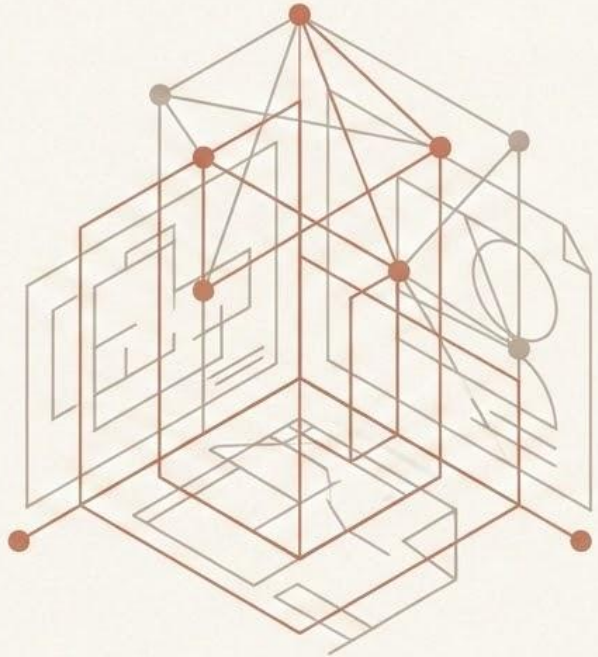
Architectural Design

Chapter 9 (Part 2): Refinement, Assessment, and Mapping

Subject: Software Engineering
Duration: 1 Hour



Architecture Refinement, Assessment, & Derivation

Moving from Concepts to Concrete Engineering

Focus: Filling the Boxes, Evaluating Quality, and Systematic Derivation

The Engineering Challenge

Opening Hook

The Current State: We have chosen an architectural style and identified the “big boxes.”

The New Questions:

- How do we fill those boxes with working components?
- How do we prove our architecture is actually good?
- Is there a systematic way to derive this structure directly from requirements?

The Goal: Today, we move from abstract concepts to concrete techniques for refinement, assessment, and derivation.

Learning Objectives

By the end of this lecture, you will be able to:

- **Refine Architecture:** Transform high-level styles into concrete components and describe system instantiations.
- **Evaluate Design:** Apply the Architecture Trade-off Analysis Method (ATAM) to evaluate design alternatives.
- **Define Structure:** Understand the purpose and utility of Architectural Description Languages (ADLs).
- **Derive Structure:** Perform Transform Mapping to scientifically derive software architecture from Data Flow Diagrams (DFDs).

Part 1: Refining the Architecture

Section 9.4.3 & 9.4.4

Focus: Moving from High-Level Styles to Concrete Implementations

Refining Architecture into Components

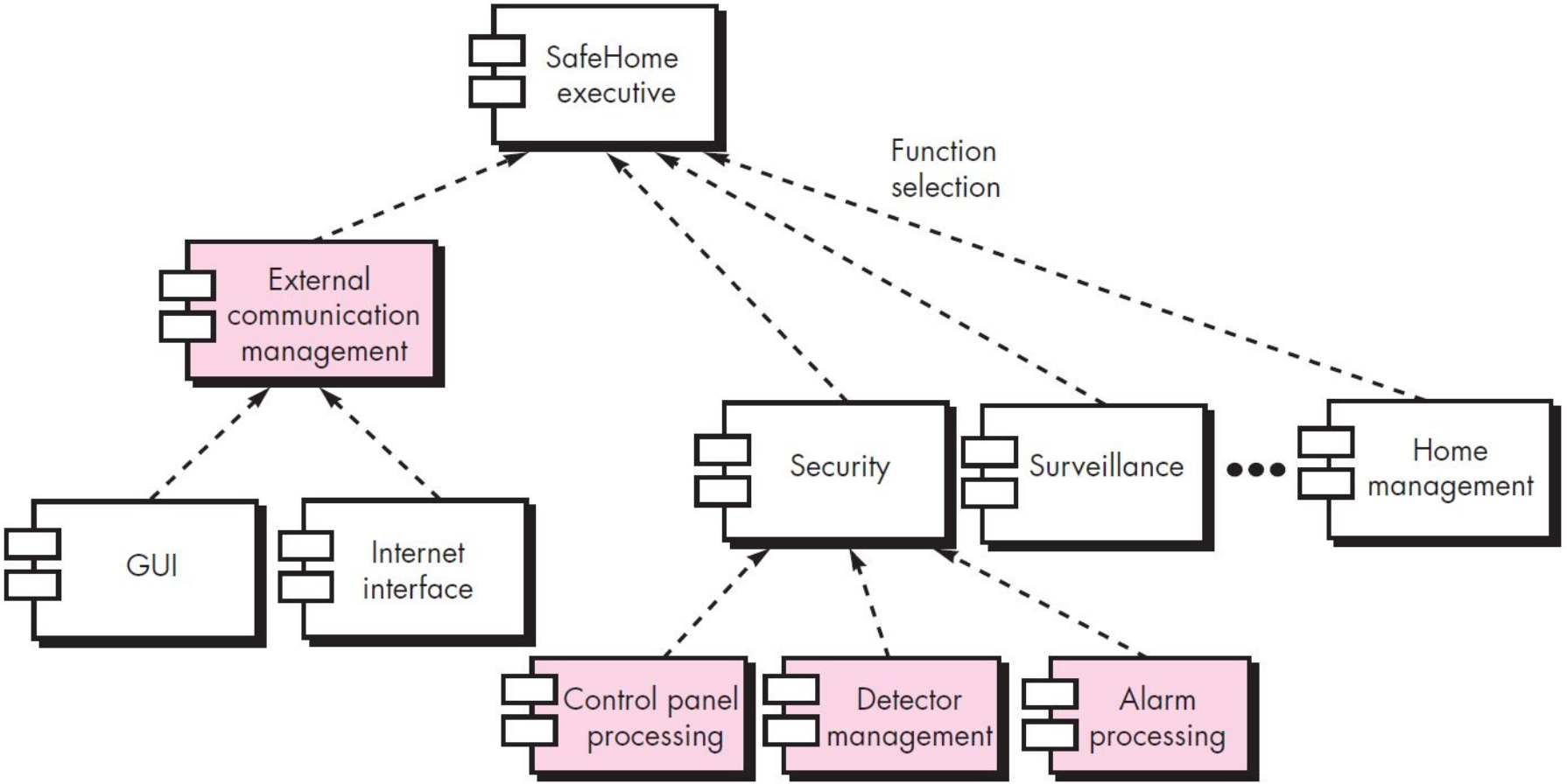
Section 9.4.3: The Decomposition Process

