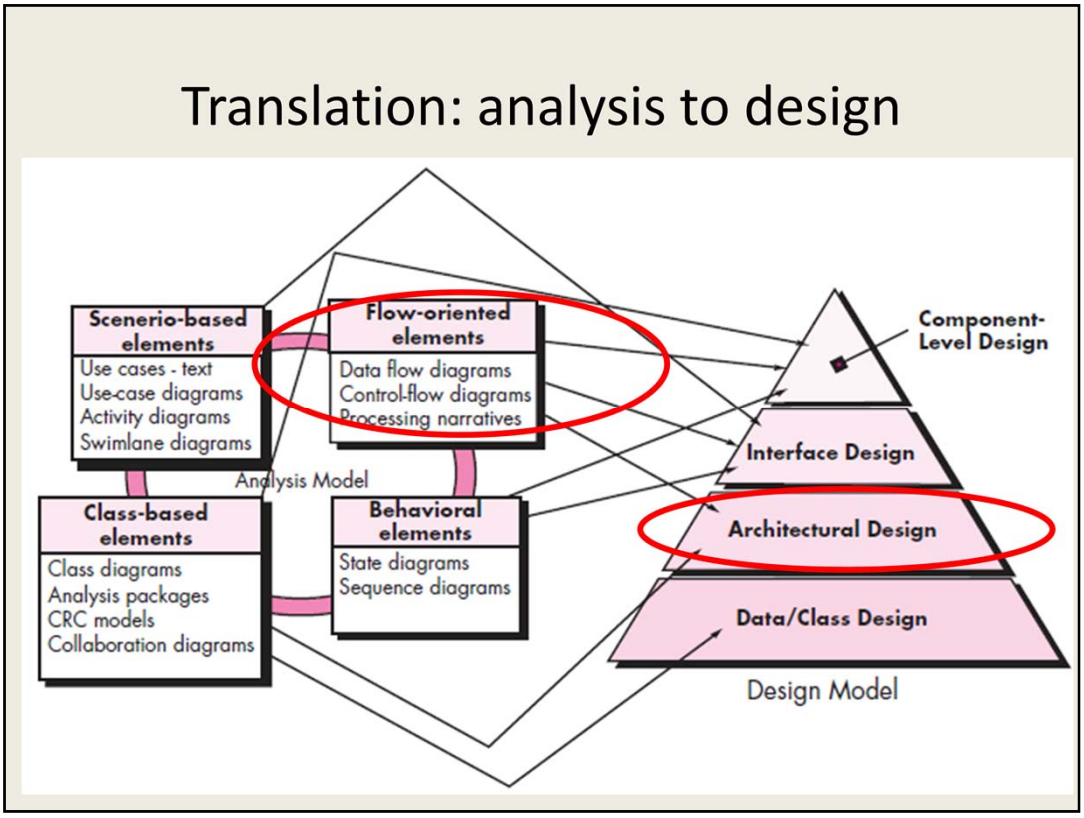
