Adobe®

Untangling Software

Observations on Architecture

Sean Parent
Sr. Computer Scientist II
September 27th, 2003

" **struc•ture**

n. The way in which parts are connected together to form a whole.

ar•chi•tec•ture

n. Orderly arrangement of parts."

— The American Heritage Dictionary, ©2000

Simple Architectural Problem

Simple Architectural Problem

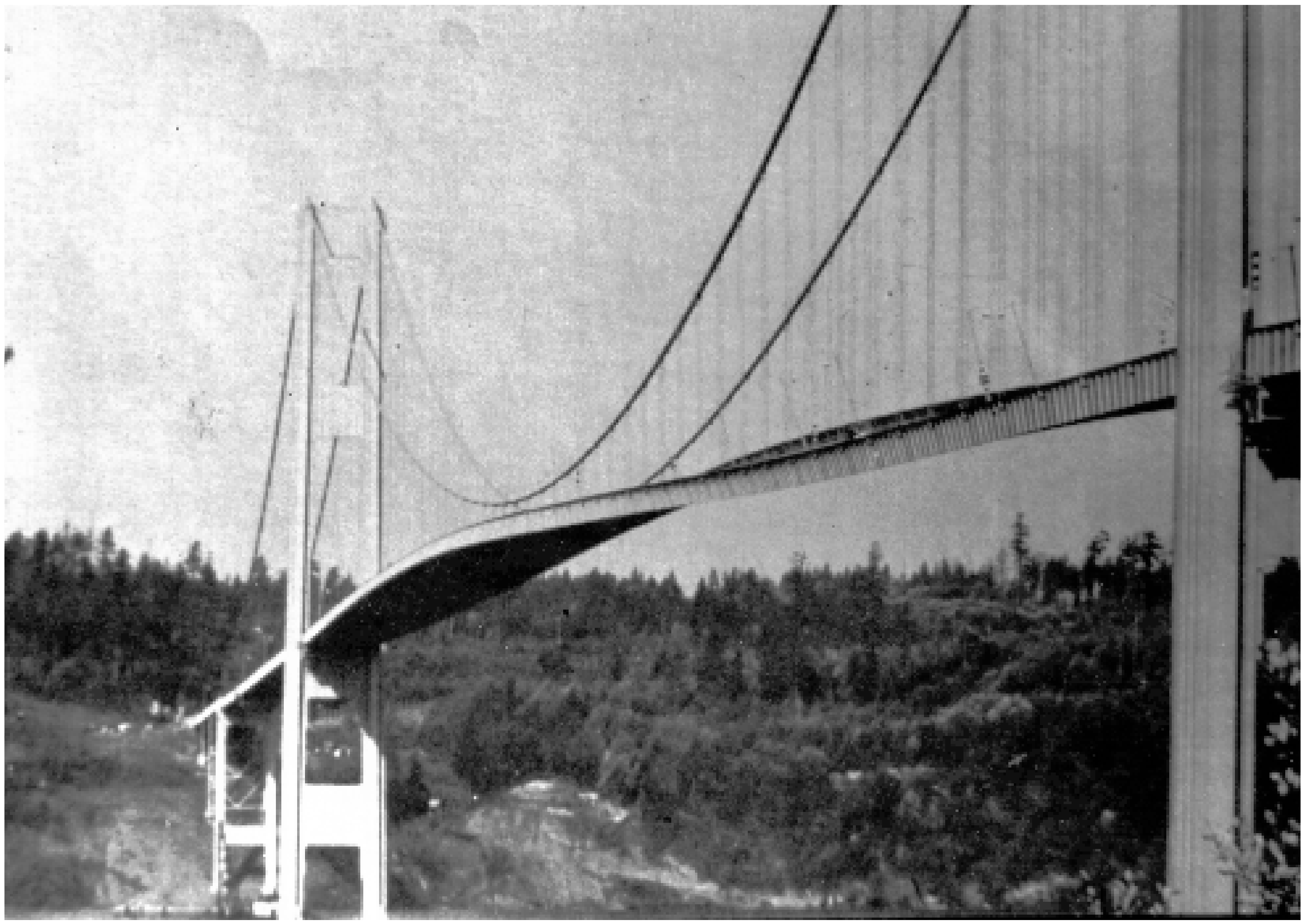
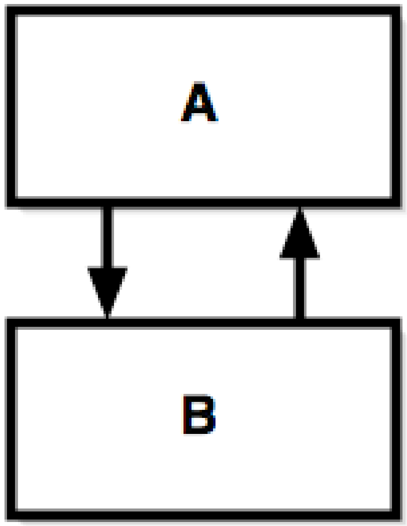
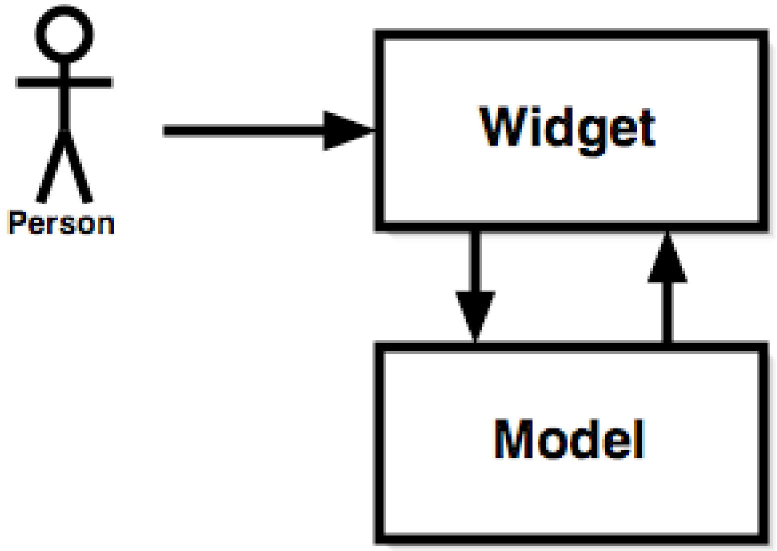


