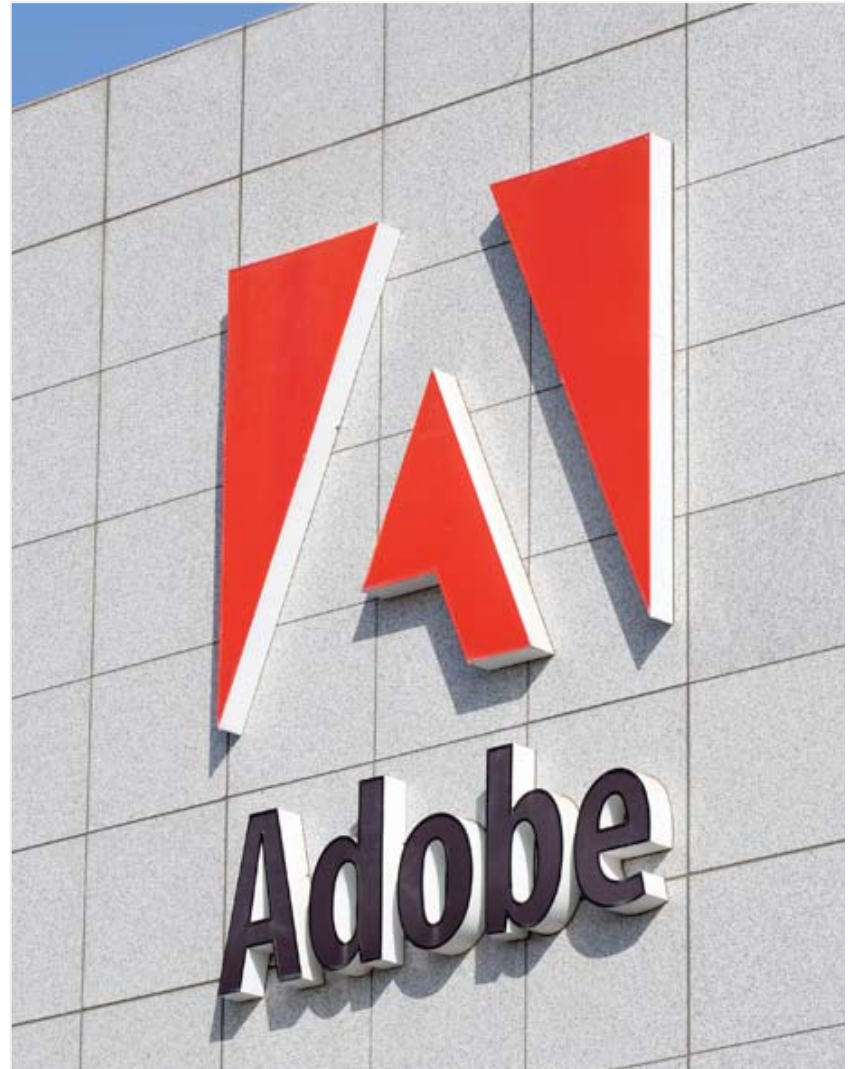


A Possible Future of Software Development

Sean Parent

Principal Scientist & Manager
Software Technology Lab

May 16, 2007



Adobe Today

Worldwide Offices



Corporate Headquarters – San Jose, California



Key Statistics

Adobe
FY 2006 Revenue

\$2.575B

Adobe
Q1 2007 Revenue

\$649.4M

Years in Business

25

Employees

6,000

Adobe Product Lines

Creative Solutions

- Creative Suite
- Photoshop
- InDesign
- Premiere Pro
- After Effects
- Flash
- Dreamweaver

Enterprise & Mobile

- Acrobat
- Acrobat Connect
- LiveCycle
- Flex
- Flash Lite
- FlashCast
- Reader LE

Technologies

- Portable Document Format (PDF)
- Reader
- Flash Player
- Postscript

Engineering Team Structure

- Product Line:
 - Photoshop, Acrobat, InDesign, ...
- Products:
 - Photoshop CS3, Photoshop CS3 Extended, Photoshop Elements, Photoshop Lightroom, ...
- Product Team:
 - Developers ≈20
 - Testers ≈30
 - User Interface Designers ≈1
- Shared Technology Groups: ≈20
 - Libraries for Vector Graphics, Type, Color, Help, Localization, XML Parsing, File Handling, etc.

