Adobe Source Libraries
Overview & Philosophy

Sean Parent
Principal Scientist & Engineering Manager
Adobe Software Technology Lab

http://stlab.adobe.com

03 April 2008
Demo
Adobe Source Libraries

- A collection of libraries to support application development
- Research artifacts of the Adobe Software Technology Lab
- Open Source: http://stlab.adobe.com/
- Used by many Adobe products
Outline

- Regular Types – libraries for efficiently handling regular types
- Forest – advantages of explicit data structures
- Layout Library – a library for placing / aligning items in an interface and a language to express layouts
- Property Model Library – describing and solving inter-related properties
Goal of ASL

- Express entire applications using a combination of generic and declarative techniques
  - 2 orders of magnitude reduction in code
  - Greater than corresponding reduction in defects

- We are still a long way from our goal
  - perhaps not as far as it would appear
Approach

- **Generic Algorithms**
  - Write algorithms with minimal requirements – maximum reuse

- **Generic Data Structures (Containers)**
  - Containers support algorithm requirements (including complexity)

- **Declarative Architecture**
  - Identify “patterns” of how components are assembled and learn to express/solve these pattern with algorithms and data structures
Challenges

- **Build a Strong Foundation**
  - See [http://stepanovpapers.com/eop/lecture_all.pdf](http://stepanovpapers.com/eop/lecture_all.pdf)
  - Our work here has a strong impact on all aspects of ASL

- **Combine Runtime Polymorphism and Generic Programming**
  - See Poly and Any Regular Libraries

- **Make Implicit Structure Explicit**
  - Work ongoing – see Forest, Property Model, and Layout Libraries

- **Discovering the Rules that Govern Large Systems**
  - Work ongoing – see Property Model Library and initial work on Sequence Models
Adobe Source Libraries – Regular Types

- Definition: Regular
- Move Library
  - How RVO works
- Creating Polymorphic Regular Types and Poly Library
- Copy On Write Library
Definition of Regular

- The requirements of Regular are based on equational reasoning
- They assure regularity of behavior and interoperability
- Types which model these requirements are regular types

- The properties of Regular are inherent in the machine model
- Regular types exist in any correct system but formalizing the requirements and normalizing the syntax is what enables interoperability

- All types are inherently regular
### Basic Requirements of Regular Type

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Syntax Example</th>
<th>Axioms &amp; Postconditions</th>
</tr>
</thead>
</table>
| **Copy**      | T x = y; ~x();  | x == y  
if (is_defined(modify, x)  
then modify(x); x != y  |
| **Assignment**| x = y;         | x == y  
if (is_defined(modify, x)  
then modify(x); x != y  |
| **Equality**  | x == y; x != y; | a == b & & b == c => a == c  
a == b <-> b == a  
a == a  |
| **Identity**  | &x;            | &a == &b => a == b  
given &x == &y  
if (is_defined(modify, x)  
then modify(x); x == y;  |
| **Size**      | sizeof(T);     | size of local part of T  |
| **Swap**      | swap(x, y);    | x’ == y; y’ == x;  
O(sizeof(T)); nothrow; |
# Extended Requirements of Regular Type

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Syntax Example</th>
<th>Axioms &amp; Postconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Default Construction</strong></td>
<td>T x;</td>
<td>T x; x = y; is equivalent T x = y;</td>
</tr>
<tr>
<td><strong>Default Comparison</strong></td>
<td>std::less&lt;T&gt;() (x, y);</td>
<td>!op(x, y) &amp;&amp; !op(y, x) =&gt; x == y</td>
</tr>
<tr>
<td><strong>Movable</strong></td>
<td>x = f();</td>
<td>0(sizeof(T)); nothrow; T x = y; z = move(x); =&gt; z == y</td>
</tr>
<tr>
<td></td>
<td>x = move(y);</td>
<td></td>
</tr>
<tr>
<td><strong>Area</strong></td>
<td>area(x);</td>
<td>Copy and Assignment are 0(area(x)); Equality is worst case 0(area(x));</td>
</tr>
<tr>
<td><strong>Alignment</strong></td>
<td>alignment(T);</td>
<td>alignment size for type</td>
</tr>
<tr>
<td><strong>Underlying Type</strong></td>
<td>underlying(T)</td>
<td>type which can be copied to/from T in 0(size(T))</td>
</tr>
</tbody>
</table>
Importance of Move

