Software Architecture (Part 3)

Michael Coblenz

Today's Example: Key Word In Context

- Why?
  - You already read about the system
  - But let's examine the tradeoffs more closely
  - And we'll see how diagrams relate to code
- Note: examples and images are from Shaw and Garlan, "Software Architecture: Perspectives on an Emerging Discipline."

Approach #1: Subroutines (Functions)

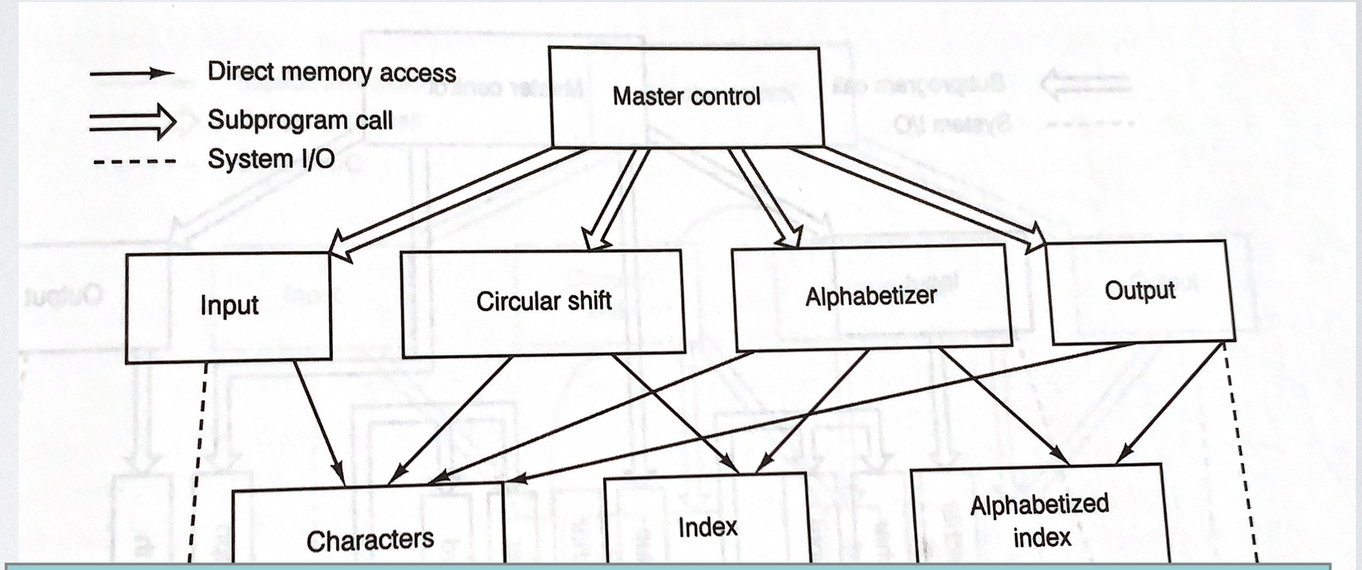
```
void kwic() {
  char *storage = …;
  Index *index = …;

  input(storage);

  // put shifts in index
  circularShift(storage, index);

  // sort index alphabetically
  alphabetize(storage, index);

  output(storage, index);
}
```



Considerations:

- A change to storage requires changes *everywhere*
- Changing overall algorithm requires rewriting kwic() function
- Can't easily reuse any components