Development Process

- Process is constrained by business model
- Schedule driven, incremental development model on 18-24 month cycles
 - Larger products and suites forced toward waterfall model
 - Press for manuals must be reserved up to 5 months in advance
- Most products ship simultaneously for Macintosh and Windows in English, French, German, and Japanese
 - Other languages follow shortly to about 24 languages

The Analysts Future

- **“Best practices”**, methodologies, and process are changing continuously
- Trend towards **Java** and C# languages
 - As well as JavaScript and VisualBasic is still strong
 - Java still has only a small presence on the desktop
 - Object Oriented is ubiquitous
- **XML** growing as data interchange format
- Web services
- Open source
 - Foundation technologies commoditized

The Analysts Future

- “Organizations need to integrate security best practices, security testing tools and security-focused processes into their software development life cycle. Proper execution improves application security, reduces overall costs, increases customer satisfaction and yields a more-efficient SDLC.”
 - Gartner Research, February 2006
- “Microsoft has been slowly moving to a new development process that will affect how partners and customers evaluate and test its software... The new process should help Microsoft gain more feedback earlier in the development cycle, but it won’t necessarily help the company ship its products on time or with fewer bugs.”
 - Directions on Microsoft, March 2006

Why Status Quo Will Fail

- “I’ve assigned this problem [binary search] in courses at Bell Labs and IBM. Professional programmers had a couple of hours to convert the description into a programming language of their choice; a high-level pseudo code was fine... Ninety percent of the programmers found bugs in their programs (and I wasn’t always convinced of the correctness of the code in which no bugs were found).”
- Jon Bentley, Programming Pearls, 1986

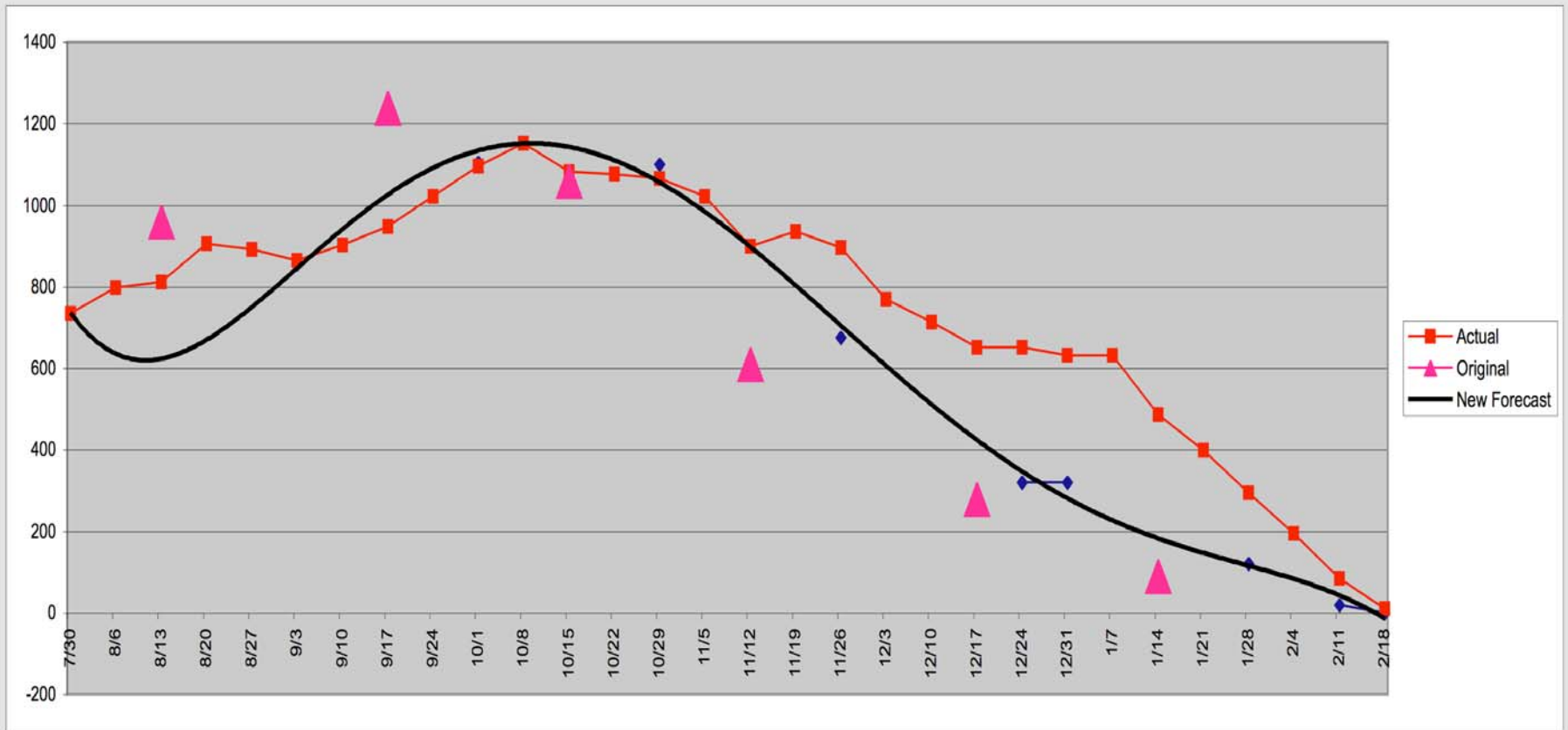
Binary Search Solution

```
int* lower_bound(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;
}
```