- Allows transfer of ownership of remote parts in small constant time
- Will not throw an exception
- Move does not refine Copy and Copy does not refine Move
- When the source will not be used after a copy, copy can be replaced with move
- An object which has been moved from is still Regular

- Reference Semantics provide move for “free”
  - But there are other costs
Quiz: What will the following code print?

```cpp
struct object_t
{
    object_t()
    { cout << "construct" << endl; }
    object_t(const object_t&)  
    { cout << "copy" << endl; }
    object_t& operator=(const object_t&)  
    { cout << "assign" << endl; return *this; }
};

object_t function()
{ object_t result; return result; }

int main()
{ object_t x = function(); return 0; }
```
struct object_t
{
  object_t()
  { cout << "construct" << endl; }
  object_t(const object_t&)
  { cout << "copy" << endl; }
  object_t& operator=(const object_t&)
  { cout << "assign" << endl; return *this; }
};

object_t function()
{ object_t result; return result; }

int main()
{ object_t x = function(); return 0; }

construct
Quiz: What will the following code print?

```cpp
struct object_t
{
    object_t()
    {
        cout << "construct" << endl;
    }
    object_t(const object_t&)
    {
        cout << "copy" << endl;
    }
    object_t& operator=(const object_t&)
    {
        cout << "assign" << endl;
        return *this;
    }
};

object_t function()
{
    object_t result; return result;
}

void sink(object_t) {}}

int main()
{
    sink(function()); return 0;
}
```
struct object_t
{
    object_t()
    { cout << "construct" << endl; }
    object_t(const object_t&) 
    { cout << "copy" << endl; }
    object_t& operator=(const object_t&) 
    { cout << "assign" << endl; return *this; }
};

object_t function()
{ object_t result; return result; }

void sink(object_t) {} 

int main()
{ sink(function()); return 0; }

construct
Sink Functions

- A sink function is any function which consumes one or more arguments by storing them or by returning them.
- By passing the argument by value and moving it into position we allow the compiler to avoid a copy.
- Assignment is a sink function.
struct object_t{
    object_t() : object_m(new int(0)) { }
    object_t(const object_t& x) : object_m(new int(*x.object_m))
    { cout << "copy" << endl; }
    object_t& operator=(const object_t& x)
    { object_t tmp = x; swap(tmp, *this); return *this; }
    ~object_t() { delete object_m; }

    friend inline void swap(object_t& x, object_t& y)
    { swap(x.object_m, y.object_m); }

private:
    int* object_m;
};

object_t function()
{ object_t result; return result; }

int main()
{ object_t x; x = function(); return 0; }


copy
Better Assignment

```cpp
struct object_t{
    object_t() : object_m(new int(0)) { }
    object_t(const object_t& x) : object_m(new int(*x.object_m))
    { cout << "copy" << endl; }
    object_t& operator=(object_t x)
    { swap(x, *this); return *this; }
    ~object_t() { delete object_m; }

    friend inline void swap(object_t& x, object_t& y)
    { swap(x.object_m, y.object_m); }

private:
    int* object_m;
};

object_t function()
{ object_t result; return result; }

int main()
{ object_t x; x = function(); return 0; }
```
```cpp
struct object_t{
    object_t() : object_m(new int(0)) { }
    object_t(const object_t& x) : object_m(new int(*x.object_m))
        { cout << "copy" << endl; }
    object_t& operator=(object_t x)
        { swap(x, *this); return *this; }
    ~object_t() { delete object_m; }

    friend inline void swap(object_t& x, object_t& y)
        { swap(x.object_m, y.object_m); }

    private:
        int* object_m;
};

object_t function()
    { object_t result; return result; }

int main()
    { object_t x; x = function(); return 0; }
```
Explicit Move

```cpp
struct object_t{
    object_t(move_from<object_t> x) : object_m(0)
    {
        swap(*this, x.source);
    }

    int& get() { return *object_m; }

    //…
};

object_t function()
    { object_t result; return result; }

object_t sink(object_t x)
    { x.get() += 5; return move(x); }

int main()
    { object_t x = sink(function()); return 0; }
```
Polymorphism and Regular Types