Approach #2: Abstract Data Types

- Idea: hide representations behind abstractions to make modification easier

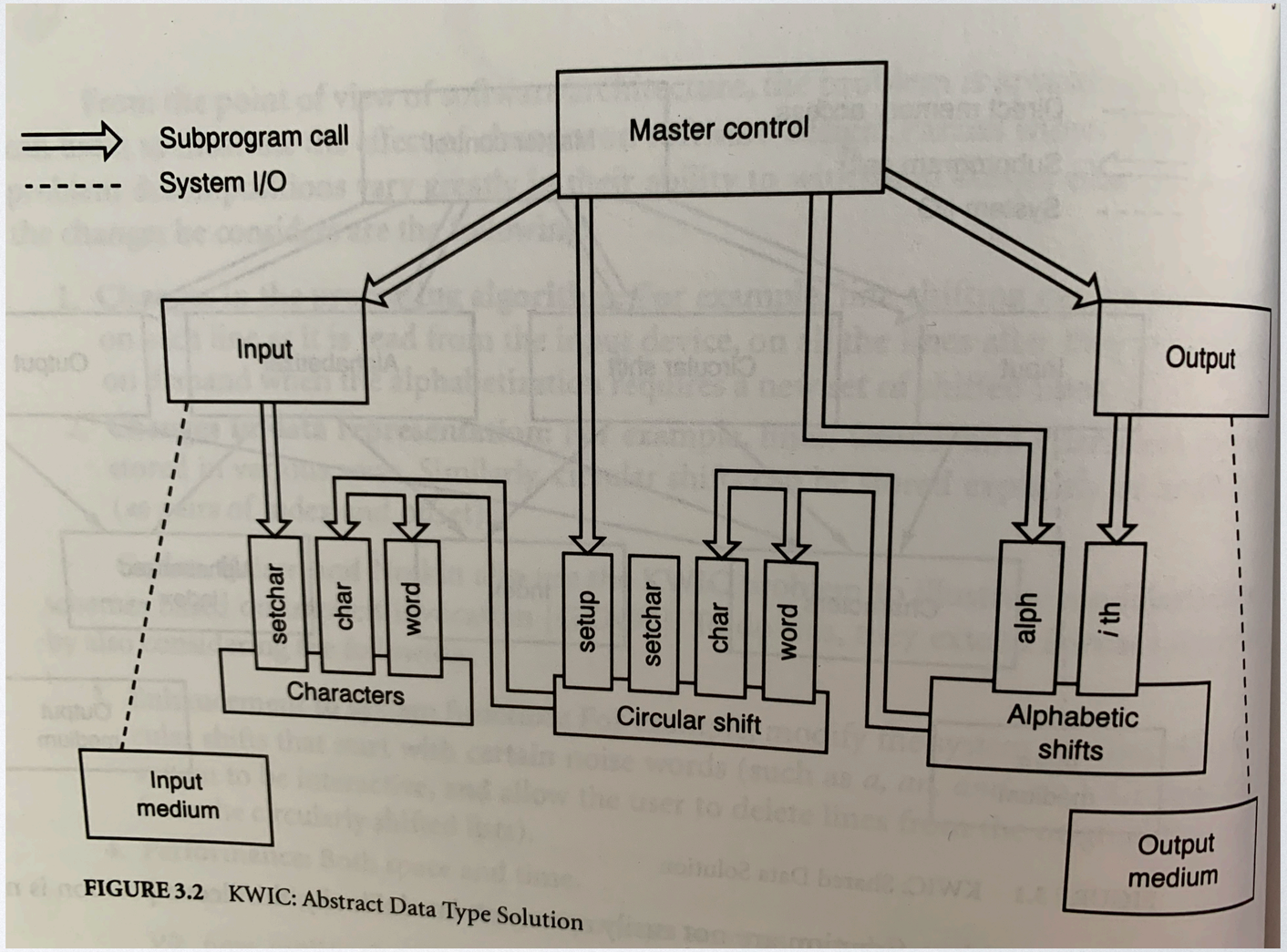


FIGURE 3.2 KWIC: Abstract Data Type Solution

Approach #2: Abstract Data Types

```
void kwic() {
  CircularShift *shift = …;

  Characters *storage = input();

  shift->setup(storage);
  Shifts *shifts =
    new Shifts(shift);

  shifts.alph();

  output(shifts);
}
```