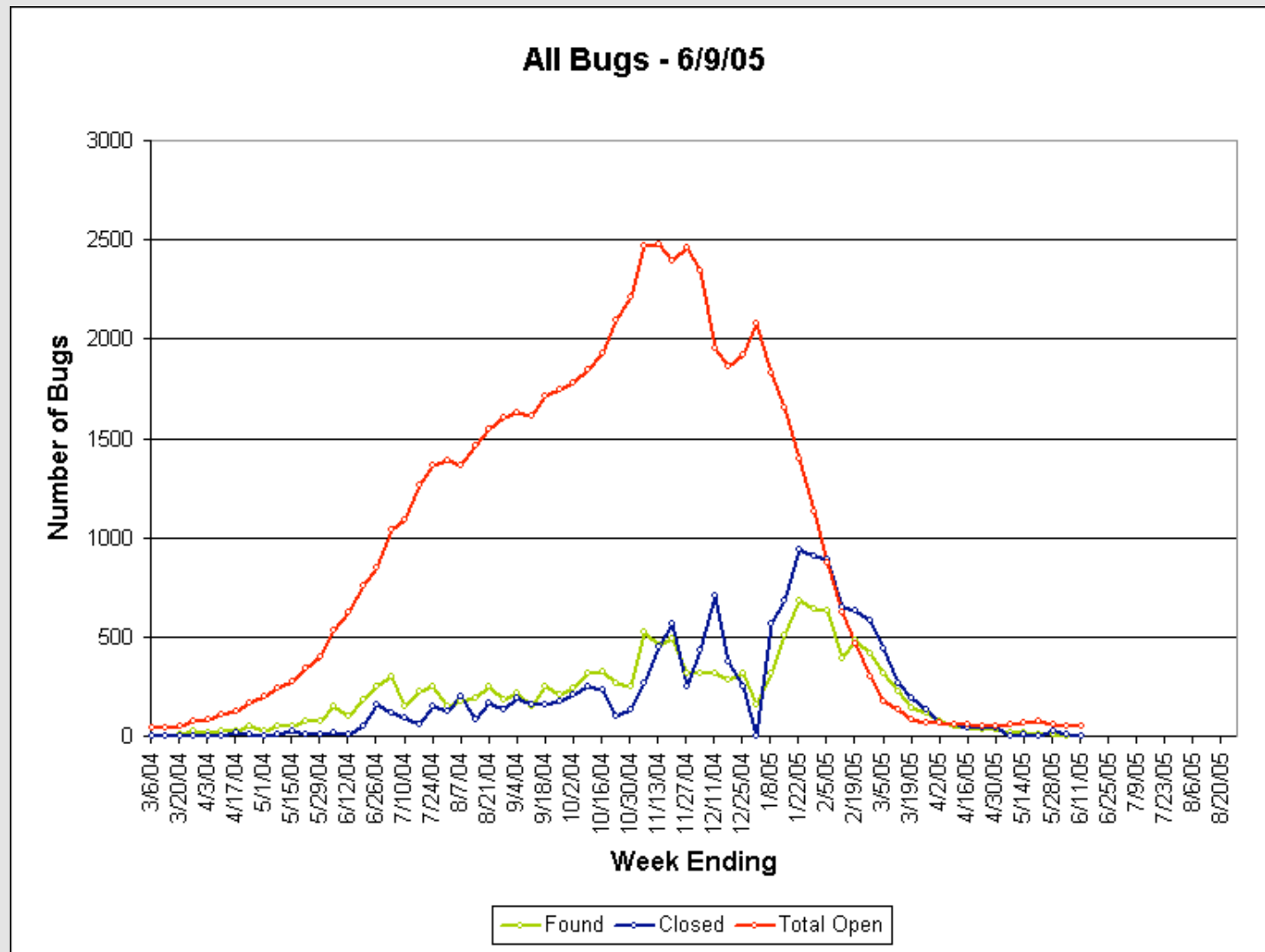
Question: If We Can't Write Binary Search...

- Jon Bentley's solution is considerably more complicated (and slower)
- Photoshop uses this problem as a take home test for candidates
 - **More than 90% of candidates fail**
- Our experience teaching algorithms would indicate that more than 90% of engineers, regardless of experience, cannot write this simple code
- ...then how is it possible that Photoshop, Acrobat, and Microsoft Word exist?