- **Current pattern:**
  - polymorphism => inheritance => specialized classes => limited code sharing
  - polymorphism => variable size => heap allocation => pointer management
  - polymorphism => virtual functions => slower dispatch

- The requirement for polymorphism comes from the need to handle heterogeneous types which satisfy a common set of requirement in a homogeneous manner

- Requirement is driven by the use of the type, there is nothing inherently polymorphic about a type
Creating a Polymorphic Regular Type

```cpp
struct object_t
{
    template <typename T> // T models Drawable
    explicit object_t(T x) : object_m(new model_t<T>(move(x))) {} 

    object_t(move_from<object_t> x) : object_m(0)
    {
        swap(*this, x.source);
    }

    object_t(const object_t& x) : object_m(x.object_m->copy_()) {}

    object_t& operator=(object_t x) { swap(x, *this); return *this; }

    ~object_t() { delete object_m; }

    friend inline void swap(object_t& x, object_t& y)
    {
        using std::swap;
        swap(x.object_m, y.object_m);
    }

    friend inline void draw(const object_t& x)
    {
        x.object_m->draw_();
    }

private:
    // ...fill in here...
    concept_t* object_m;
};
```
Creating a Polymorphic Regular Type

```cpp
struct concept_t
{
    virtual ~concept_t() {}
    virtual concept_t* copy_() const = 0;
    virtual void draw_() const = 0;
};

template <typename T>
struct model_t : concept_t
{
    explicit model_t(T x) : value_m(move(x)) {}
    concept_t* copy_() const { return new model_t(*this); }
    void draw_() const { draw(value_m); }

    T value_m;
};
```
Using our Poly Drawable Type

template <typename T> void draw(const T& x) { cout << x << endl; }

template <typename T> void draw(const vector<T>& x) {
    typedef typename vector<T>::const_iterator iterator_t;
    cout << "<vector>" << endl;
    for (iterator_t f(x.begin()), l(x.end()); f != l; ++f) {
        draw(*f);
    }
    cout << "</vector>" << endl;
}

int main() {
    vector<object_t> x;

    x.push_back(object_t(10));
    x.push_back(object_t(string_t("Hello World!")));
    x.push_back(object_t(x));
    x.push_back(object_t(string_t("Another String!")));

    draw(x);
    return 0;
}
Results

<vector>
  10
   Hello World!
<vector>
  10
   Hello World!
</vector>
  Another String!
</vector>

Indenting Added for clarity
Summary

- Non-Intrusive – client need only satisfy requirements
- Existing types can be used in a polymorphic fashion without wrapping
- Cost of virtual dispatch the same – but only required when object used in a polymorphic setting
- Client isn’t burdened by managing pointers – can use efficiently with containers and algorithms

- The Poly Library provides facilities for:
  - Virtualization of the properties of Regular
  - Refinement
  - Dynamic Type Information
template <typename T>
void draw(const copy_on_write<T>& x) { draw(x.read()); }

int main(){
    typedef copy_on_write<object_t> cow_t;

    vector<cow_t> x;

    x.push_back(cow_t(object_t(10)));
    x.push_back(cow_t(object_t(string_t("Hello World!"))));
    x.push_back(cow_t(object_t(x)));
    x.push_back(cow_t(object_t(string_t("Another String!"))));

    draw(x);

    return 0;
}
STL provides sequence and associative containers and algorithms

Because the STL data types are Regular they can be composed to create new structures