Diagram of The Problem



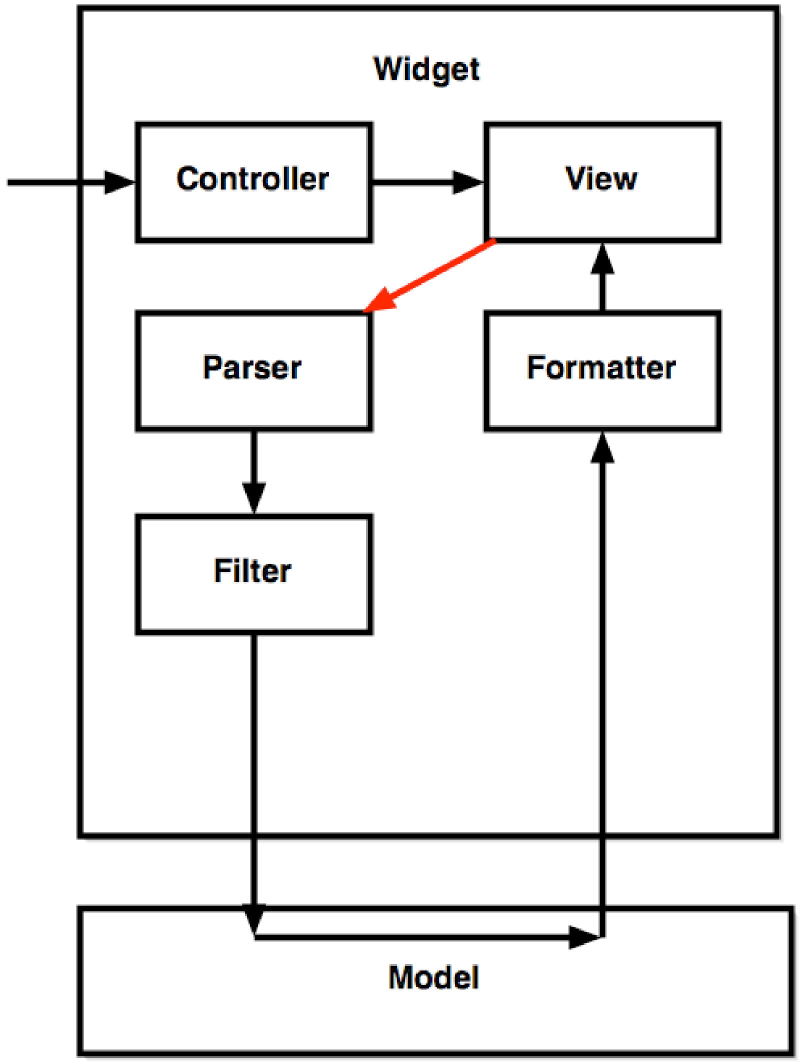
A More Complicated Problem



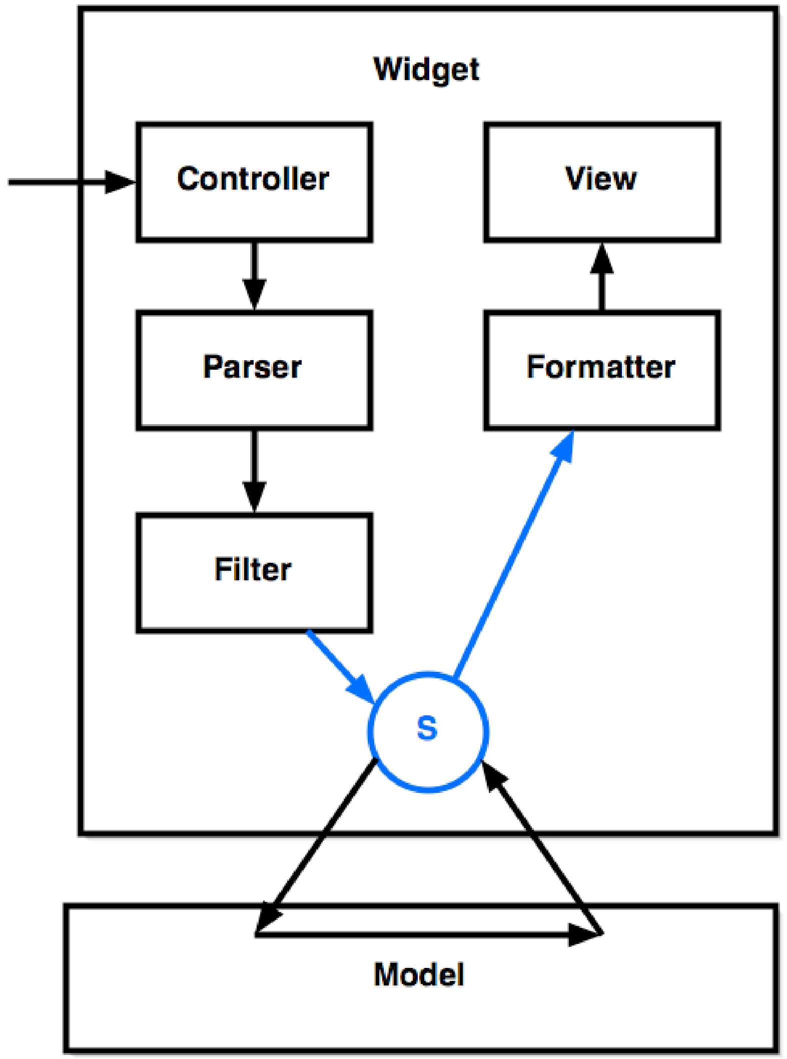
Questions

- **Is the cycle an infinite loop? Recursive? Threaded?**
- **The model and person send to the widget.**
  - Is there a connection inside the widget?
  - Does the model receive when the model sends? When the user sends? Is what the user sends displayed by the widget or what the model sends or both?

Typical Widget Implementation



Another Widget Implementation



Observations

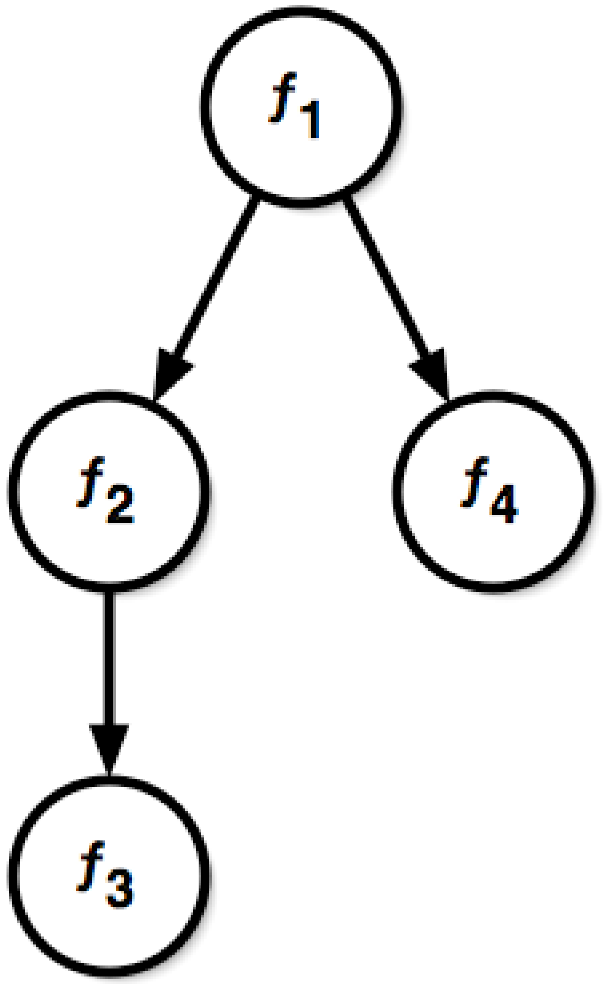
- **To understand structure understand connections**
- **Connections in software can be formed with both state and logic**
  - The Church-Turing Thesis shows these are equivalent when state and logic are computationally complete