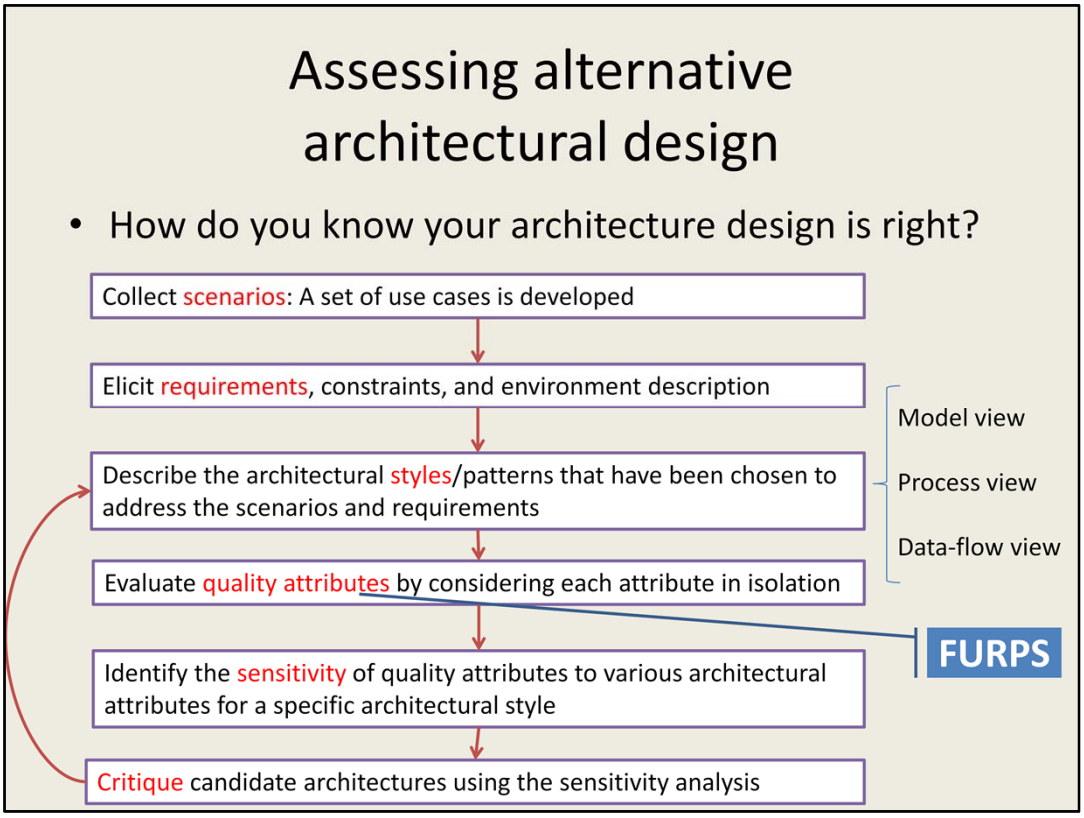
Architectural Design
(II)