Not all structures are best represented by composition

Hierarchies can be represented through containment
  - as we saw with object_t

Other representations provide other advantages
Forest

Diagram:
- Forest
  - A
    - B
    - C
  - E
    - D
Forest (full-order traversal)
Forest (pre-order traversal)
Forest (post-order traversal)
Forest (child traversal)
Forest (insert and erase)
template <typename T> // T models Regular
ostream& operator<<(ostream& stream, const forest<T>& x)
{
    typedef typename forest<T>::const_iterator iterator_t;
    typedef depth_fullorder_iterator<iterator_t> depth_iterator_t;

    for (depth_iterator_t f(begin(x)), l(end(x)); f != l; ++f)
    {
        for (size_t n(f.depth()); n != 0; --n) stream << "\t";
        stream << (f.edge() ? "<" : "/") << *f << "" << endl;
    }

    return stream;
}
Example

```cpp
int main()
{
    typedef forest<const char*> forest_t;
    typedef forest_t::iterator  iterator_t;

    forest_t x;

    iterator_t i = x.insert(x.end(), "me");
x.insert(x.end(), "brother");
    ++i;
    iterator_t j = x.insert(i, "son");
    ++j;
x.insert(j, "grandson");
x.insert(i, "daughter");

    cout << x;
    return 0;
}
```
Example

```c++
int main()
{
    typedef forest<const char*> forest_t;
    typedef forest_t::iterator  iterator_t;
    forest_t x;

    iterator_t i = x.insert(x.end(), "me");
    x.insert(x.end(), "brother");
    ++i;
    iterator_t j = x.insert(i, "son");
    ++j;
    x.insert(j, "grandson");
    x.insert(i, "daughter");

    cout << x;

    return 0;
}
```
Example

```cpp
int main()
{
    typedef forest<const char*> forest_t;
    typedef forest_t::iterator iterator_t;

    forest_t x;

    iterator_t i = x.insert(x.end(), "me");
    x.insert(x.end(), "brother");
    ++i;
    iterator_t j = x.insert(i, "son");
    ++j;
    x.insert(j, "grandson");
    x.insert(i, "daughter");

    cout << x;

    return 0;
}
```
```cpp
int main()
{
    typedef forest<const char*> forest_t;
    typedef forest_t::iterator iterator_t;

    forest_t x;

    iterator_t i = x.insert(x.end(), "me");
    x.insert(x.end(), "brother");
    ++i;
    iterator_t j = x.insert(i, "son");
    ++j;
    x.insert(j, "grandson");
    x.insert(i, "daughter");

    cout << x;

    return 0;
}
```
Example

```cpp
int main()
{
    typedef forest<const char*> forest_t;
    typedef forest_t::iterator iterator_t;

    forest_t x;

    iterator_t i = x.insert(x.end(), "me");
    x.insert(x.end(), "brother");
    ++i;
    iterator_t j = x.insert(i, "son");
    ++j;
    x.insert(j, "grandson");
    x.insert(i, "daughter");

    cout << x;

    return 0;
}
```
Example

```cpp
int main()
{
    typedef forest<const char*> forest_t;
    typedef forest_t::iterator iterator_t;

    forest_t x;

    iterator_t i = x.insert(x.end(), "me");
x.insert(x.end(), "brother");
++i;
    iterator_t j = x.insert(i, "son");
++j;
x.insert(j, "grandson");
x.insert(i, "daughter");

    cout << x;

    return 0;
}
```
Example

```cpp
int main()
{
    typedef forest<const char*> forest_t;
    typedef forest_t::iterator iterator_t;

    forest_t x;
    iterator_t i = x.insert(x.end(), "me");
    x.insert(x.end(), "brother");
    ++i;
    iterator_t j = x.insert(i, "son");
    ++j;
    x.insert(j, "grandson");
    x.insert(i, "daughter");
    cout << x;
    return 0;
}
```
Example

```c
int main()
{
    typedef forest<const char*> forest_t;
    typedef forest_t::iterator iterator_t;

    forest_t x;

    iterator_t i = x.insert(x.end(), "me");
    x.insert(x.end(), "brother");
    ++i;
    iterator_t j = x.insert(i, "son");
    ++j;
    x.insert(j, "grandson");
    x.insert(i, "daughter");

    cout << x;

    return 0;
}
```
```cpp
int main()
{
    typedef forest<const char*> forest_t;
    typedef forest_t::iterator iterator_t;

    forest_t x;

    iterator_t i = x.insert(x.end(), "me");
    x.insert(x.end(), "brother");
    ++i;
    iterator_t j = x.insert(i, "son");
    ++j;
    x.insert(j, "grandson");
    x.insert(i, "daughter");

    cout << x;

    return 0;
}
```
Example

```c++
int main()
{
    typedef forest<const char*> forest_t;
    typedef forest_t::iterator iterator_t;

    forest_t x;

    iterator_t i = x.insert(x.end(), "me");
    x.insert(x.end(), "brother");
    ++i;
    iterator_t j = x.insert(i, "son");
    ++j;
    x.insert(j, "grandson");
    x.insert(i, "daughter");

    cout << x;

    return 0;
}
```