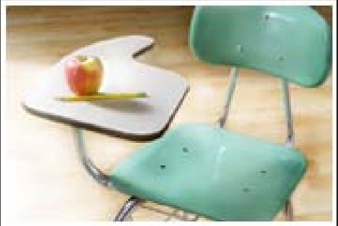
Understanding Logical Connections

- Functional programming imposes a structure on connections
- Understanding the nature of these connections gives insight into working structures

Functional Dependencies

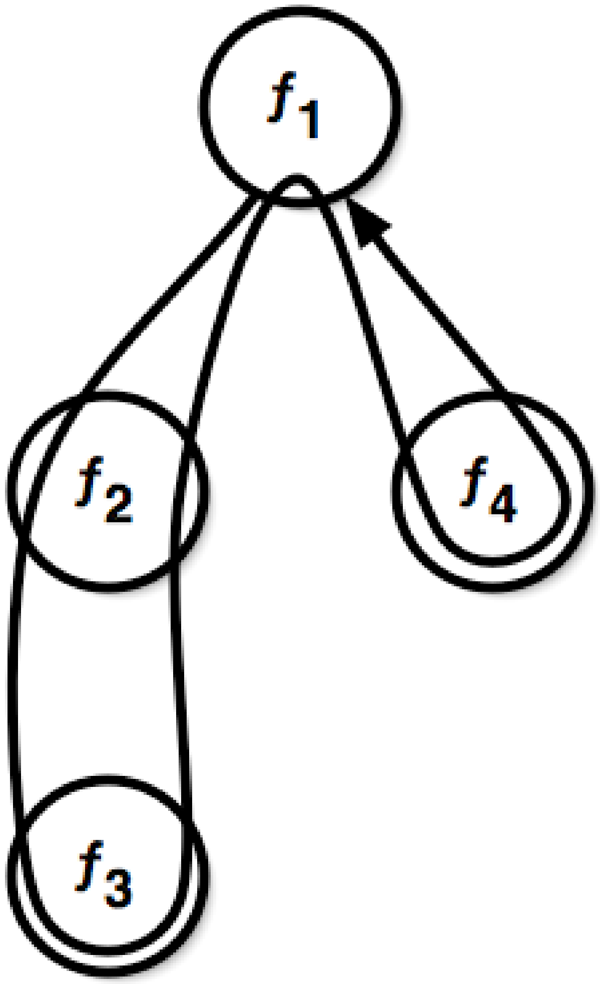
```
T f1(T x) {
    return f4(f2(x));
}

T f2(T x) {
    return f3(x + k);
}

T f3(T x);
T f4(T x);
```


$$\begin{tikzcd}
 & f_1 \arrow[dl] \arrow[dr] & \\
f_2 \arrow[d] & & f_4 \\
f_3 & &
\end{tikzcd}$$



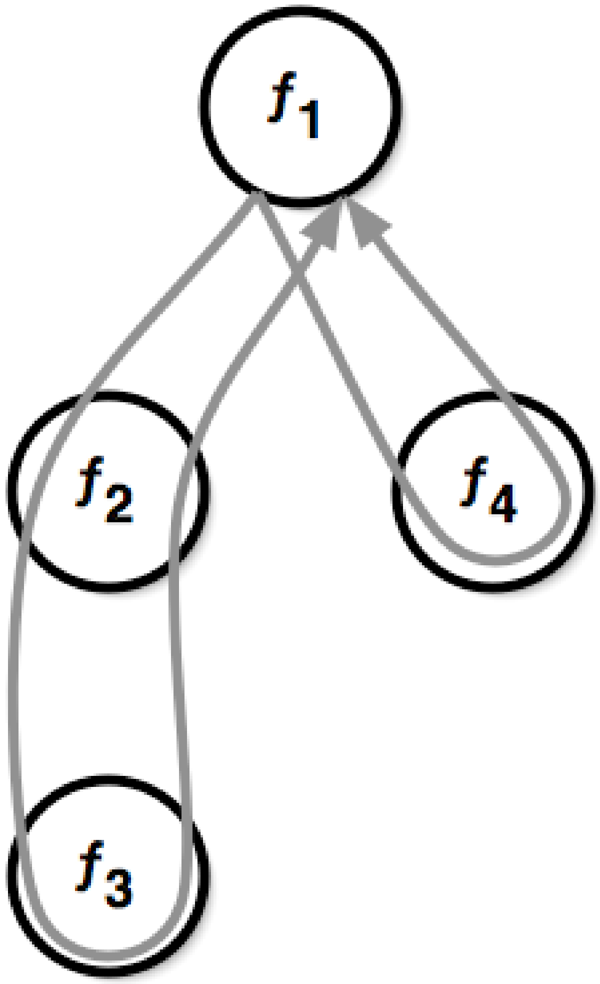
Functional Data Flow

```
T f1(T x) {
    return f4(f2(x));
}

T f2(T x) {
    return f3(x + k);
}

T f3(T x);
T f4(T x);
```



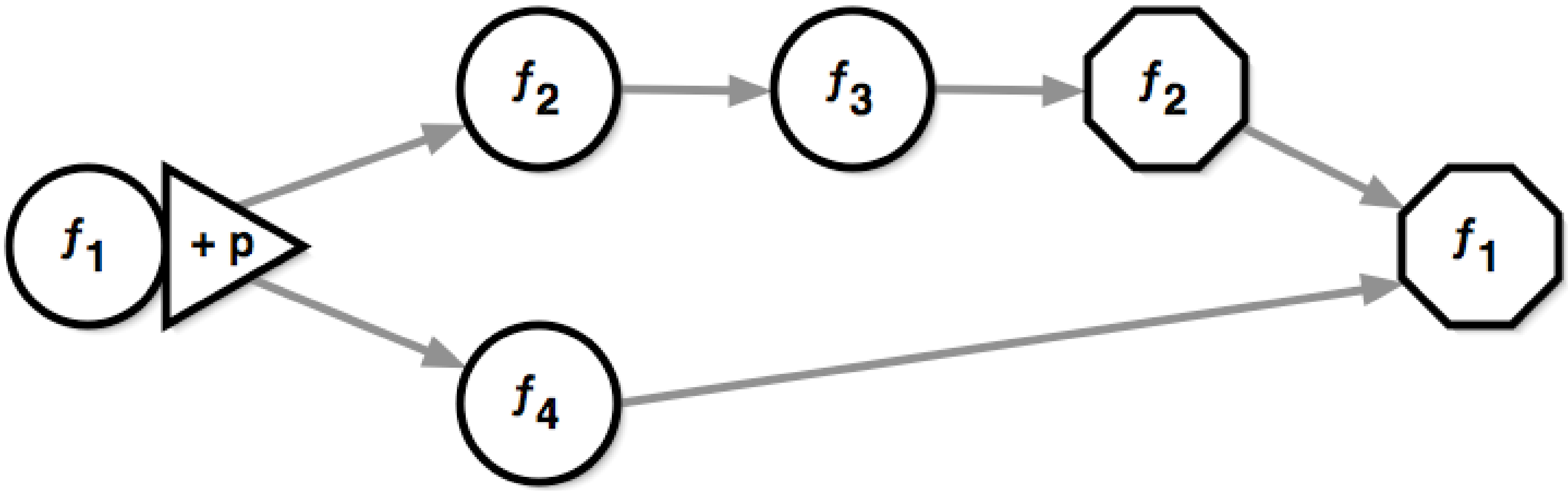
Conditional Data Flow

```
T f1(T x, bool p) {
    return p ? f2(x) : f4(x);
}
T f2(T x) {
    return f3(x + k);
}

T f3(T x);
T f4(T x);
```