Now you see why we need different elements in requirements modeling?
Each of the elements of the requirements model (Chapters 6 and 7) provides information that is necessary to create the four design models required for a complete specification of design.

Data/Class design: transforms analysis classes into implementation classes and data structures
Architectural design: defines relationships among the major structural elements of the software, the architectural ***styles and patterns***
Interface design–defines how software elements, hardware elements, and end-users communicate. Usage scenarios and behavioral models are used
Component-level design–transforms structural elements into procedural descriptions of software components. Class-based models and behavioral models server as the basis

Only scenario-based elements serve just one design model



Design results in a number of architectural alternatives that are each assessed to determine which is the most appropriate for the problem to be solved

•*Module view* for analysis of work assignments with components and the degree to which information hiding has been achieved.

•*Process view* for analysis of system performance.

•*Data flow view* for analysis of the degree to which the architecture meets functional requirements

FURPS - Quality attributes of a design : Functionality Usability Reliability Performance Supportability

Sensitivity analysis

- Make small changes to architecture design and observe the deviation of

quality attributes.

- Changes cause big variation on quality attributes are sensitivity points.

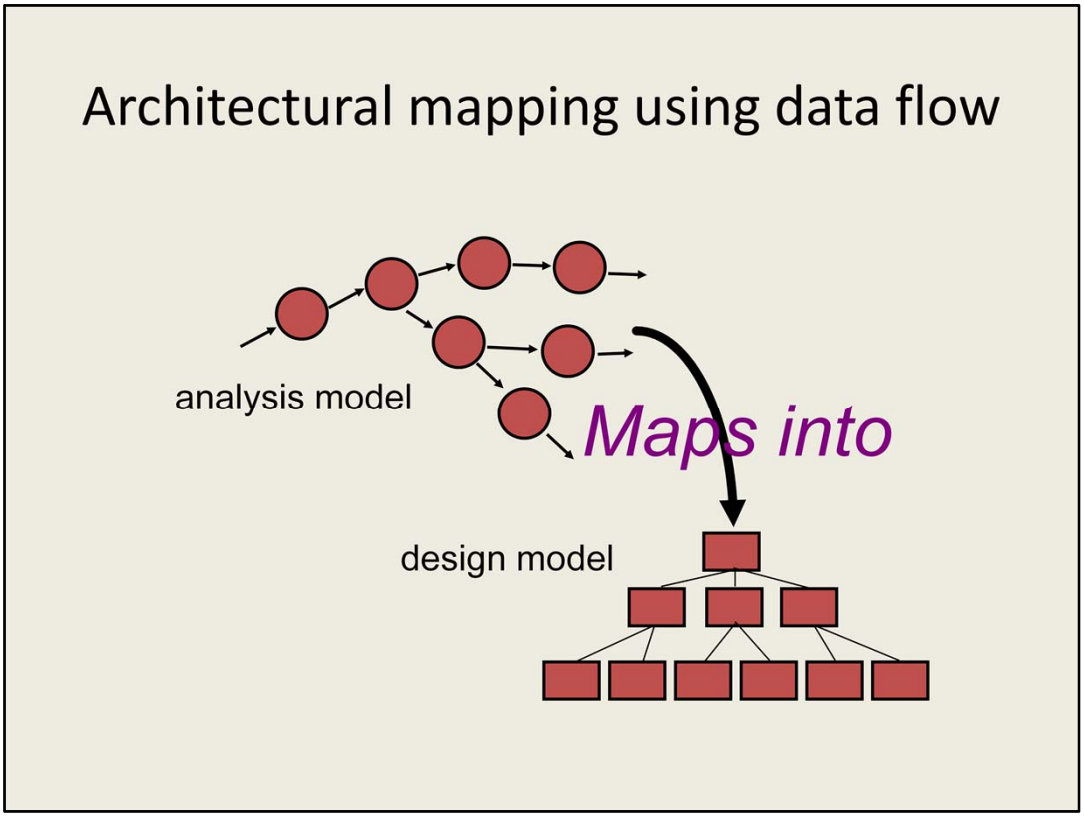
Assessing alternative architectural design

- Architectural Complexity
  - Sharing / flow/ constrained dependencies
- Architecture description language (ADL)
  - semantics and syntax for describing a software architecture

A useful technique for assessing the overall complexity of a proposed architecture is

to consider dependencies between components within the architecture

Although the software architect can draw on UML notation, other diagrammatic forms, and a few related tools, there is a need for a more formal approach to the specification of an architectural design.



The architectural styles discussed in Section 9.3.1 represent radically different architectures.

It should come as no surprise that a comprehensive mapping that accomplishes the transition from the requirements model to a variety of architectural styles **does not exist**