Bugs During Product Cycle



Bugs During Product Cycle



Answer: Iterative Refinement.

- Current programming methodologies lend themselves to iterative refinement.
- We don't solve problems, we approximate solutions.

Writing Correct Algorithms

- Study how to write correct algorithms
- Write algorithms once in a general form that can be reused
- Focus on the common algorithms

Generic Programming

- Start with a concrete algorithm
- Refine the algorithm, reducing it to its minimal requirements
- Clusters of *related* requirements are known as Concepts
- Define the algorithms in terms of Concepts - supporting maximum reuse
- Data structures (containers) are created to support algorithms

Programming is Mathematics

- Generic Programming

- Semantic Requirement
- Concept
- Model (types model concepts)
- Algorithms
- Regular Function
- Complexity

- Mathematics

- Axiom
- Algebraic Structure
- Model
- Theorems
- Function
- _____

- Refined Concept - a Concept defined by adding requirements to an existing concept.

- monoid: semigroup with an identity element
- BidirectionalIterator: ForwardIterator with constant complexity decrement

- Refined Algorithm - an algorithm performing the same function as another but with lower complexity or space requirements on a refined concept

Syntax of Concepts - According to the Standard

Concept Regular

expression	return type	post-condition
$T(t)$		t is equivalent to $T(t)$
$T(u)$		u is equivalent to $T(u)$
$t.\sim T()$		
$\&t$	T^*	denotes address of t
$\&u$	$\text{const } T^*$	denotes address of u

Table 30 - CopyConstructable

$t = u$	$T\&$	t is equivalent to u
---------	-------	--------------------------

Table 64 - Assignable

$a == b$	bool	$==$ is an equivalence relation
----------	---------------	---------------------------------

Table 3 – EqualityComparable

Semantics of Concepts

- A Concept is an algebraic structure formed of *connected* requirements
- Equality is a unique equivalence relation...

$$\forall a : a = a \text{ (reflexive)}$$

$$a = b \Rightarrow b = a \text{ (symmetric)}$$

$$a = b \text{ and } b = c \Rightarrow a = c \text{ (transitive)}$$

- ...connected to copy and assignment:

$$b \rightarrow a \Rightarrow a = b \text{ (copies are equal)}$$

$$a = b = c, d \neq a, d \rightarrow a \Rightarrow b = c \text{ (copies are disjoint)}$$

- A model of Regular supports equational reasoning with computability

Importance of Concepts

- Concepts enable equational reasoning about code
- Concepts can be applied to any coding *style*
 - Generic programming is not a programming style, it is the mathematics of code
 - [see my talk, *Concept-Based Runtime Polymorphism*, tomorrow to learn how generic programming relates to object oriented programming]
- The success of this approach requires years of individual study and organized effort by the community

Simple Generic Algorithm

```
template < typename I,    // I models RandomAccessIterator
          typename T,    // T models Regular
          typename O>    // O models BinaryFunction on T
// requires value_type(I) == T
// precondition: o provides a strict weak ordering on T
// precondition: [first, last) is in non-decreasing order by o

I lower_bound(I first, I last, T x, O o)
{
    while (first != last) {
        I middle = first + (last - first) / 2;
        if (o(*middle, x)) first = middle + 1;
        else last = middle;
    }
    return first;
}
```

Future of Concepts

- Language support for Concepts
 - This is a large area of focus for the C++ committee
- Learn to teach programming in terms of Concepts
- Extending Concepts to runtime
 - Replacing inheritance as a mechanism for polymorphism
 - *[see my talk tomorrow on this topic]*
- Construct a large library of algorithms and containers
 - STL is only a beginning - must be considered an example

Question: Is this enough to build an application?

**Conjecture: All problems
of scale become a
network problem**

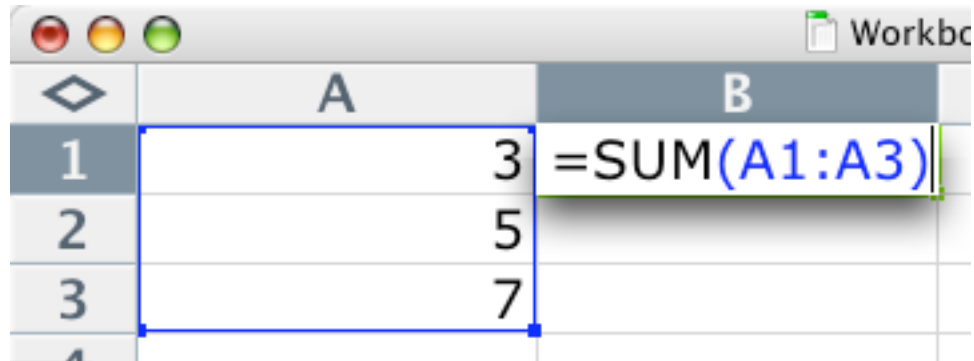
Current Design of Large Systems

- Networks of objects form implicit data structures
- Messaging among objects form implicit algorithms
- Design Patterns assist in reasoning about these systems
 - Local rules which approximate correct algorithms and structures
- Iteratively refine until quality is *good enough*