The Goal: Transition from abstract styles (e.g., "Layered," "MVC") to specific, named components with clear responsibilities.

The Process:

- **Decompose Subsystems:** Break down each major element (layer, tier) into finer-grained modules or packages.
- **Assign Functionality:** Allocate specific functions from requirements to components. Traceability is key.

FIGURE 9.8 Overall architectural structure for *SafeHome* with top-level components



Defining Interfaces & Patterns

Specifying the Connections

- **Define Interfaces**: For each component, you must specify:

  Provided Interfaces: Services it offers to others.

  Required Interfaces: Services it needs from others to function.

- **Apply Design Patterns**: Use lower-level patterns (e.g., Factory, Strategy) inside the components to solve specific problems.
- **The Deliverable**: A refined Component Diagram showing components and their dependencies.

Describing Instantiations of the System

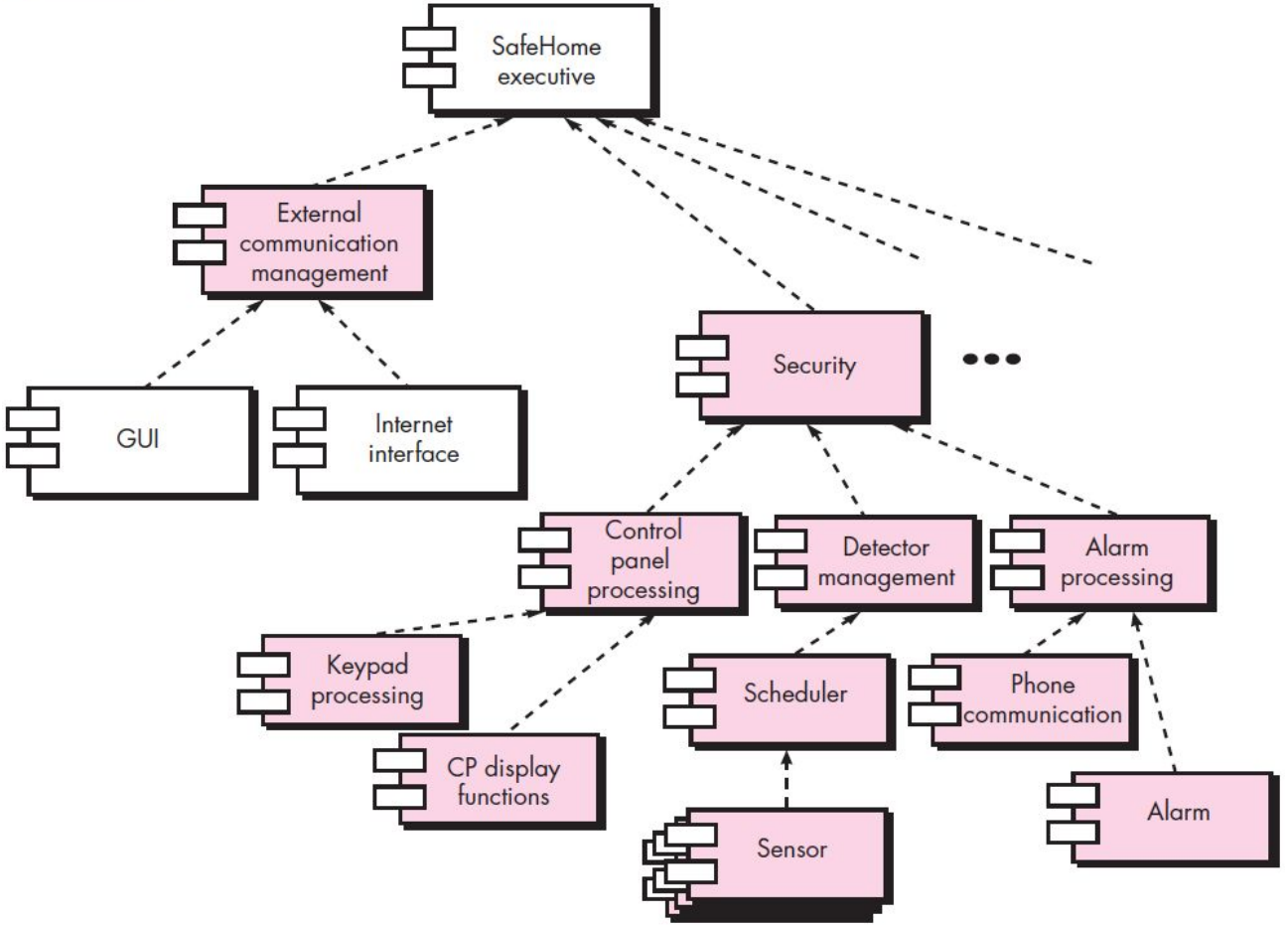
Section 9.4.4: Making it Concrete

- **Definition:** An instantiation is a concrete manifestation of the architecture for a specific set of requirements.
- **The Analogy:**

  Architecture: The “3-bedroom house” generic blueprint.