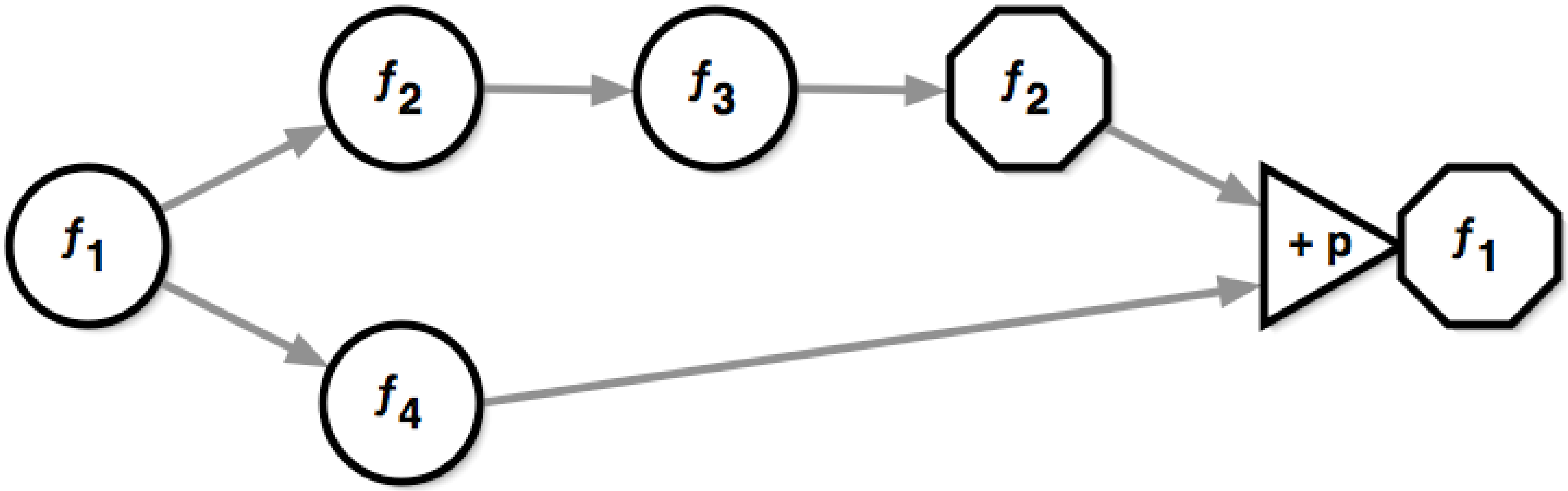
Constraint Representation Flow (almost)

```
T f1(T x, bool p) { return p ? f2(x) : f4(x); }
T f2(T x) { return f3(x + k); }
T f3(T x);
T f4(T x);
```



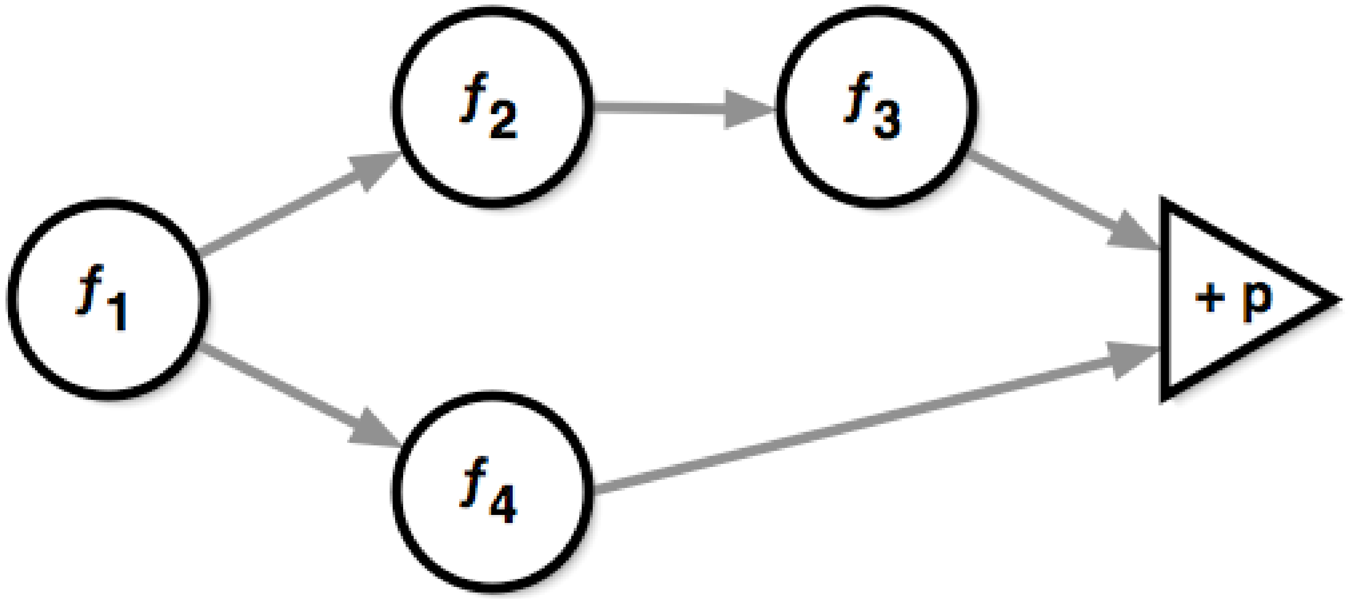
Constraint Representation Flow

```
T f1(T x, bool p) { return p ? f2(x) : f4(x); }
T f2(T x) { return f3(x + k); }
T f3(T x);
T f4(T x);
```