A "Generic Algorithm" for Solving Large Systems

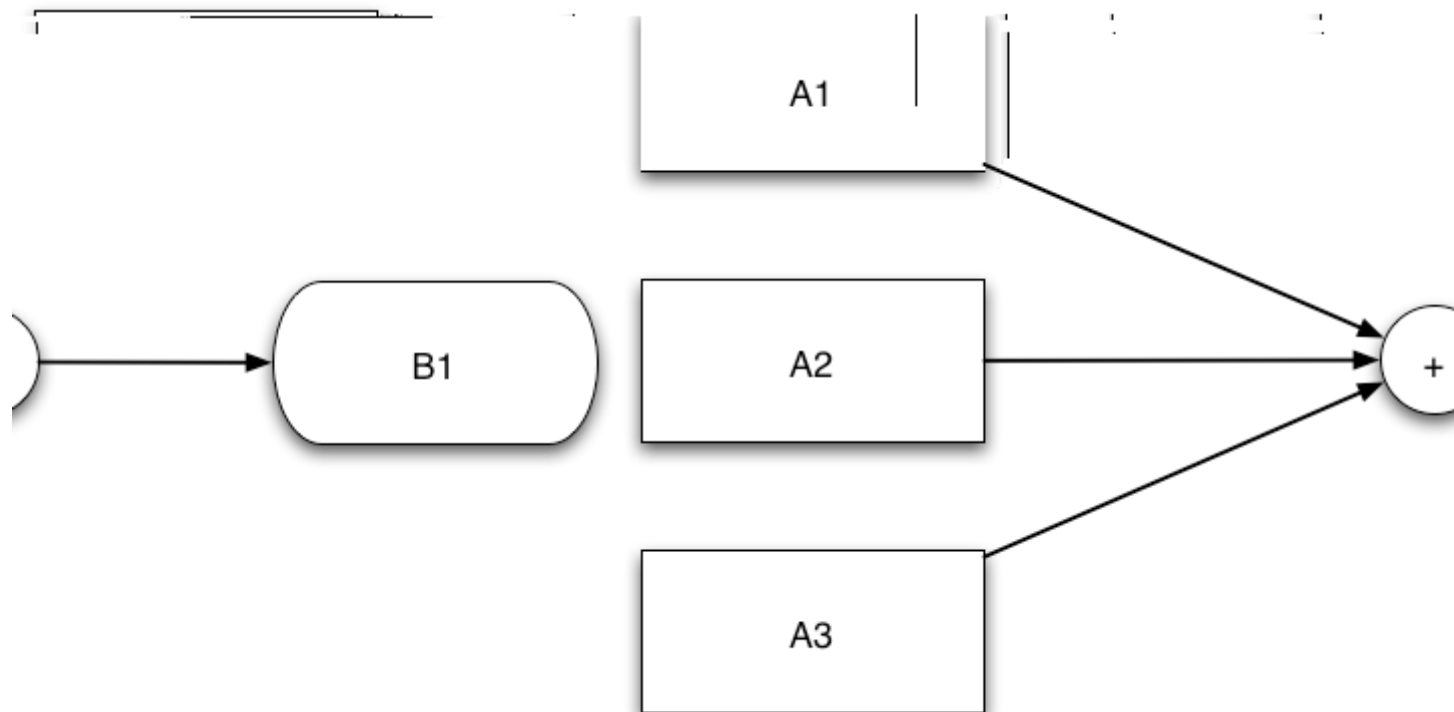
- Identify the components and how they connect and inter-relate
- Organize the system into a directed acyclic graph (DAG)
 - Identify the algorithm in any cyclic areas, factor them into a separate algorithm and replace that area with a node executing the algorithm
- Ensure that for all states of the system a DAG structure is maintained
- Bonus - write a declarative language for describing similar structures in the problem domain