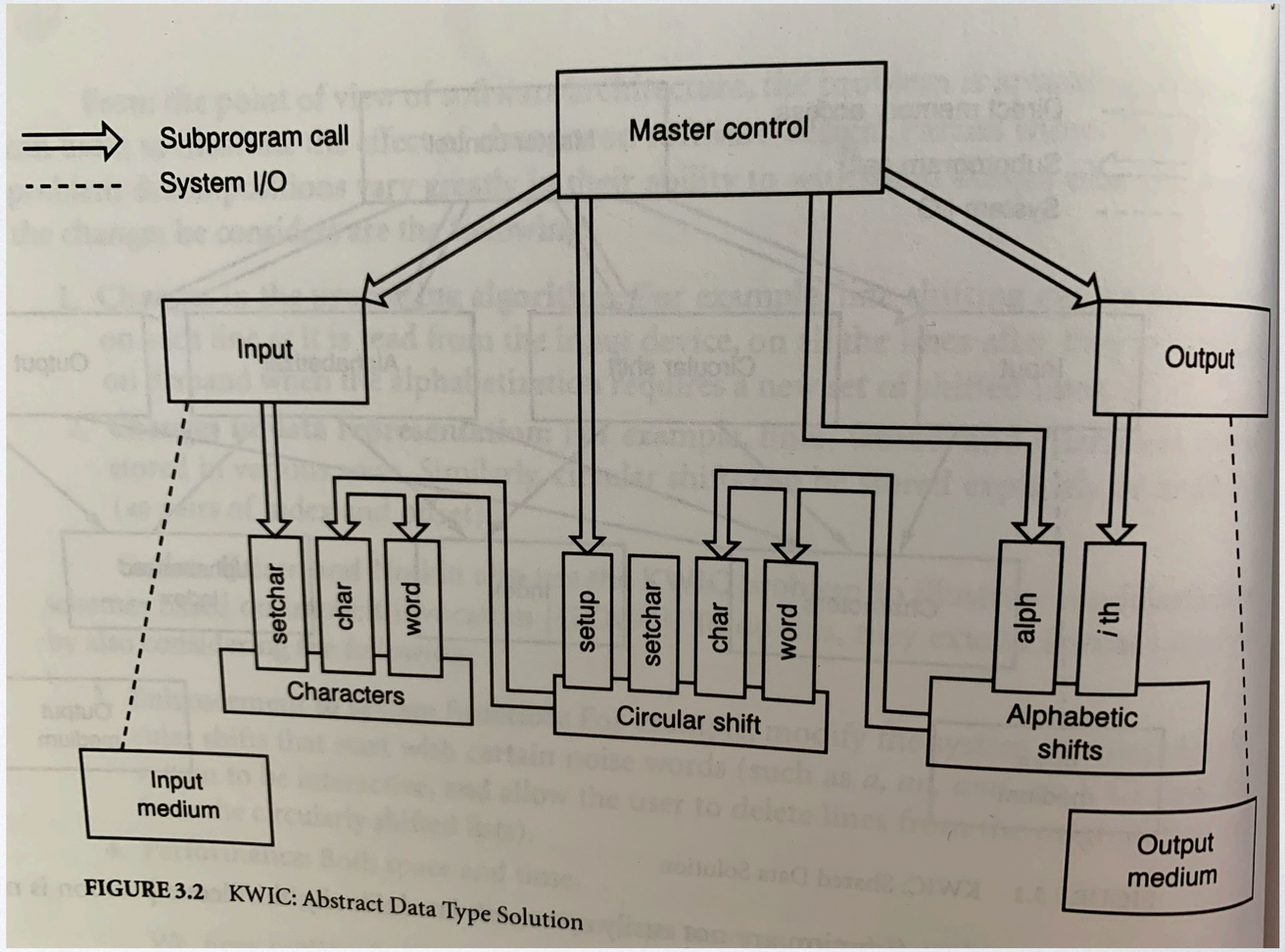


FIGURE 3.2 KWIC: Abstract Data Type Solution

Approach #3: Implicit Invocation

```
void kwic() {
  Lines l = new Lines();
  CircularShift shift = new CircularShift();
  eventBus.subscribe("LineInserted", shift.lineInserted);
  input(lines);
  output();
}

void input(Lines lines) {
  while (line = getLine()) {
    lines.insert(line);
}

class Lines {
  void insert(String line) {
    int index = …;
    eventBus.notify("LineInserted", index);
  }
}
```

really this goes elsewhere

```
class CircularShift {
  void lineInserted(i) {
    String line = inputLines.ith(i);
    alphabetizerLines.insert(line);
    eventBus.shiftLineInserted();
  }
}
```

