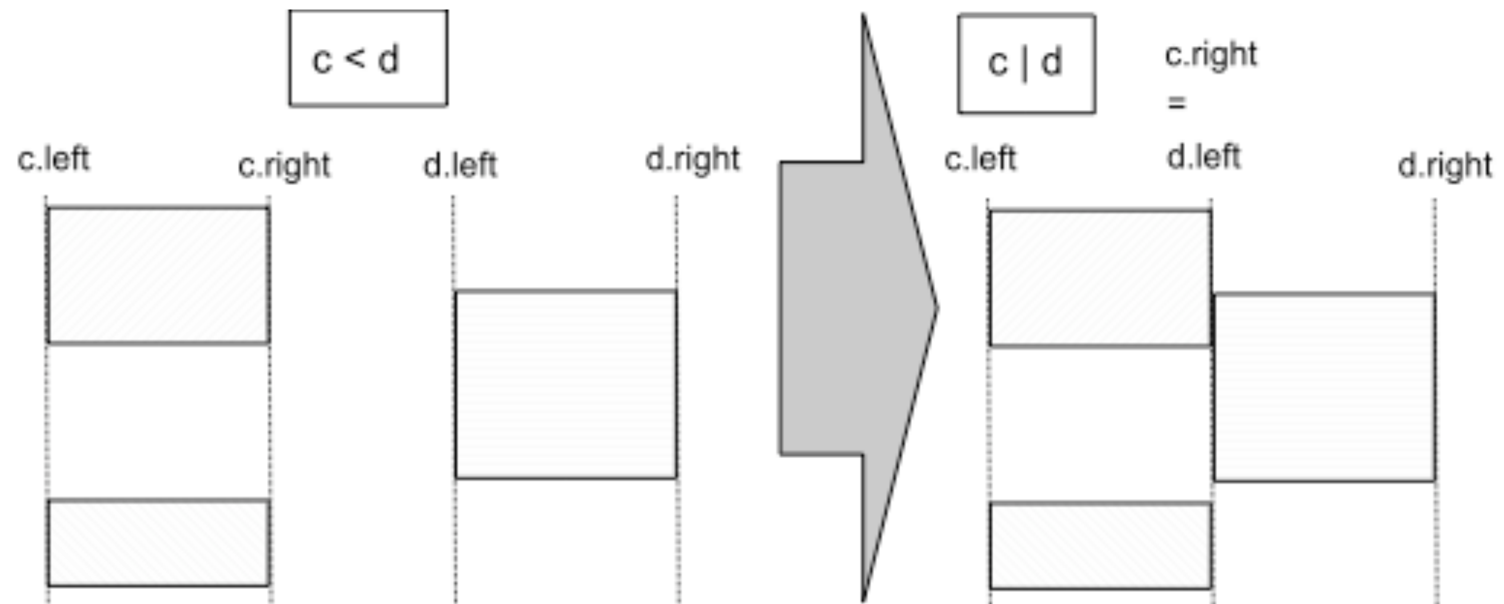
Example 1



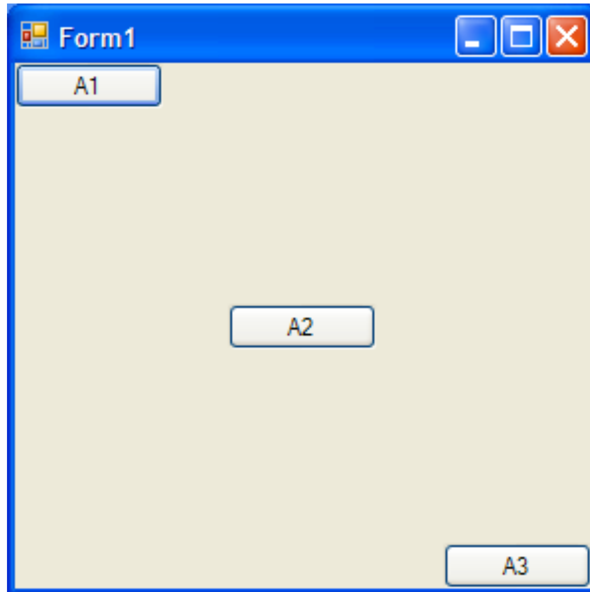

```
LayoutSpec ls = new LayoutSpec();
XTab x1 = ls.AddXTab();
YTab y1 = ls.AddYTab();

ls.AddArea(ls.Left, ls.Top, x1, y1, button1);
ls.AddArea(x1, ls.Top, ls.Right, y1, button2);
ls.AddArea(ls.Left, y1, ls.Right, ls.Bottom, button3);

ls.AddConstraint(new double[] { 2, -1 }, new
    Variable[] { x1, ls.Right }, OperatorType.EQ, 0);
ls.AddConstraint(new double[] { 2, -1 }, new
    Variable[] { y1, ls.Bottom }, OperatorType.EQ, 0);
```