  Instantiation: The specific house built on Lot #5, with a red roof and a specific garage layout.

FIGURE 9.9 An instantiation of the security function with component elaboration



How to Describe an Instantiation

Populating the Structure

The Method: Show how the generic architecture is populated with actual components for this project.

- **Layered**: Fill the "Business Layer" with actual packages like UserManagement and OrderProcessing.
- **Client-Server**: Specify the exact number of clients and server nodes.
- **Pipe-and-Filter**: Define the actual filters (e.g., SpamFilter, LogParser).

Purpose: Demonstrates that the abstract architecture actually works for the specific problem at hand.

Part 2: Assessing Architectural Designs

Section 9.5

Focus: Systematic Evaluation, Complexity, and Formal Description

The Critical Question

How do we know if our architecture is good?

- **The Reality:** We cannot just "guess." We must evaluate systematically.
- **The Method:** ATAM (Architecture Trade-Off Analysis Method).
- **Core Philosophy:** Architecture is a game of trade-offs.

  Example: ↑ Improving Performance might ↓ hurt Modifiability.

  Example: ↑ Increasing Security might ↓ lower Usability.

ATAM (Architecture Trade-Off Analysis Method)
Section 9.5.1: The Process (Steps 1-3)

- 1. Present Architecture: The architect describes the drivers, styles, patterns, and views.
- 2. Identify Approaches: Document the key design decisions (e.g., "We used a 3-tier style").
- 3. Generate Quality Attribute Utility Tree:
  - Stakeholders define specific scenarios for critical attributes (Performance, Security, Availability).
  - Each scenario is prioritized (High/Medium/Low).

ATAM Analysis
Section 9.5.1: The Process (Steps 4-6)

- 4. Analyze Approaches: Evaluate how the design supports the high-priority scenarios.
- 5. Identify Key Points:
  - Sensitivity Points: A parameter that drastically affects one quality attribute (e.g., "Buffer size affects latency").
  - Trade-off Points: A parameter that affects multiple attributes in opposing ways (e.g., "Encryption level improves Security but degrades Performance").
- 6. Outcome: A prioritized list of Risks (vulnerabilities) and Non-Risks (good decisions).

Architectural Complexity
Section 9.5.2: Measuring Difficulty

Definition: A measure of the difficulty in understanding, maintaining, and evolving the architecture.

Key Factors:

- Number of components.
- Density of interconnections (High connector-to-component ratio).
- Heterogeneity of technologies (Mixing too many languages/frameworks).

Goal: Minimize unnecessary complexity. A simpler architecture is almost always more robust.

Architectural Description Languages (ADLs)
Section 9.5.3: Formal Specification

- What are they?: Formal languages used to model and specify software architecture.
- Components: They formally describe Components, Connectors, Configurations, and Constraints.
- Examples:
  - Acme: Generic interchange language.
  - AADL: Popular for real-time/embedded systems.
  - Wright: Focuses on deadlock analysis.
- Purpose: Enables computer-aided analysis (simulation, verification) and sometimes automatic code generation.
- Context: More rigorous than UML, but less commonly used in general industry.

Part 3: Architectural Mapping Using Data Flow

Section 9.6

Focus: Deriving Architecture from Analysis Models

A Systematic Method

From DFD to Structure

The Challenge: How do we bridge the gap between **analysis** (DFD) and **design** (Architecture)?

The Premise: The structure of the Data Flow Diagram (DFD) reveals the **natural partitions** of the system.

The Goal: To derive a Call-and-Return architecture (like layered or hierarchical) directly from the flow of data.

Types of Data Flow

Identifying the Pattern

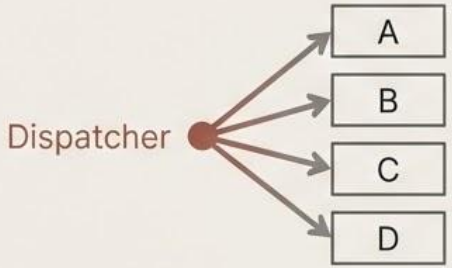
1. Transform Flow

Data enters, undergoes distinct transformations, and exits.

Linear “Transform Center”

2. Transaction Flow

A single data item (the transaction) triggers one of many possible action paths.



Distinct “Transaction Center” (dispatcher)

Transform Mapping: Step 1

Isolating the Center

Action: Review the DFD to identify the core processes.

Identify Flows:

Afflerent Flow: Incoming paths that bring external data into the system.

Efferent Flow: Outgoing paths that format and send results out.

Transform Center: The central processes between input and output where the ‘real work’ happens.