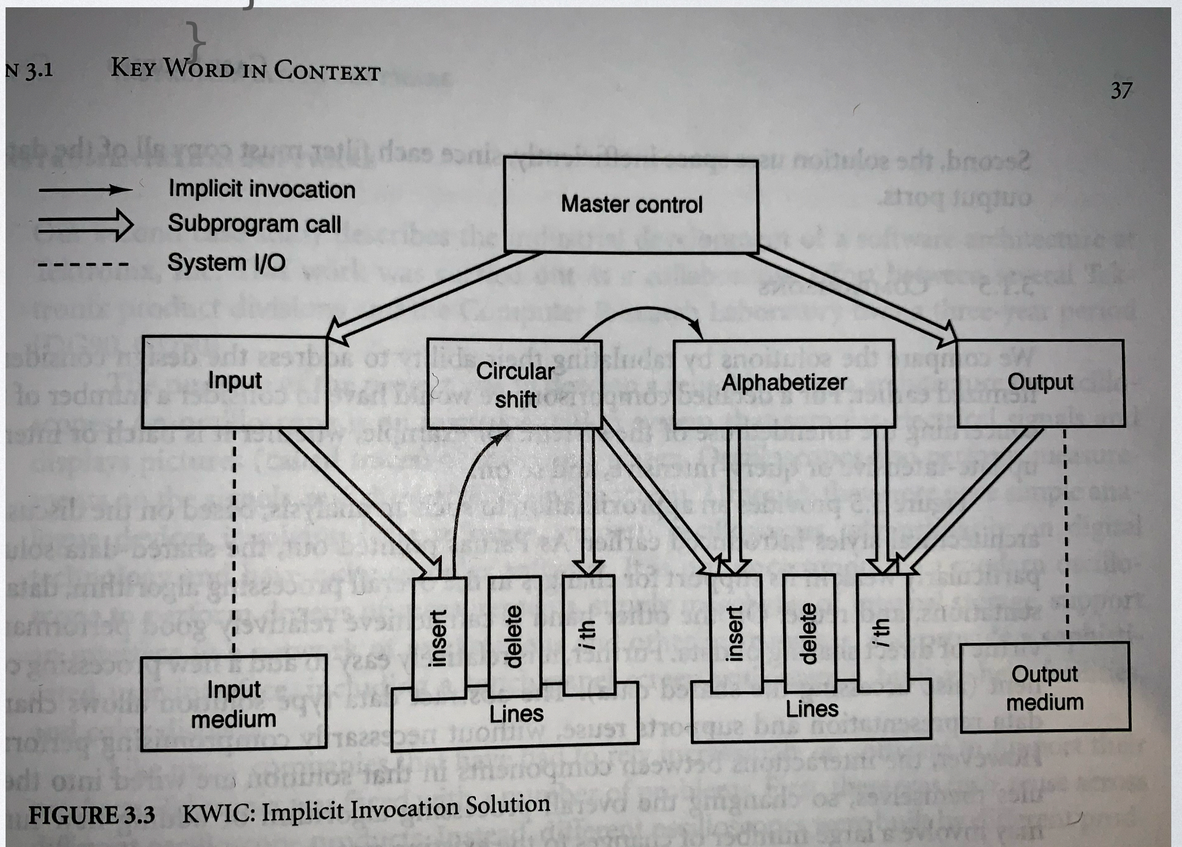
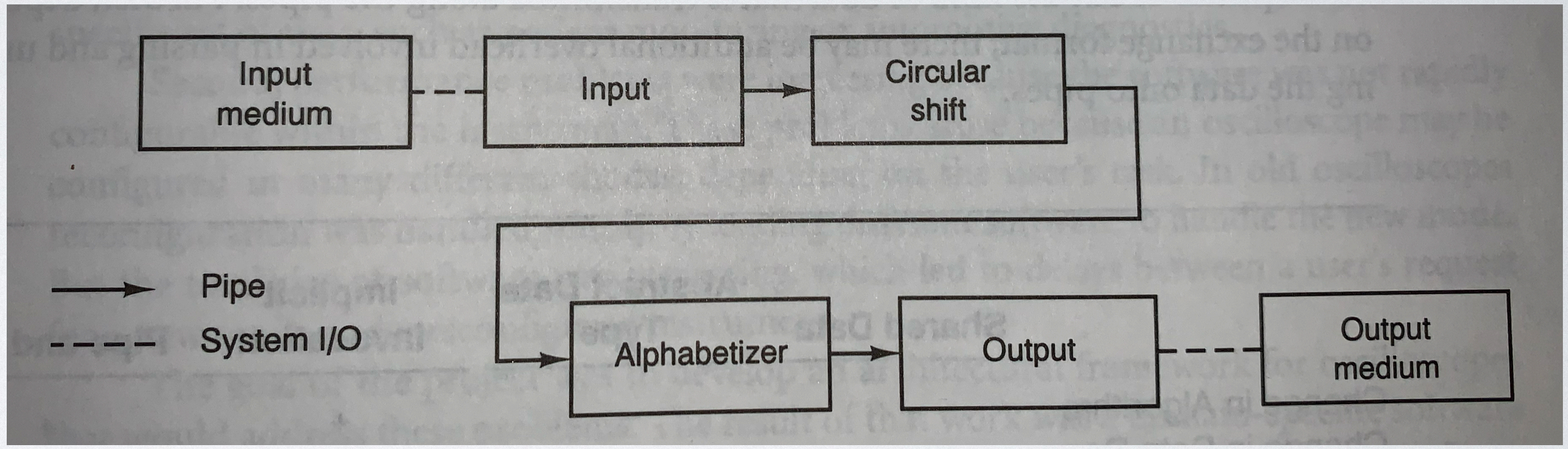