Rows

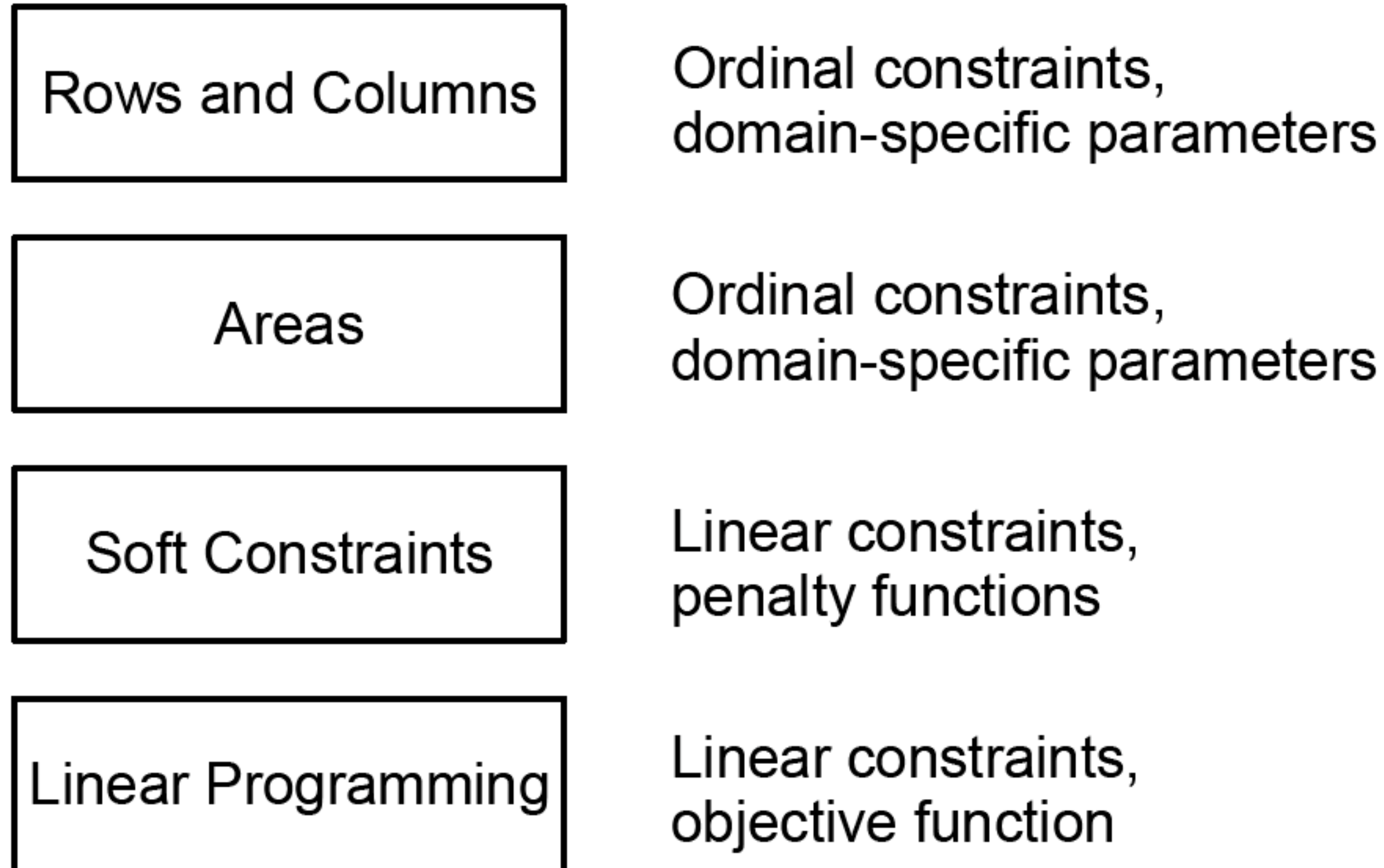


Example 2




```
LayoutSpec ls = new LayoutSpec();
Column c1 = ls.AddColumn();
Row r1 = ls.AddRow();
Row r3 = ls.AddRow();
r1.Next = r3;
Row r2 = ls.AddRow();
r2.InsertAfter(r1);
// …

Area a1 = ls.AddArea(r1, c1, b1);
a1.HAlignment = HorizontalAlignment.LEFT;
a1.VAlignment = VerticalAlignment.TOP;
Area a3 = ls.AddArea(r3, c1, b3);

r2.HasSameHeightAs(r1);
r3.HasSameHeightAs(r1);
```

ALM Levels of Abstraction



Modularity of ALM Specifications

- Compositional nature of constraints
  - Separate different concerns in a GUI into modules
  - Manage them separately and recombine them later
- Developers may omit irrelevant details
  - Solution space can be infinite (→ more flexibility)
  - E.g. partial order of GUI elements
  - Automatic inference of default values where necessary
  - E.g. parameters for resizing behavior

Module Example

```
class MyModule {
    public XTab Left, Right;
    public YTab Top, Bottom;
    …
    public
    MyModule(LayoutSpec ls, …) {
        Left = ls.AddXTab();
        ls.AddArea(…,
            new Button());
        ls.AddConstraint(…);
        …
    }
}
```

```
LayoutSpec ls =
    new LayoutSpec();
XTab x1 = ls.AddXTab();

MyModule m1 =
    new MyModule(ls, …);
ls.addEq(ls.Left, m1.Left);
ls.addEq(ls.Top, m1.Top);
ls.addEq(x1, m1.Right);
ls.addEq(ls.Bottom, m1.Bottom);

MyModule m2 =
    new MyModule(ls, …);
ls.addEq(x1, m2.Left);
…
```

Conclusion

- The Auckland Layout Model (ALM):
  tabs, areas, linear constraints, …
- Modularity through
  - Compositional nature of constraints
  - Partial specifications and default values
  - Classes as modules
- Multi-platform, open-source
- Linear programming makes it possible to transform hard constraints into soft constraints
- Project website:
  `http://www.cs.auckland.ac.nz/~lutteroth/projects/alm/`