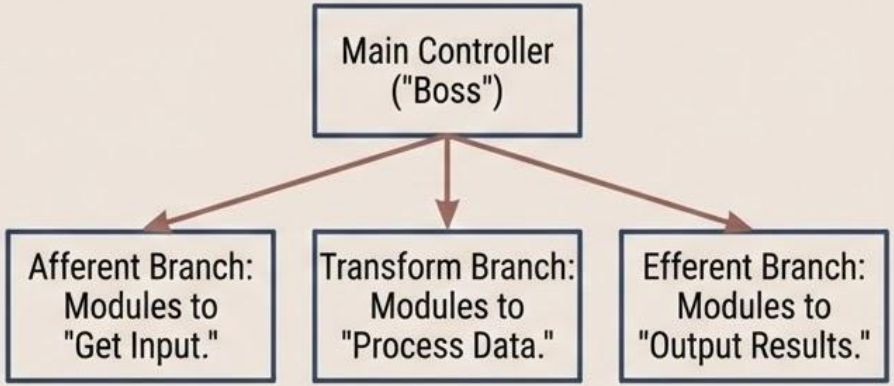
Transform Mapping: Step 2

First-Level Factoring

Action: Design the top-level hierarchical structure.

Create the **Main Controller**: A "Boss" module that coordinates everything.

Create Subordinate Branches:



Transform Mapping: Step 3

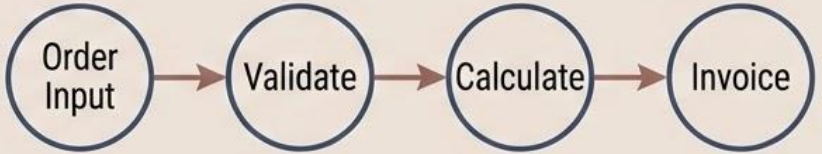
Second-Level Factoring & Mapping

Action: Decompose each branch by mapping individual DFD “bubbles” (processes) to sub-modules.

Result: A Program Structure Chart (Hierarchy Chart).

Example:

DFD:



Structure:

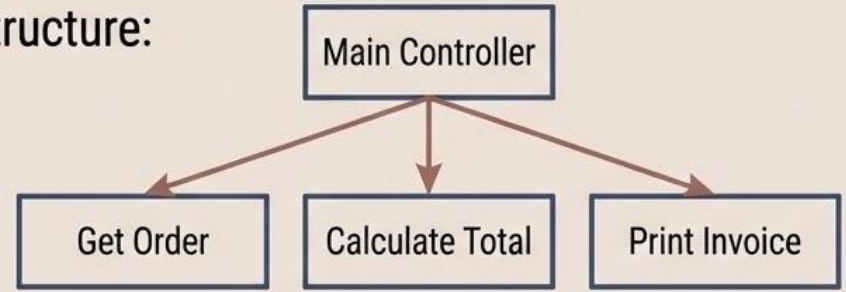


FIGURE 9.10

Context-level DFD for the *SafeHome* security function

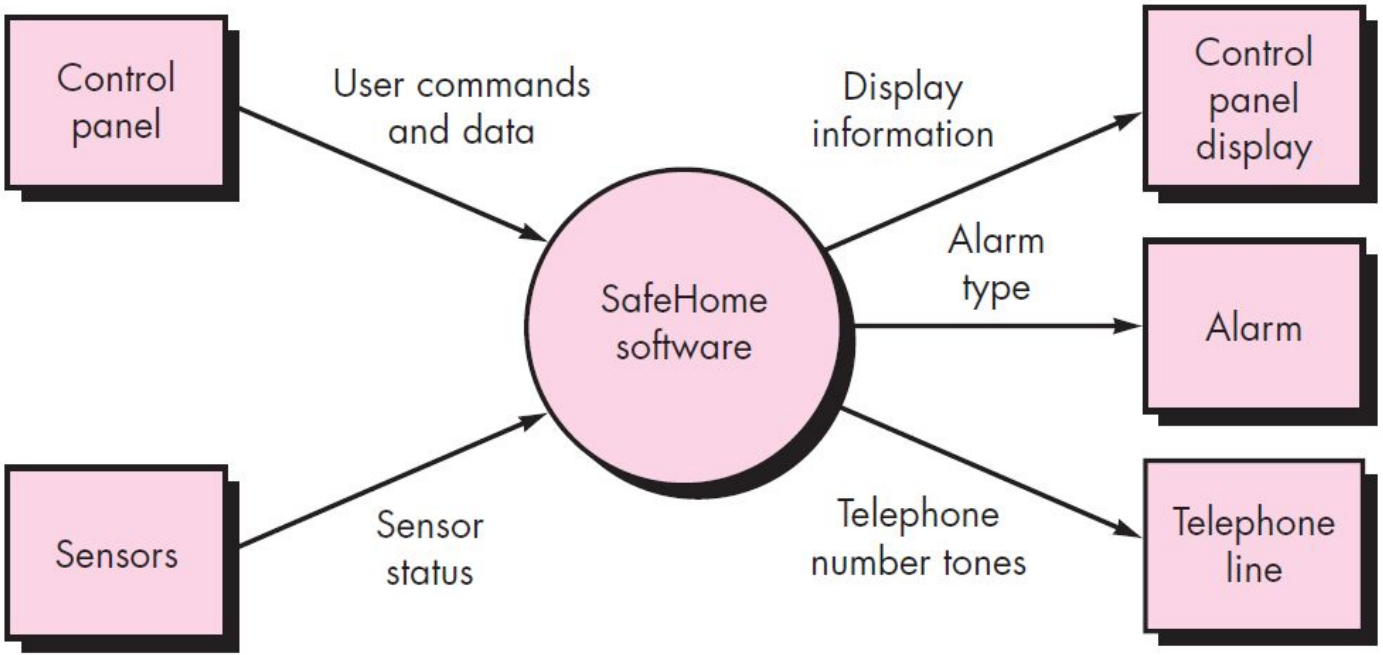


FIGURE 9.11

Level 1 DFD for the *SafeHome* security function