Constraint Representation Flow

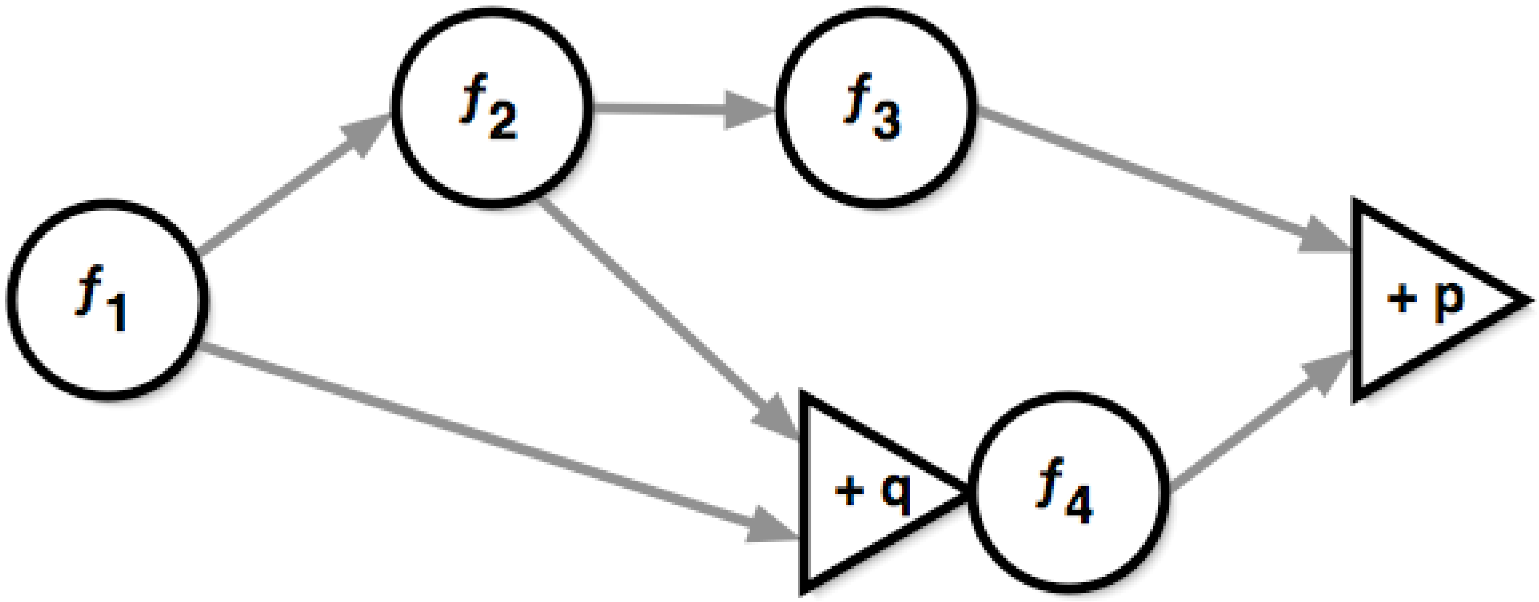
```
T f1(T x, bool p) { return p ? f2(x) : f4(x); }
T f2(T x) { return f3(x + k); }
T f3(T x);
T f4(T x);
```



Observations

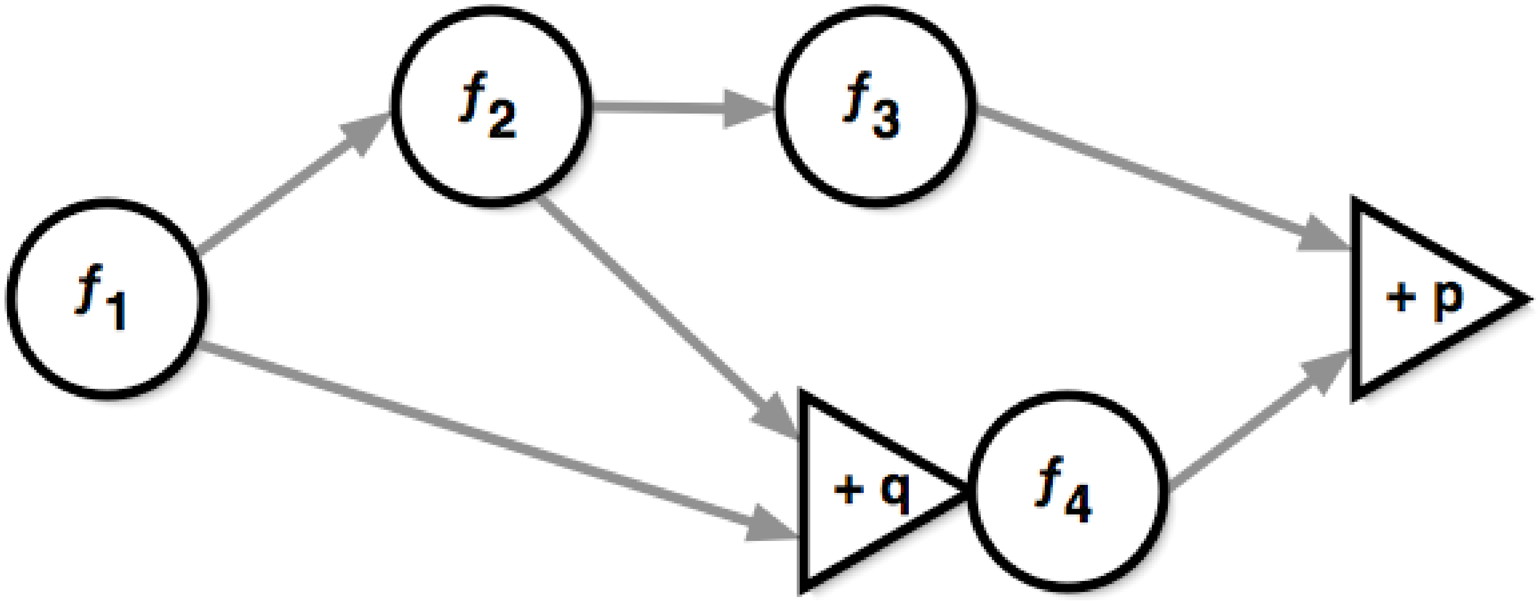
- **"Tautological Join"**
  - A constraint which cannot yield a contradiction
  - Interesting tautological join functions: max, ordering, queues
- **A functional program can be described as a (potentially) infinite directed acyclic constraint system with tautological joins.**
  - A cycle in the system is the equivalent of an infinite graph
- **A *finite* directed acyclic constraint system with tautological joins, "tautoldag", is solvable**
  - This is not Turing complete (guaranteed to halt)
- **There are tautoldags which cannot be simply mapped to a functional program**