Solving a Spreadsheet



	A	B
1	3	=SUM(A1:A3)
2	5	
3	7	
4		

A DAG Representation



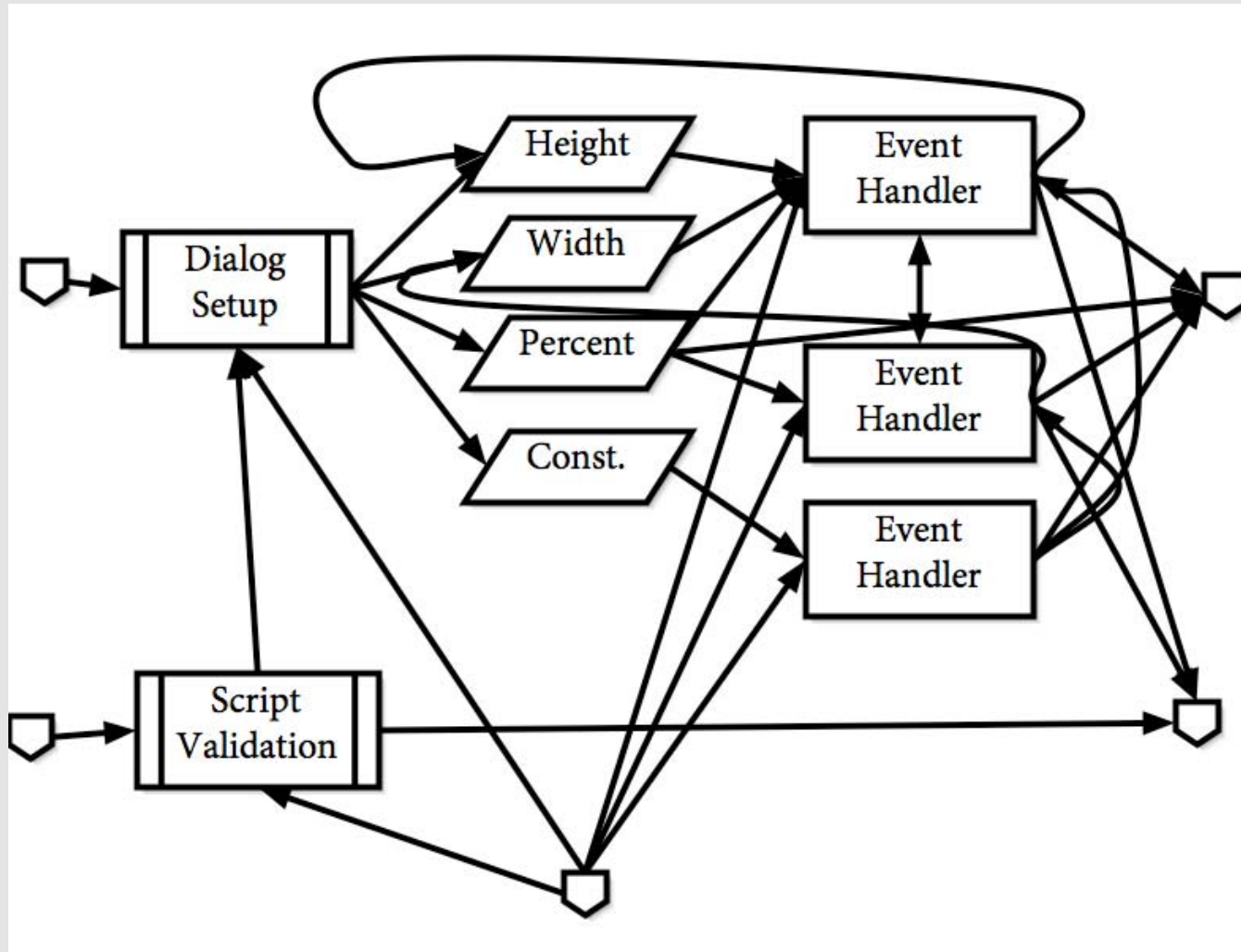
Declarative Programming

- Describe software in terms of rules rather than sequences of instructions
 - Rules define a structure upon which solving algorithms can operate
- Examples of non-Turing complete* systems:
 - Lex and YACC (and BNF based parsers)
 - Sequel Query Language (SQL)
 - HTML (if we ignore scripting extensions)
 - Spreadsheet
- Can be Turing complete (i.e. Prolog)
 - But Turing complete systems lead us back to the complexity of algorithms

**Some of these systems are “accidentally” Turing complete or support extensions that make them Turing complete (such as allowing cycles in a spreadsheet engine). In practice though, this can often be effectively ignored and disallowed*

Property Model Library

Event Flow in a Simple User Interface



Facts:

- 1/3 of the code in Adobe's desktop applications is devoted to event handling logic
- 1/2 of the bugs reported during a product cycle exist in this code

If Writing Correct Algorithms is Difficult...

- ...Writing correct implicit algorithms is very difficult
- We need to study what these implicit algorithms do, and express the algorithms explicitly on declared data structures

STLab Research: “Declarative UI Logic”

- 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.
- We’re starting with what it means to assist the user in providing a valid set of parameters to a function...