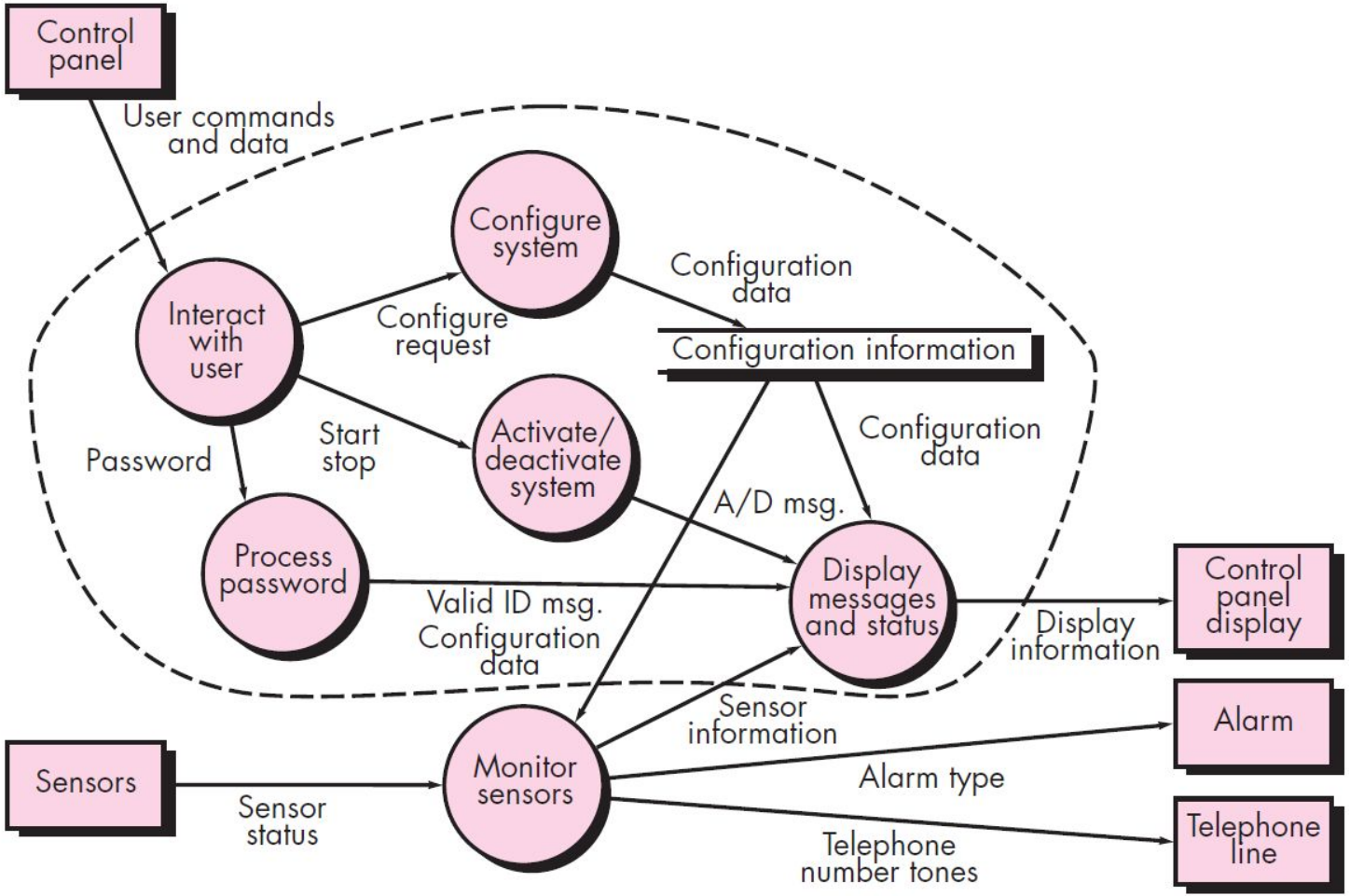


FIGURE 9.12

Level 2 DFD that refines the *monitor sensors* transform

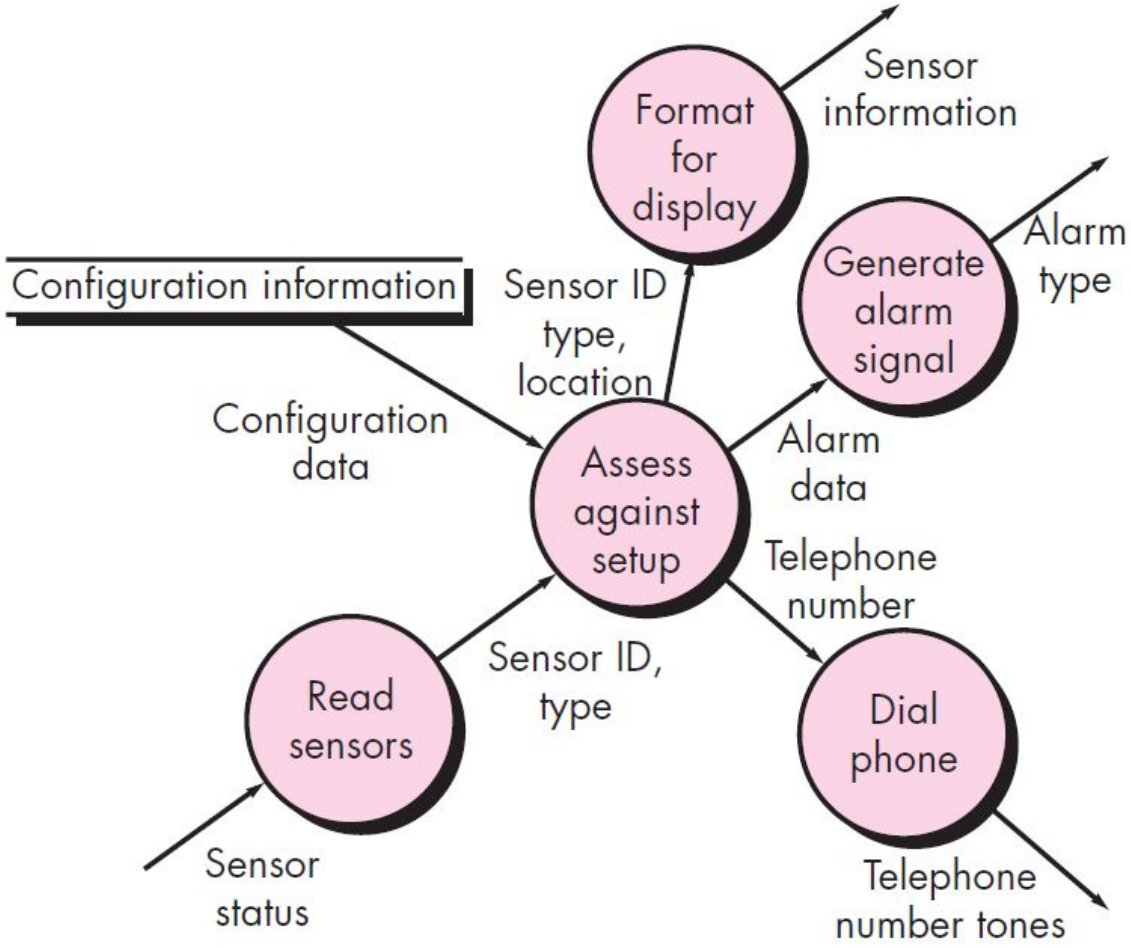