Declarative UI with ASL

- Introduction
  - What a User Interface Is
  - Identifying UI Mechanisms
  - What MVC Is
  - Property Models and Layouts Libraries
  - Modeling the Form
  - Presenting the Form

- Property Model Basics
  - An Overview of The Property Model Syntax
  - CEL expression and the Begin Inspector
  - Invariants & Dependency Tracking
  - Relationships & Logic

- Layout Library Basics
  - An Overview of the Layout Library Syntax
  - Placement and Alignment
  - Spacing, Margins, and Indenting
  - Guides
  - Optional and Panel

- Advanced Topics
  - Scripting and Localization
  - How Layouts Work
    - What you can't do
  - How Property Models Work
    - What you can't do
What is a User Interface?

Discussion
What a User Interface Is

- **Definition:** A *User Interface* (UI) is a system for assisting a user in selecting a function and providing a valid set of parameters to the function.
- **Definition:** A *Graphical User Interface* (GUI) is a visual and interactive UI.
Mechanisms to Assist the User

Discussion
UI Mechanisms

- **Context**
  - Current Document, Selection, Tools, Modal Dialogs
  - Context Provides a Function or One or More parameters to the Function
    - The current item is referred to as the subject
    - The selected function is the verb

- **Sentences**
  - subject-verb(function)-[object]
  - Drag and Drop, Cut/Copy/Paste

- **Constraints**
  - Disabled Options, Rejecting Invalid Input, Modality

- **Consistency**
UI Mechanisms (Continued)

- **Interactivity**
  - Tracking: \(\approx 1/30 \) s
  - Acknowledge: \(\approx 1/5 \) s
  - Confirmation: \(\approx 1 \) s

- **Precognition**
  - Specifying Parameters in Terms of Desired Results:
    - Compress this movie to fit on a DVD
    - Scale this image to fit the Page

- **Time-Travel**
  - Undo, Preview, Non-Destructive Editing

- **Metaphors**
  - Using knowledge transference
Introduction

Demo
Model-View-Controller

- View & Controller Logically Separate
- Most Descriptions get MVC Wrong - see Design Patterns or Smalltalk, not Apple or Microsoft.
- CMV Would be a Better Term
Model View Controller
Model-View-Controller
Property Models and Layouts Libraries

- Property Model Library is *only* concerned with the model portion
  - It is not the only way to construct a model

- Layout Library is *only* concerned with how the view portions are positioned in a coordinate space

- Within our Layout Descriptions we'll also providing *binding* to connect the widgets to the model
  - It is important to note that the layout library does not have any built in knowledge about the widgets - we provide a sample set of widgets but they are not complete implementations.
Relation to MVC
Property Model Basics
sheet my_sheet
{
    interface:
    team_1: "Giants";
    team_2: "Patriots";
    score_1: 0;
    score_2: 0;

    output:
    result <= { 
        team_1: team_1, team_2: team_2, 
        score_1: score_1, score_2: score_2 };
}
Property Model Descriptions