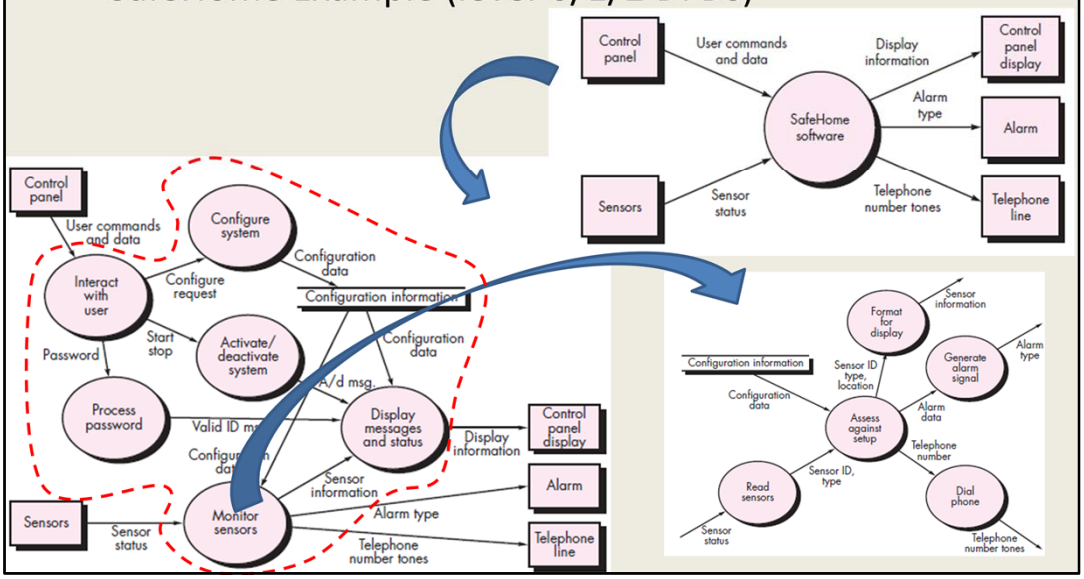
Architectural mapping using data flow

- Data-flow oriented design
  - DFD -> software architecture: data-flow mapping
    - **Step 1. Review the fundamental system model**
      - Context DFD (level 0 DFD)
    - **Step 2. Review and refine data flow diagrams for the software**
      - Detailed levels of DFD

A mapping technique, called *structured design,* is often characterized as a data flow-oriented design method because it provides a convenient transition from a data flow diagram (Chapter 7) to software architecture

Architectural mapping using data flow

- SafeHome Example (level-0/1/2 DFDs)

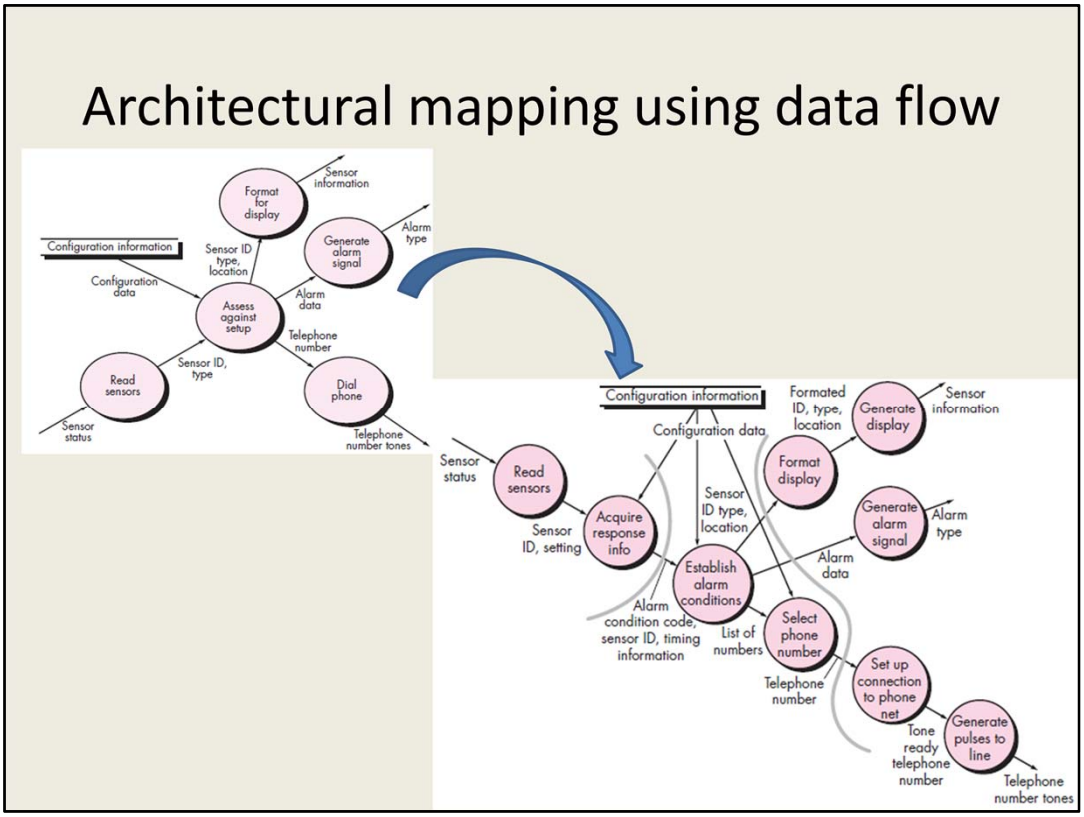


Each bubble is refined until it does just one thing

The expansion ratio decreases as the number of levels increase

Most systems require between 3 and 7 levels for an adequate flow model

A single data flow item may be expanded as levels increase.