FIGURE 9.13 Level 3 DFD for *monitor sensors* with flow boundaries

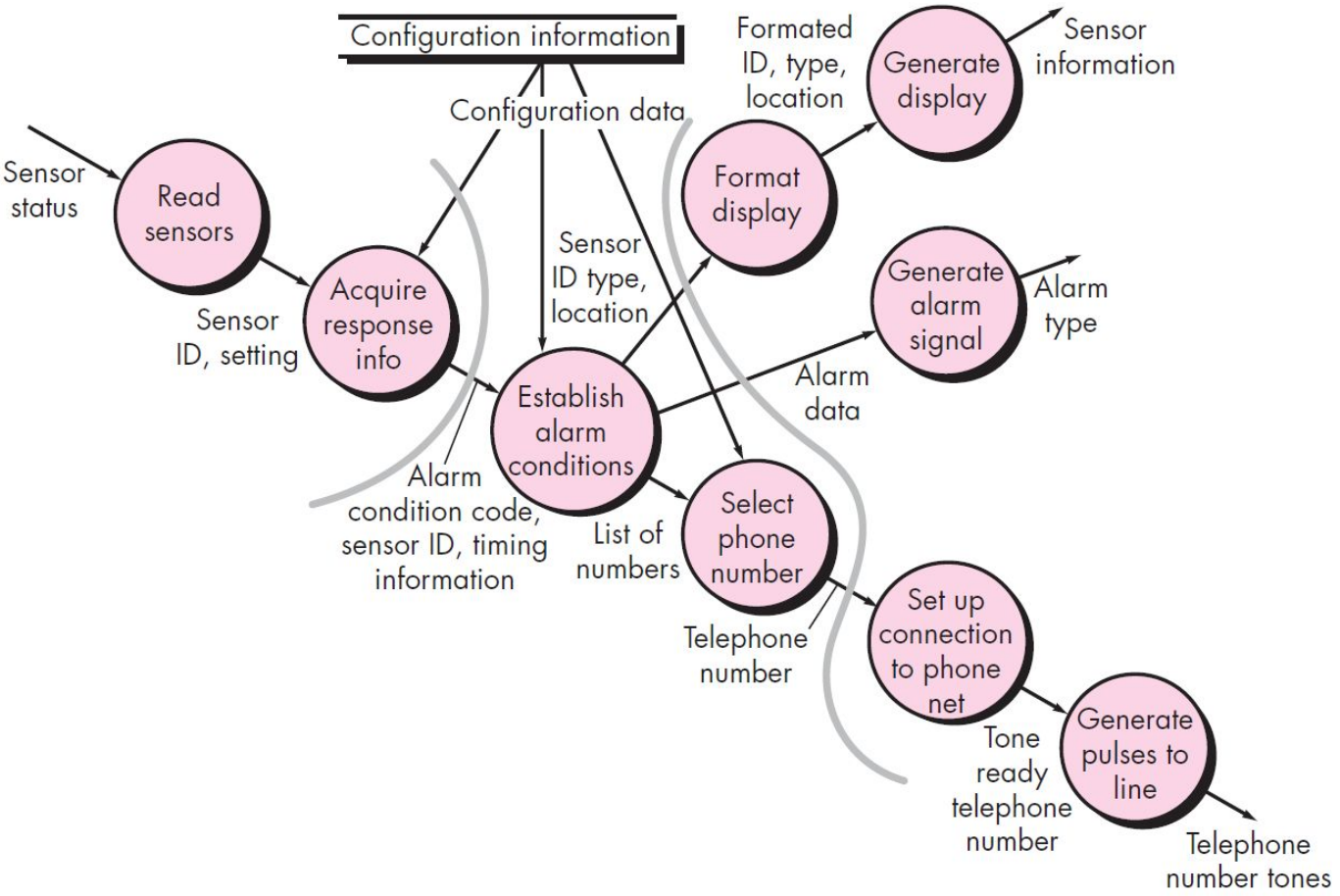


FIGURE 9.14

First-level factoring for *monitor sensors*

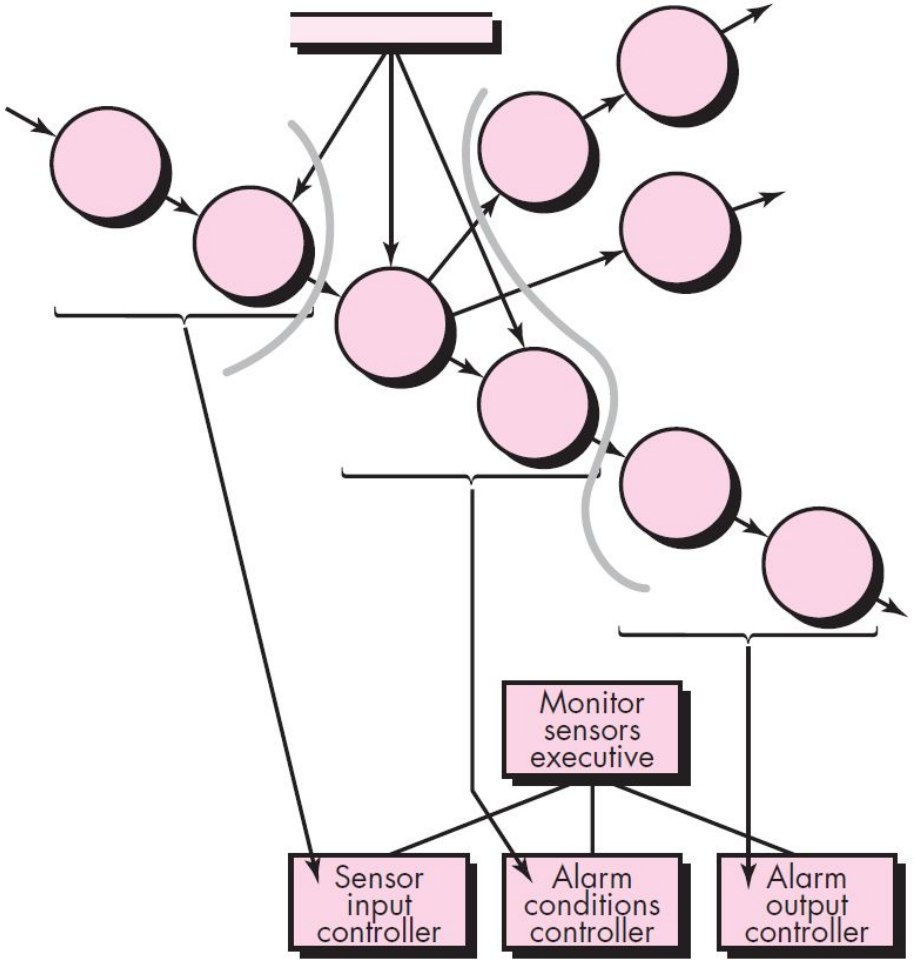


FIGURE 9.15