- **Interface Cells**
  - Optional Initializer and Expression
    
    ```
    score_1: 0 <= score_2 * 2;
    ```

- **Output Cells**
  - Require Expression
    
    ```
    result <= [score_1, score_2];
    ```
CELEXpressions

- **Built-In Data Types**
  - **number**: -17.3
  - **string**: "Hello" ' world!'
  - **name**: @identifier
  - **boolean**: true
  - **array**: [false, "Test", @key]
  - **dictionary**: {key_1: "Value", key_2: 10}
  - **empty**: empty

- **Variables and Function**
  - **variable**: score_1
  - **function**: max(10, score_1)
    scale(m: base, x: 10, b: offset)
CEL Expressions

- **Operators**
  - number: *, /, +, -
  - number: <, >, <=, >=
  - boolean: !, &&, ||
  - any: ==, !=
  - array: [number_expression]
  - dictionary: [name_expression], .
  - any: expression ? expression : expression
  - empty: empty

- **C order of Precedence**

- **Example**

```plaintext
{ width: 10, height: 20 }[ p ? @width : @height]
```
Property Model Descriptions

- **Invariant Cells**
  - Requires Boolean Expression
    
    ```
    invariant:
    check <= a < b;
    ```
  
- **The pre-conditions to a function are an invariant of the functions arguments**

- **Cells that contribute to an invariant are poison**

- **Cells derived from poison are invalid**
Property Model Descriptions

- **Logic Cells**
  - **Requires Expression**
    
    ```
    logic:
    rate <= a * b;
    ```
  
  - **A logic cell is simply a named expression**

- **Relate Expression**
  
  ```
  logic:
  relate {
    a <= b * c;
    b <= a / c;
    c <= a / b;
  }
  ```

- **N-Way, Exactly One Expression Is Executed For A Given State**
Mini-Image Size Example
Declarative Solution using the Property Model Library

```plaintext
sheet mini_image_size
{
    input:
        original_width : 5 * 300;
        original_height : 7 * 300;
    interface:
        constrain : true;
        width_pixels  : original_width  <= round(width_pixels);
        height_pixels : original_height <= round(height_pixels);
        width_percent;
        height_percent;
    logic:
        relate {
            width_pixels  <= round(width_percent * original_width / 100);
            width_percent <= width_pixels * 100 / original_width;
        }
        relate {
            height_pixels <= round(height_percent * original_height / 100);
            height_percent <= height_pixels * 100 / original_height;
        }
        when (constrain) relate {
            width_percent  <= height_percent;
            height_percent <= width_percent;
        }
    output:
        result <= { height: height_pixels, width: width_pixels };
}
Imperative Solution to Mini-Image Size
Event Flow in a Simple User Interface
Layout Library Basics
layout my_dialog
{
  interface:
    display : true;
  constant:
    dialog_name : "My Dialog";

  view dialog(name: dialog_name) {
    reveal(name: "Display", bind: @display);
    optional(bind: @display) {
      button(name: "OK");
    }
  }
}
Placement and Alignment

- Placement is a container property
  - placement: place_row, place_column, place_overlay
  - The containers row(), column(), and overlay() are non-creating containers with the corresponding placement.

- Alignment is a general property that applies to horizontal and vertical
  - horizontal: align_left, align_right, align_center, align_proportional, align_fill
  - vertical: align_top, align_bottom, align_center, align_proportional, align_fill

- Alignment of children can be imposed from container
  - child_horizontal:
  - child_vertical:

- Tip: If widgets are stuck top/left, it is likely because the container they are in isn't using align_fill.
Spacing, Margins, Indenting

- **Spacing is a container property**
  - spacing: number
  - spacing: array
  - The spacing between each element in the container

- **Margin is a container property**
  - margin: number
  - margin: [top, left, bottom, right]

- **Indent is a general property**
  - Indent: number
  - The indent applies to the horizontal position of an item in a column and vertical position of an item in a row and is relative to the left or right alignment

- **Tip:** Define meaningful constants for these elements - don't use raw values and don't use to "fake" alignment.
- Guides are Defined By Widgets (Currently)

- There are (Currently) Two Guide Types: @guide_baseline, @guide_label

- Guides Propagation Can Be Suppressed:
  - guide_mask: [@guide_xxxx]
  - The default mask for columns is [@guide_baseline]

- Guides Can Also Be Balanced Within A Container
  - guide_balance: [@guide_xxxx]

- Guides only apply to items which are aligned left/right or top/bottom or filled. Fill left or right is determined by widget (and may vary by local).

- Tip: Guides can be allowed to propagate from overlays to get consistent column widths on tab panels.
Optional and Panel

- `optional()` and `panel()` are containers whose visibility can be bound
- An `optional()` container is removed from the layout when hidden
- A `panel()` remains part of the layout when hidden

*Tip: Use `panel()` with a `tab_group()`. A `tab_group()` is like a popup but is also a container that defaults to `place_overlay`.*