FIGURE 3.3 KWIC: Implicit Invocation Solution

Approach #4: Pipes and Filters

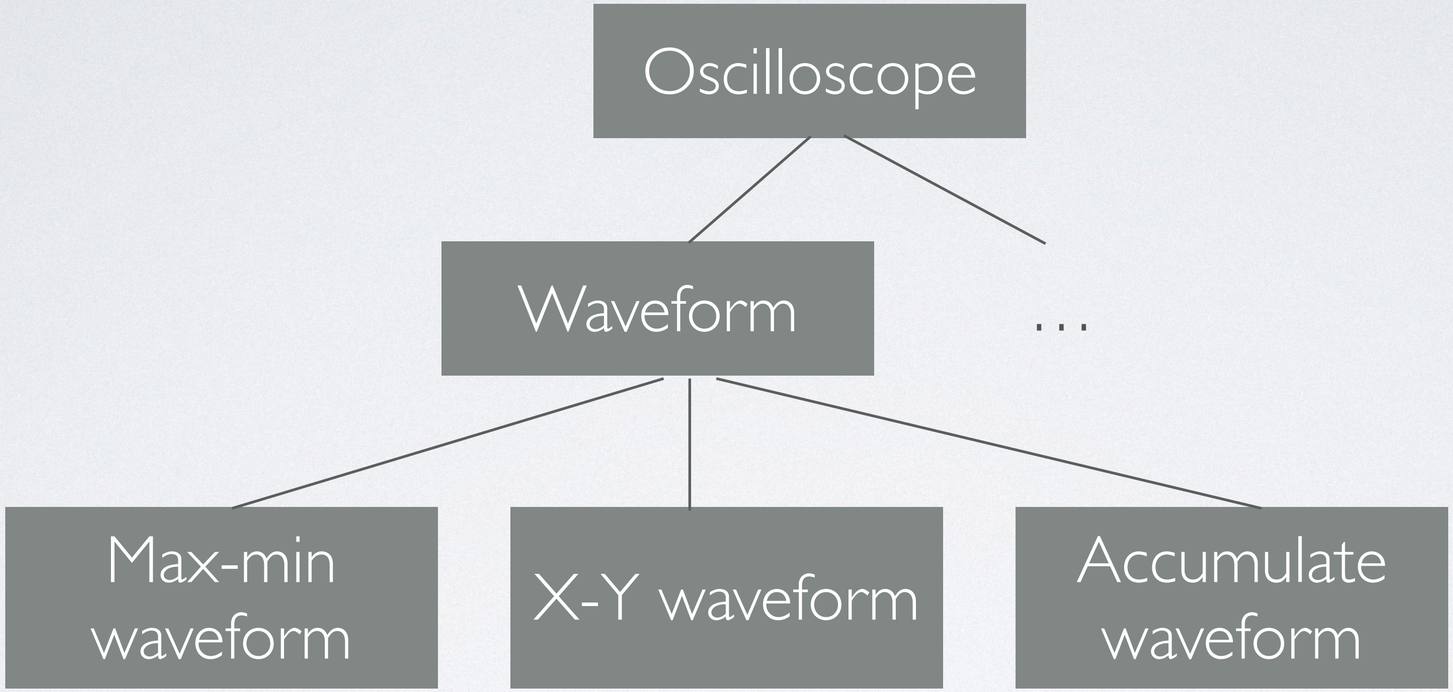


- +
  - Filters are isolated
  - Functions easily added or removed
- -
  - Can't support interactive system (e.g. deleting a line)
  - Inefficient space usage

Example 2: Oscilloscope

- Context: fancy oscilloscope (Tektronix, Inc.)
- Problem 1: want to reuse software across products (different hardware, different user interfaces)
- Problem 2: software not configurable in different modes for different tasks