Demo

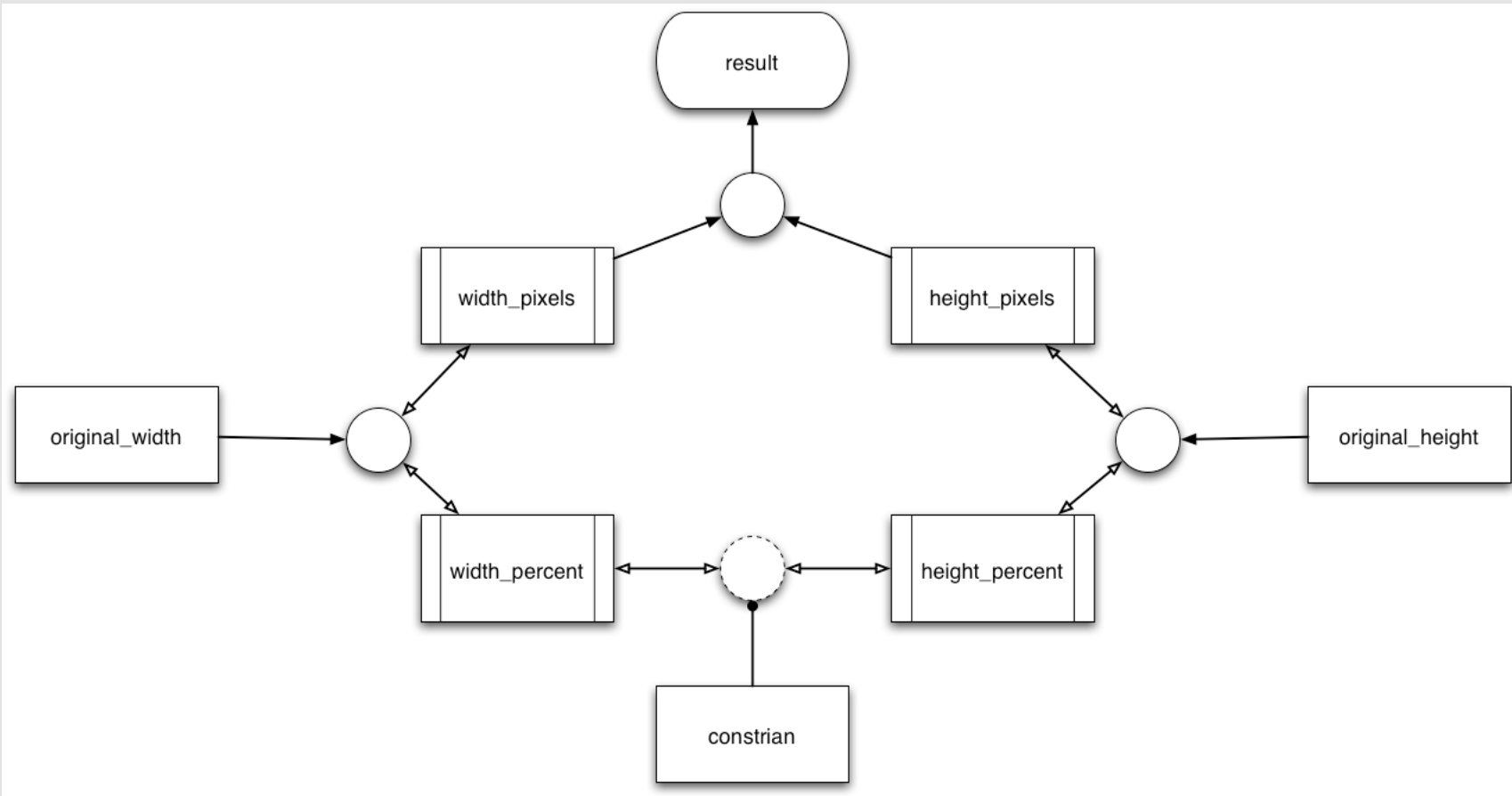
Imperative Solution to Mini-Image Size

[illegible]

Declarative Solution using the Property Model Library

```
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 };
}
```