the level 2 DFD for *monitor sensors* (Figure 9.12) is examined, and a level 3 data flow diagram is derived as shown in Figure 9.13. At level 3, each transform in the data flow diagram exhibits relatively high cohesion;

That is, the process implied by a transform performs *a single, distinct function that can be implemented as a component* in the *SafeHome* software

Architectural mapping using data flow

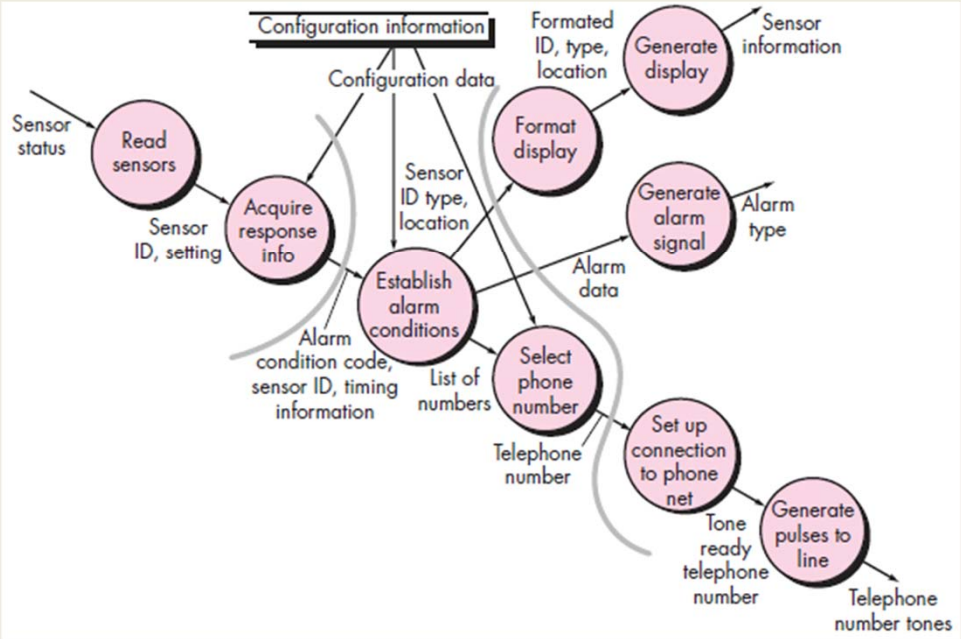
- DFD -> software architecture: data-flow mapping
  - **Step 3. Determine whether the DFD has transform or transaction flowcharacteristics**
    - Transform flow: linear / sequential
    - Transaction flow: branching at transaction center
  - **Step 4. Isolate the transform center by specifying incoming and outgoing flow boundaries.**

One type of information flow is called *transform flow* and exhibits a linear quality. Data flows into the system along an *incoming flow path* where it is transformed from an external world representation into internalized form. Once it has been internalized, it is processed at a *transform center.* Finally, it flows out of the system along an *outgoing flow path* that transforms the data into external world form

In transaction flow, a single data item, called a *transaction,* causes the data flow to branch along one of a number of flow paths defined by the nature of the transaction.

Some DFDs may include both types of information flow.