Cross Hierarchy Joins

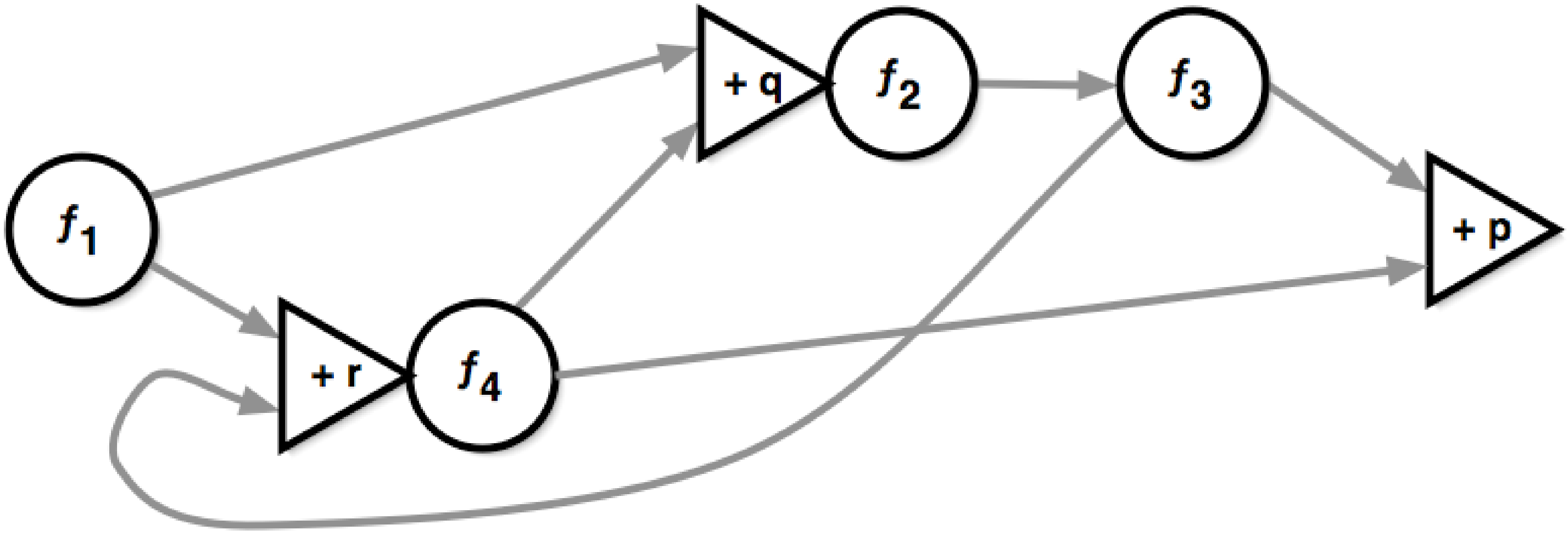


One Possible Representation

```
T f1(T x, bool p, bool q) {
    return p ? f2(x, f3) : (q ? f2(x, f4) : f4(x));
}
T f2 (T x, F f) {
    return f(x + k);
}
```