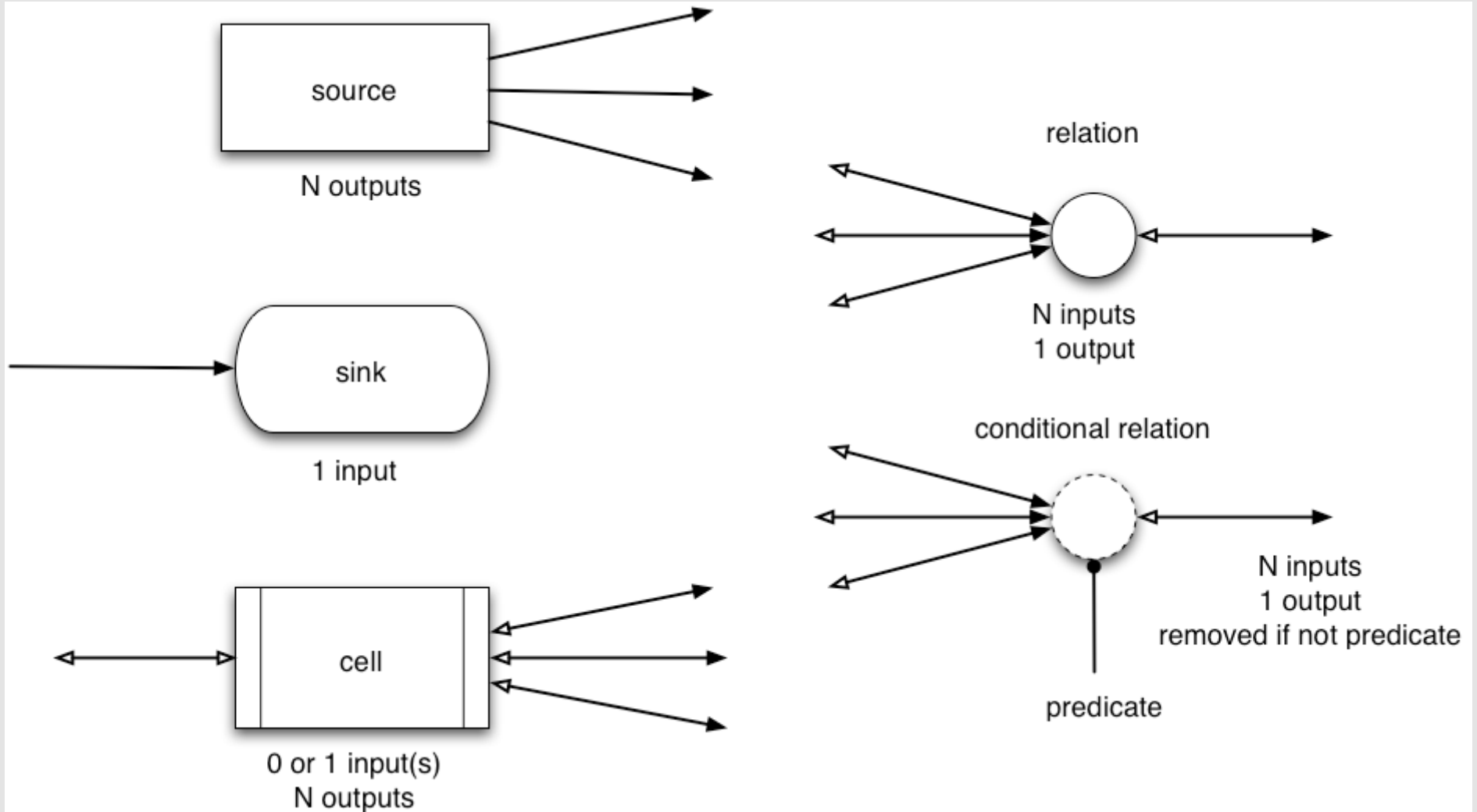
Structure of Simple User Interface



Property Model Structure

- A bipartite graph consisting of values and relations
- Similar in structure to a spreadsheet, except directionality can be determined at runtime based on priority of cells
- Data is flowed from higher priority cells outward to dependent cells
 - a link reversal algorithm is employed to resolve *soft* cycles
- A cell which is acting as a source (no inputs) is said to contribute
- The priority among contributing cells does not change the structure
- All permutations can be checked for hard cycles by checking each permutation of contributing values and pruning out clusters
- Same model used for UI, script recording, and playback

Property Model Primitives



Future of Software Development

- The property model library attacks 30% of the problem
- Estimate 85% of existing code base can be replaced with small declarations and a small library of generic algorithms
 - *My gut-feeling* is that we are a full 2 orders of magnitude off from the minimal expression of any large application
- Formally describe application behavior by expressing algorithm requirements and structure invariants
- Extend the ideas from STL to encompass richer structures
- Extend generic programming to apply to runtime polymorphism
 - Replace inheritance with non-intrusive modeling

More Information

- <http://opensource.adobe.com>
- <http://stepanovpapers.com>
 - Specifically:
 - http://www.stepanovpapers.com/eop/lecture_all.pdf
 - <http://www.stepanovpapers.com/notes.pdf>
 - <http://www.stepanovpapers.com/PAM.pdf>

Better by Adobe.™