Architectural mapping using data flow



we see data entering the software along one incoming path and exiting along three outgoing paths

Incoming data flows along a path in which information is converted from external to internal form; outgoing flow converts internalized data

to external form.

different designers may select slightly different points in the flow as boundary locations. In fact, alternative design solutions can be derived by varying the placement of flow boundaries

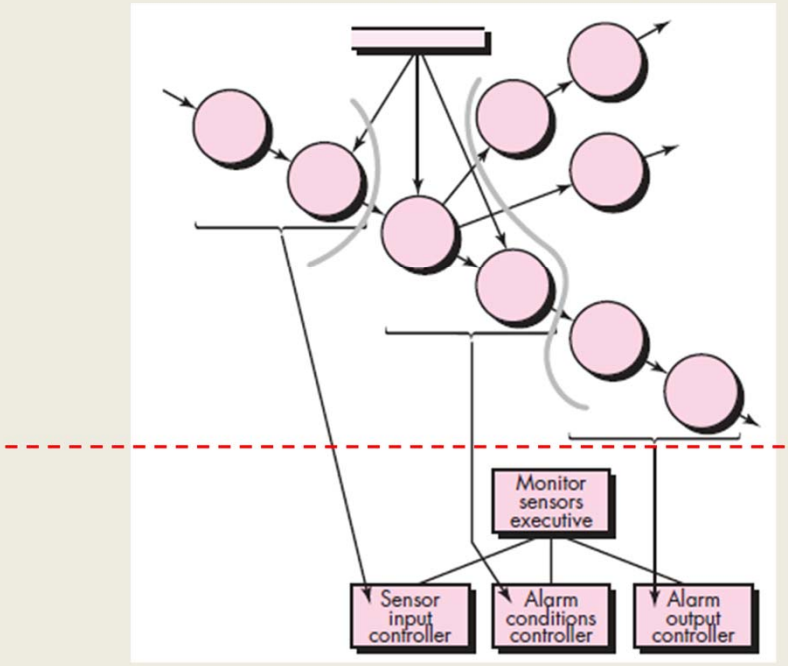
Architectural mapping using data flow

- DFD -> software architecture: data-flow mapping
  - **Step 5. Perform "first-level factoring."**
    - Top level: decision making
    - Low level: perform input, computation, and output
    - Middle level: control and moderate

Factoring leads to a program structure in which **top-level** components perform decision making and **lowlevel** components perform most input, computation, and output work. **Middle-level** components perform some control and do moderate amounts of work.

When transform flow is encountered, a DFD is mapped to a specific structure (a call and return architecture)

Architectural mapping using data flow



An incoming information processing controller, called *sensor input controller,* coordinates receipt of all incoming data

A transform flow controller, called *alarm conditions controller,* supervises all operations on data in internalized form (e.g., a module that invokes various data transformation procedures).

An outgoing information processing controller, called *alarm output controller,* coordinates production of output information

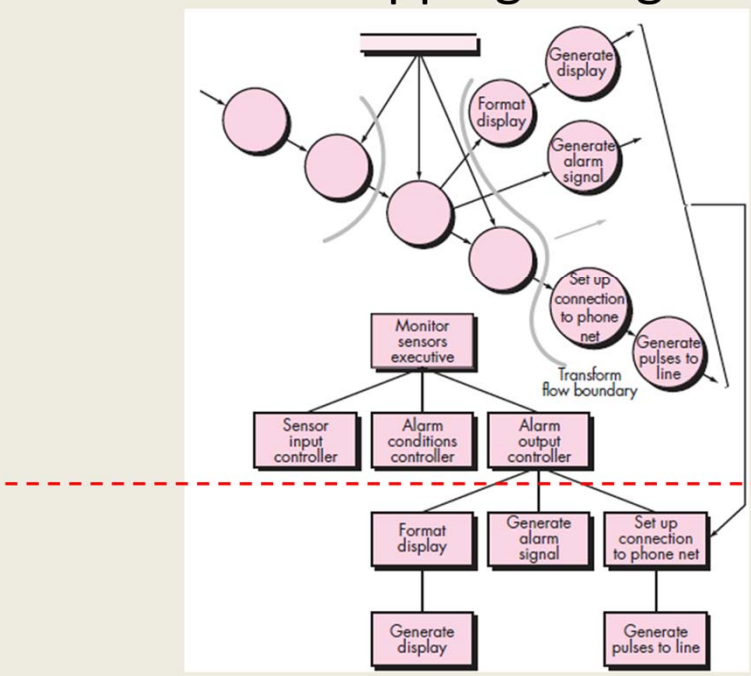
Architectural mapping using data flow

- DFD -> software architecture: data-flow mapping
  - **Step 6. Perform "second-level factoring."**
    - Mapping individual transforms into modules