```javascript
tab_group(bind: @x,
    item: [{name: "tab 1", value: @tab_1},
            {name: "tab 2", value: @tab_2}]) {
    panel(bind: @x, value: @tab_1) { /*...*/ }
    panel(bind: @x, value: @tab_2) { /*...*/ }
}
```
Scripting and Localization

- Contributing values form the basis for intelligent recording
  - Difference between "fixed" values and contributing captures "intent"
- Same model is used for playback - handling all script validation
- Model assists script writers in the same way it assists users - letting them specify the parameters in terms they understand

- ASL contains an experimental xstring library:
  
  button(name: localize("<xstr id='ok'>OK</xstr>");
How Layouts Work

- A layout is a container of *placeable* objects
- When a description is parsed a hierarchy of placeable objects is stored in the layout
- The basic algorithm is:
  - Gather horizontal metrics of each item in the hierarchy, depth first post order
  - Solve the horizontal layout
  - Gather vertical metrics - providing final horizontal metrics
  - Solve the vertical layout
  - Place each item
How Layouts Work

[Diagrams showing layout structures]
How Layouts Work

alignment on shorter paths

if B and C have compatible guides they collapse to one node
What you can't do

- Layouts must be able to be decomposed into rows, columns, and overlays

- No space filling or best fit algorithms

- You can plug-in your own layouts if they can behave as a placeable object.
A property model is a container of cells, relationships, views and controllers.

When the description is parsed, cells and relationships are added.

Views and controller are added from the layout description.

Each cell attached to a relationship has a priority as well as a value, priority is usually based on how recently the element changed.
How Property Models Work

- The basic algorithm is:
  - Calculate the predicates for any conditional relate clauses
    - Predicates cannot be involved in relate clauses
  - Flow the active relate clauses using the priority on the cells
    - After this point, the flow will be use to direct calculations
    - Flow and calculate run in opposite directions on the graph.
  - Calculate the invariants
    - If an invariant is false, any reached source is marked as poison
  - Calculate the output expressions
    - Reached sources are marked enabled
    - If a reached source is poison result is marked invalid
  - Calculate any remaining interface cells to which a view is attached
What you can't do

- There are many other types of models that the property model library can't handle - some of the more common ones:
  - Sequences (manipulating lists of elements)
    - Although the property model can describe invariants on the sequence and pre- and post-conditions on the functions that manipulate it.
  - Grammars
    - The property model library is not a parser
  - Triggers - imperative actions
    - There is no way to say "when this happens do this"
- The property model library cannot handle distributing values (yet)
  - From our exercise - there is no way to construct a UI which given a final score calculates how many tds, field goals, and extra points are needed to reach it.
Closing Comments

- Website [http://stlab.adobe.com](http://stlab.adobe.com)
- Don't be afraid to ask questions - subscribe to our mailing list
- Please contribute to ASL - our charter is to improve how software is written - by contributing you will learn and help others
  - We prefer small contributions - contribute the big functions when they become small functions leveraging the rest of the library
Revolutionizing how the world engages with ideas and information