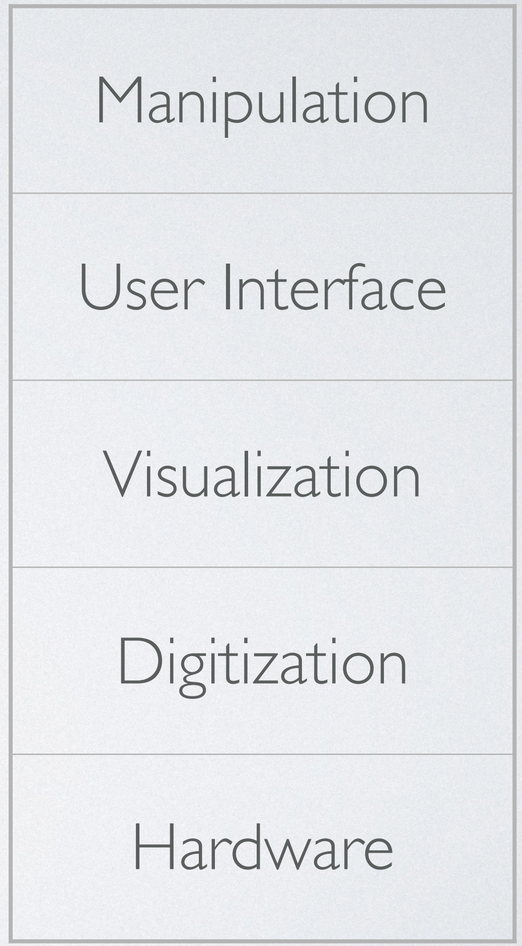
Approach 1: Object-Oriented



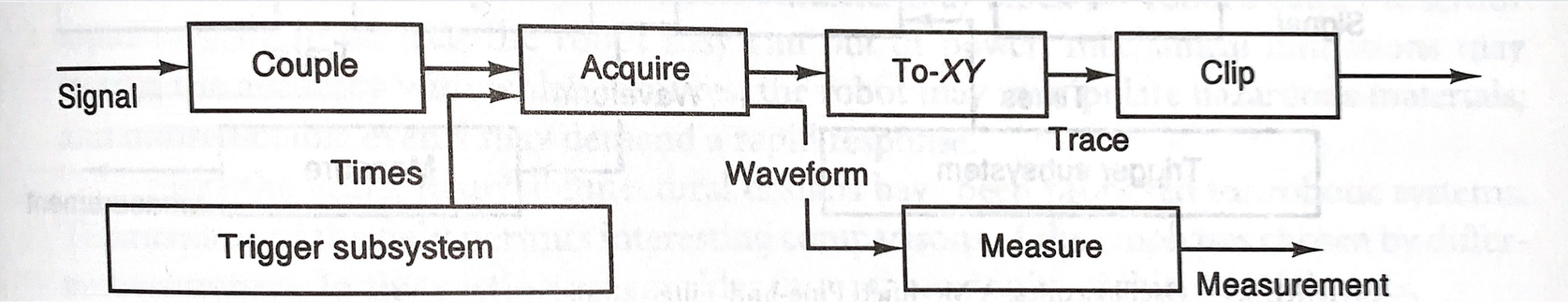
- How should functionality be partitioned?
- Should measurements be associated with the data being measured, or have their own representation?
- Which objects should the UI interact with?

Approach 2: Layers

- Digitization: waveform acquisition
- Visualization: waveform manipulation
- But abstractions conflict with interactions among functions
  - User interactions aren't always in terms of visual representations
  - User may need to set attenuation in the digitization layer
- If there are too many tunnels needed, maybe you have the wrong architecture.

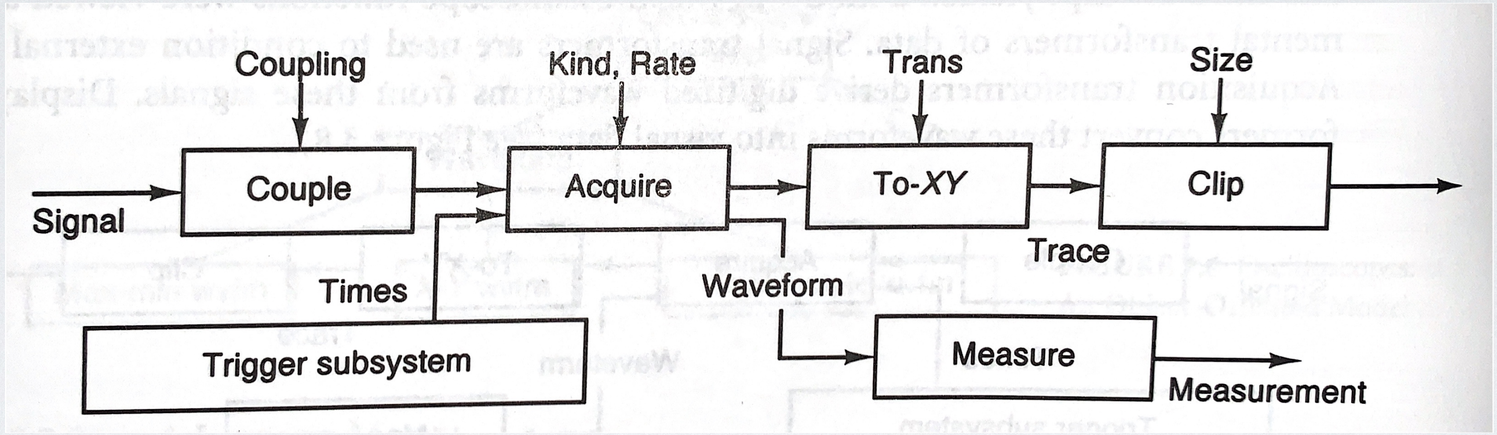


Approach 3: Pipes and Filters



- Avoids isolating functions in separate partitions
  - Could feed signal directly to display filters if needed
- But how should the user interact with it?

Approach 4: Modified Pipe and Filter



- Approach: add control inputs to each filter.
- Separates analysis from actual user interface (not shown).
- But this caused performance problems: too much copying along pipes!
- Solution: several kinds of pipes: no-copy, ignore-incoming-data-while-busy

Your Turn

- Design an architecture for an elevator.
- Functional requirements: comes when called, stops at floors.
- Non-functional requirements:
  - Modifiability. Need to support re-labeling floors. May want to play ads according to the current floor.