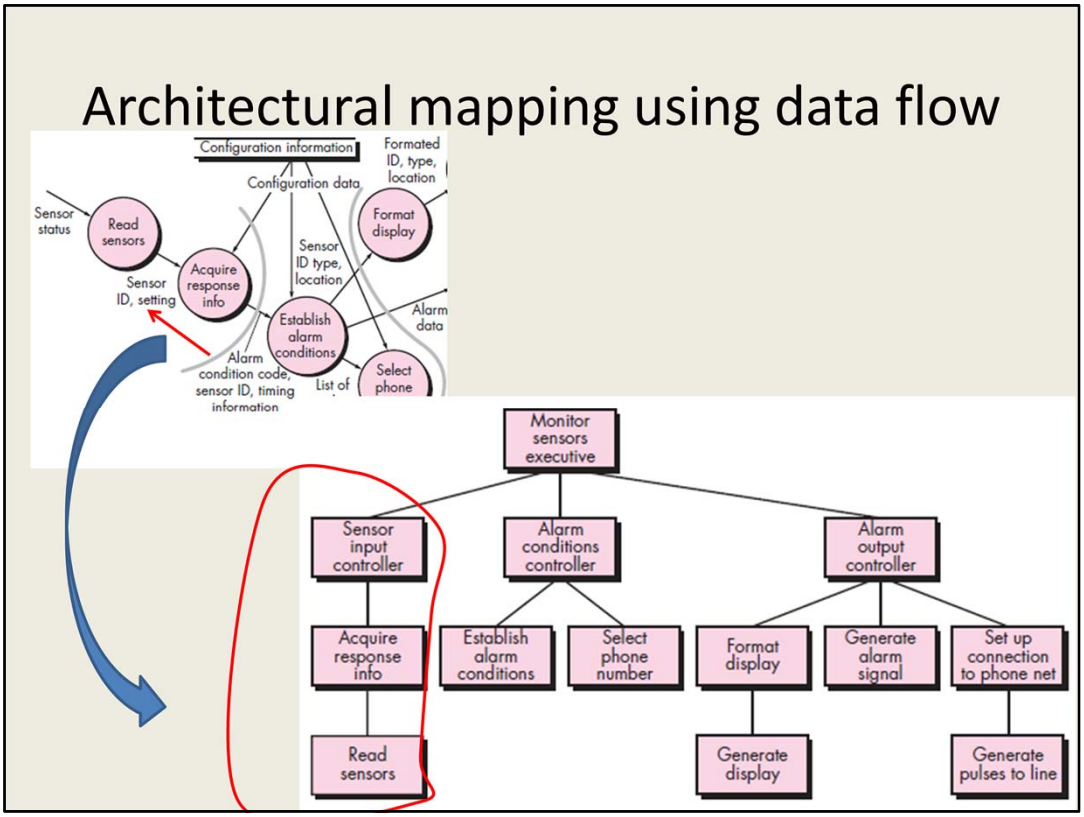
Second-level factoring is accomplished by mapping individual transforms (bubbles) of a DFD into appropriate modules within the architecture.

Beginning at the transform center boundary and moving outward along incoming and then outgoing paths, transforms are mapped into subordinate levels of the software structure

Architectural mapping using data flow



Although Figure 9.15 illustrates a one-to-one mapping between DFD transforms and software modules, different mappings frequently occur. Two or even three bubbles can be combined and represented as one component, or a single bubble may be expanded to two or more components.



Second-level factoring for incoming flow follows in the same manner. Factoring is again accomplished by moving outward from the transform center boundary on the incoming flow side.