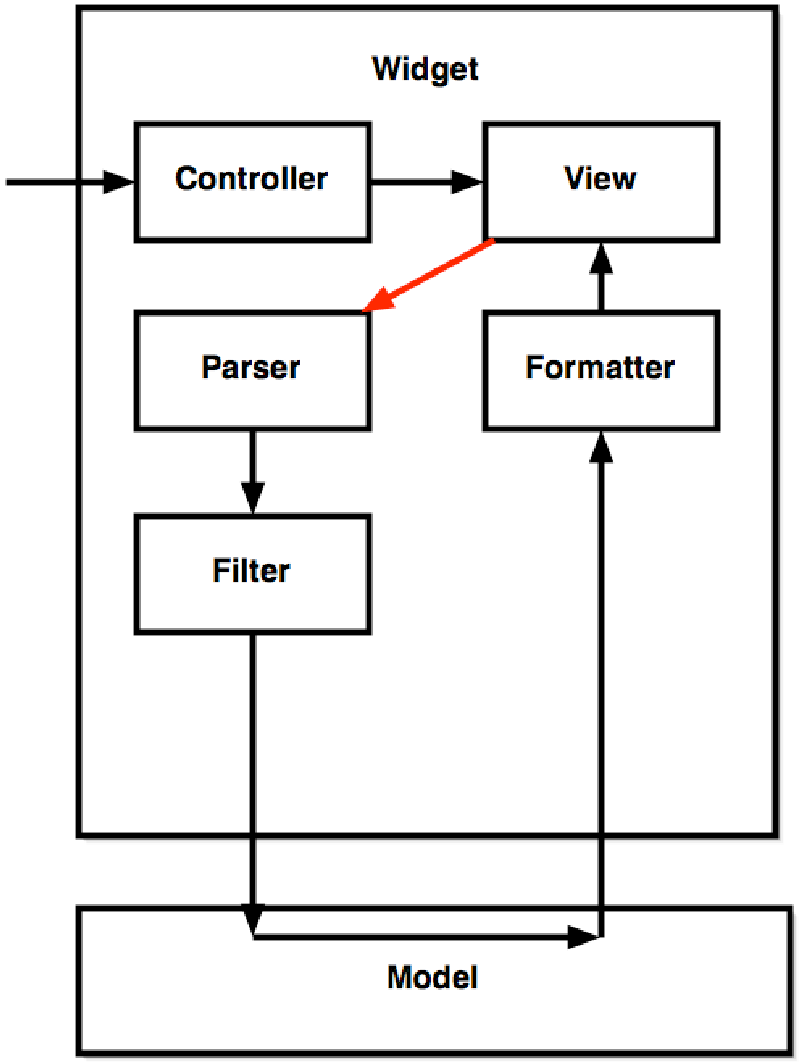
Beware of Cross Hierarchy Joins

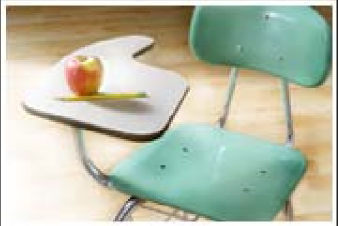


Observations

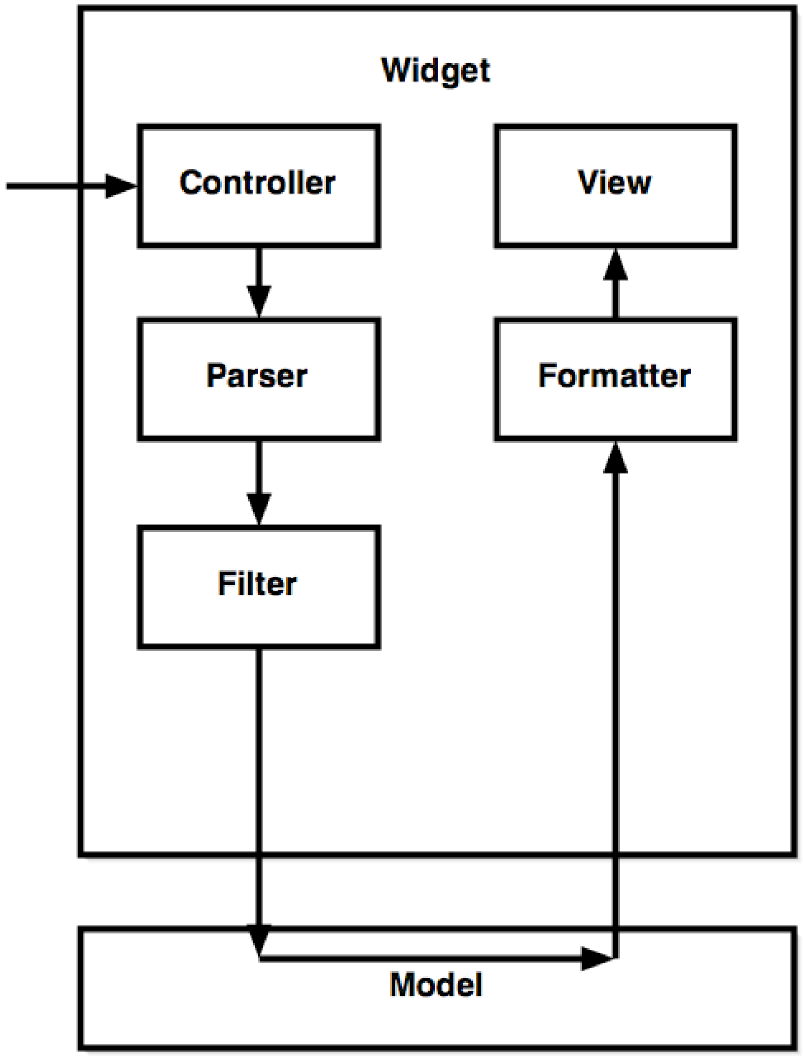
- **Although I may have named tautoldags - they are not my discovery:**
  - Functional Programming *works* because of tautoldags
  - Unix pipes are tautoldags - the queue structure provides enough “elasticity” to avoid contradictions (deadlocks).
  - Implicit hierarchies in object oriented programming *work* because they are tautoldags
- **Architectural failures are often rooted in feedback loops and contradictory, or underspecified joins**
- **Great care must be taken when feedback is required, through state or logic, to isolate the effects**

Typical Widget Implementation

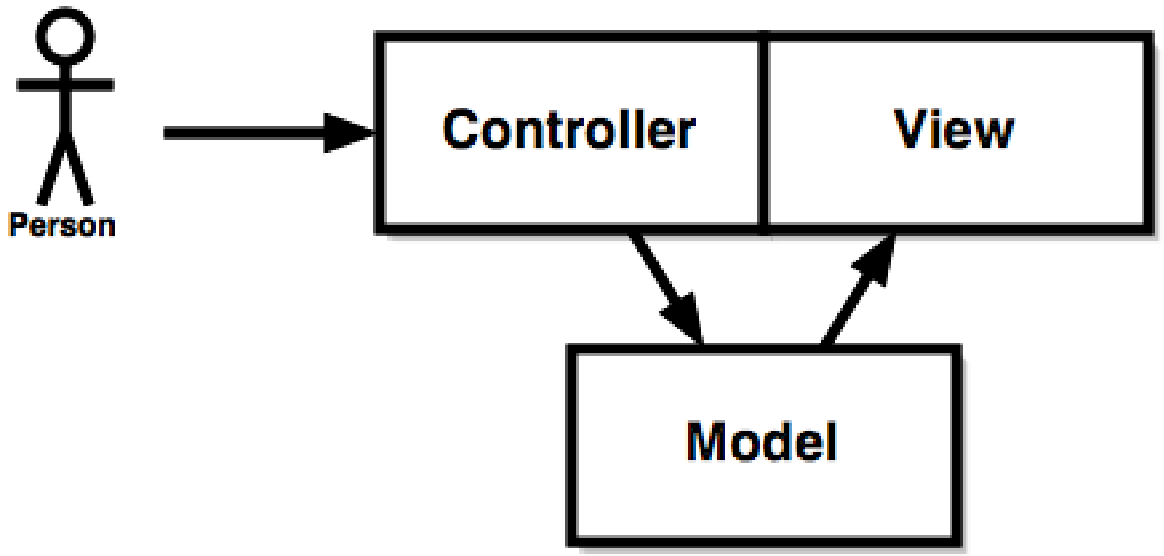




Reasonable Widget Implementation



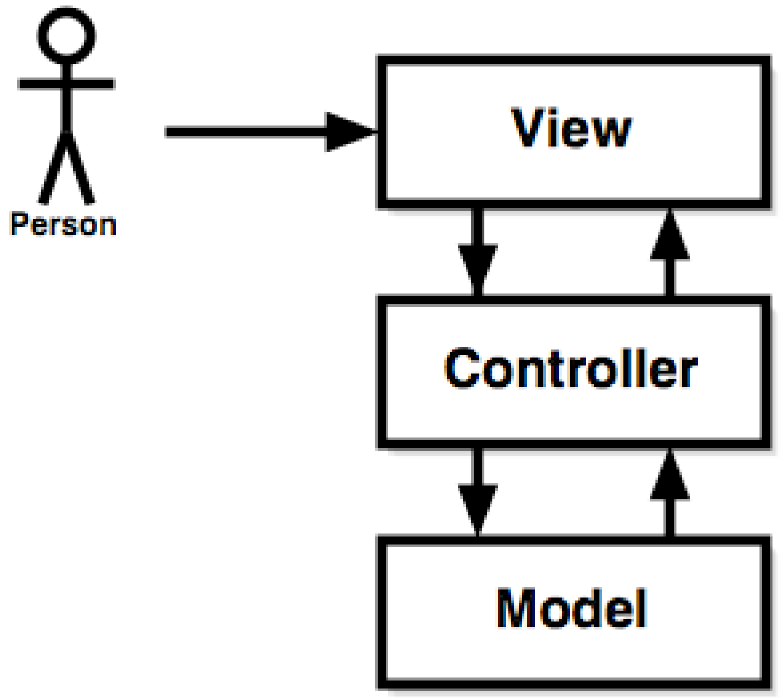
We’ve Discovered MVC!