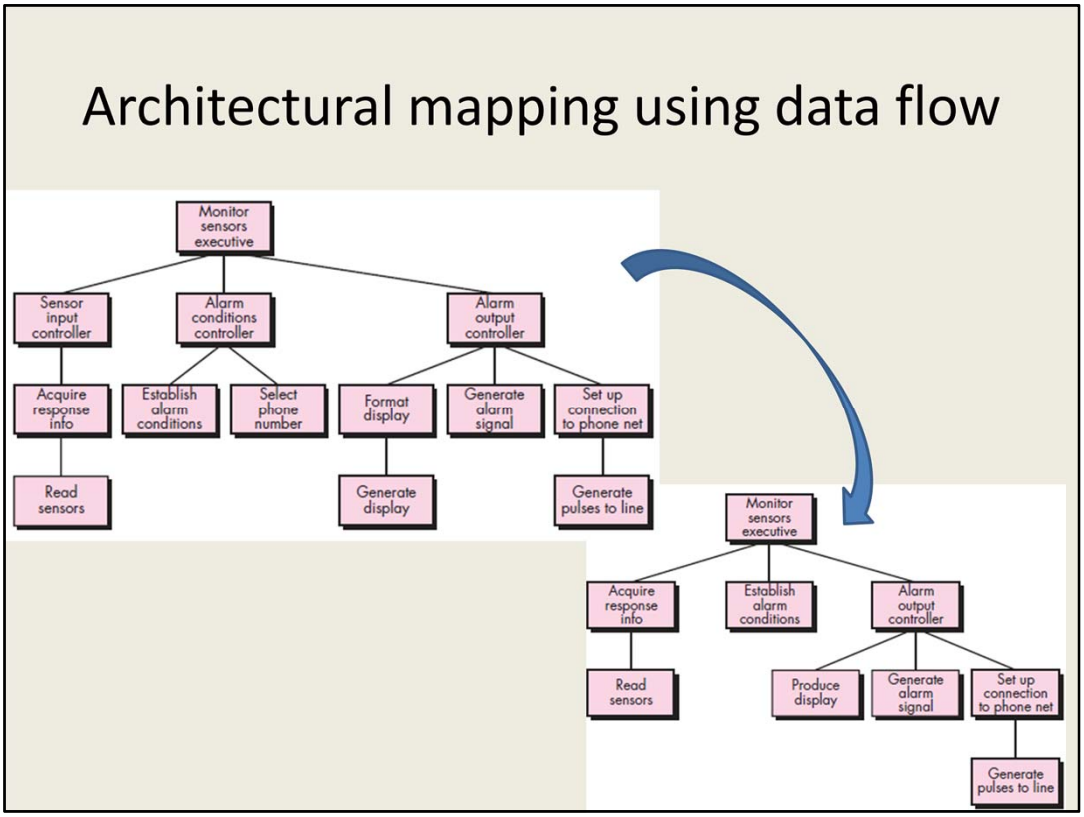
Architectural mapping using data flow

- DFD -> software architecture: data-flow mapping
  - **Step 7. Refine the first-iteration architecture using design heuristics for improved software quality.**
    - Mapping individual transforms into modules

A first-iteration architecture can always be refined by applying concepts of functional independence

Components are exploded or imploded to produce sensible factoring, separation of concerns, good cohesion, minimal coupling, and most important, a structure that can be implemented without difficulty, tested without confusion, and maintained without grief

Architectural mapping using data flow



- why did you use the *sensor input controller* component?

- Because you need a controller for the mapping

- Not really. The controller doesn't do much, since we're managing a single flow path for incoming data. We can eliminate the controller with no ill effects.

- We can also implode the components *establish alarm conditions* and *select phone number*. The transform controller you show isn't really necessary, and the small decrease in cohesion is tolerable

- And while we're making refinements, it would be a good idea to implode the components *format display* and *generate display*. Display formatting for the control panel is simple. We can define a new module called *produce display.*