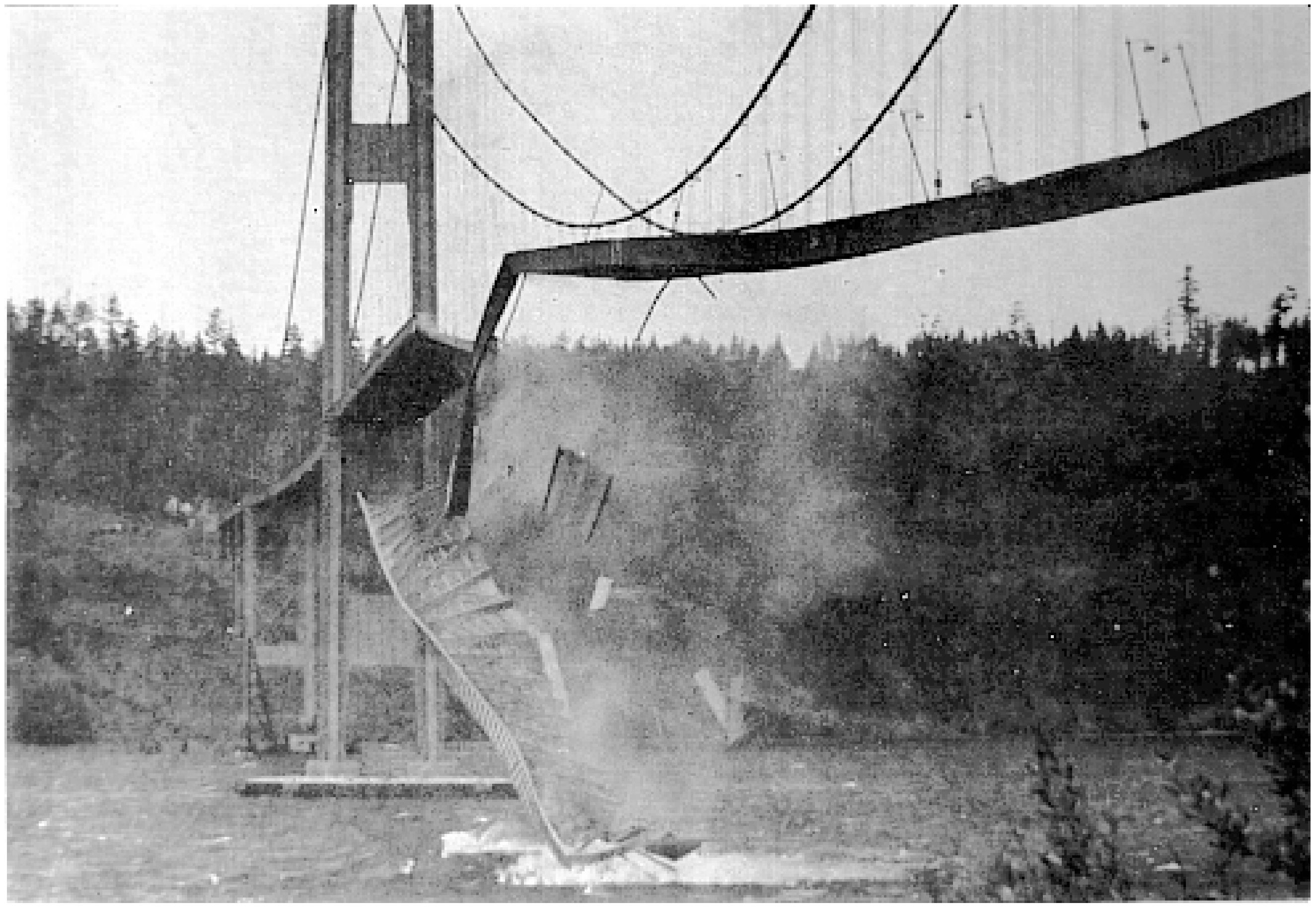
“A view object knows how to display **and possibly edit** data from the application’s model… A controller object **acts as the intermediary between the application’s view objects and its model objects**… Controllers are often the **least reusable objects** in an application, but that’s acceptable…”

— The Model-View-Controller Design Pattern according to developer.apple.com

Ummm…



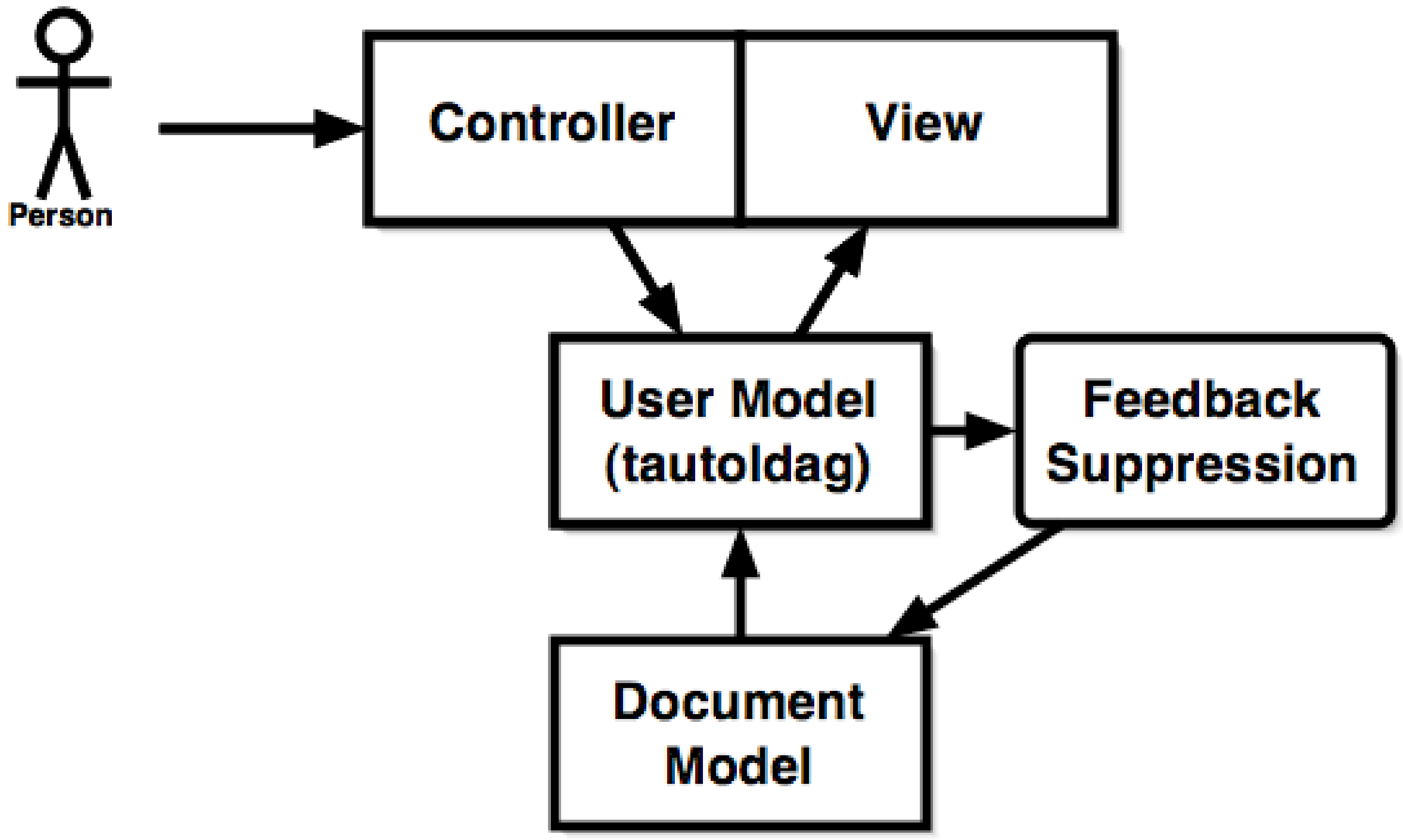
Ummm…



Adobe®

Tools for the New Work™

Working Architecture



Feedback Suppression

- **Using an “OK” button or other control to latch command**
  - Action is repeated each time button is pressed
- **Resetting the user model to a no-op state**

Adobe®

Tools for the New Work™