Second-level factoring for *monitor sensors*

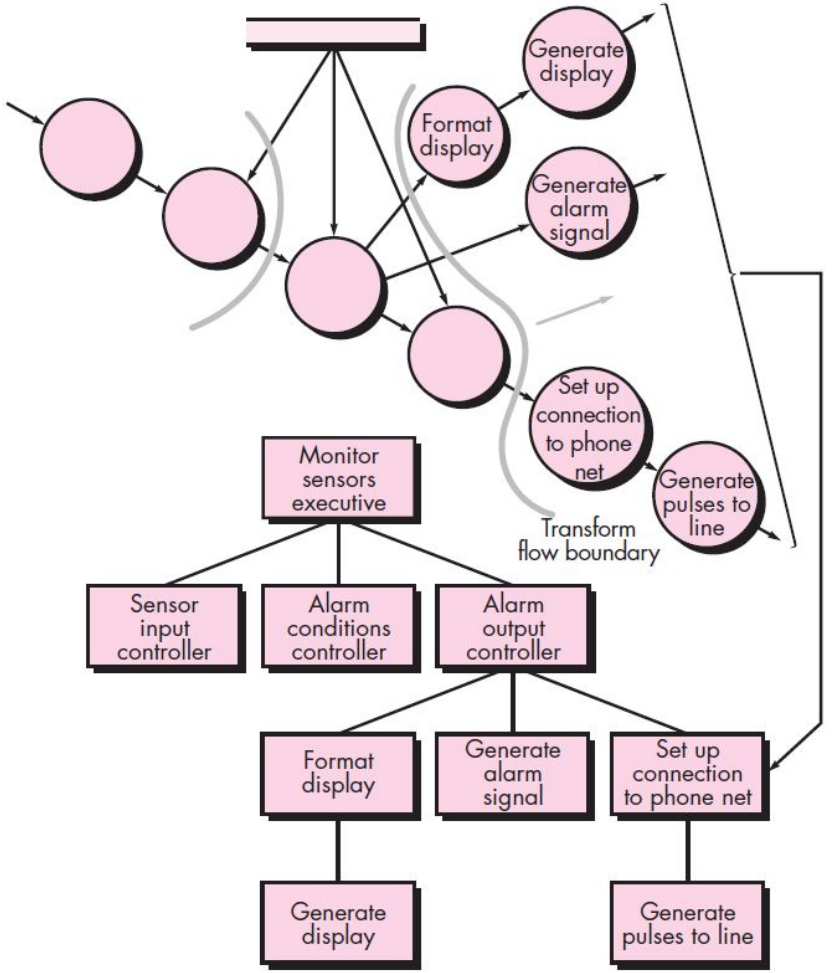


FIGURE 9.16

First-iteration structure for *monitor sensors*

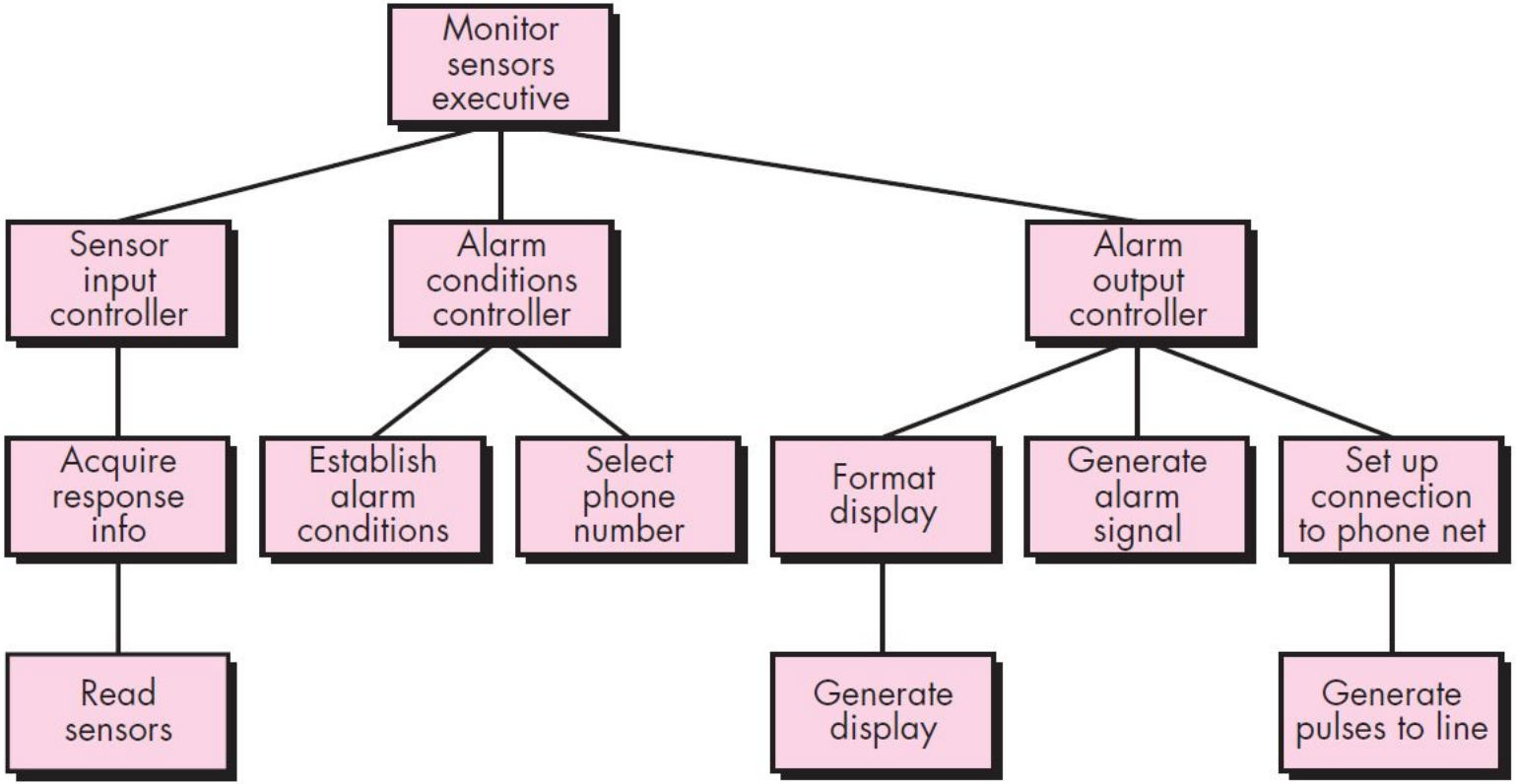
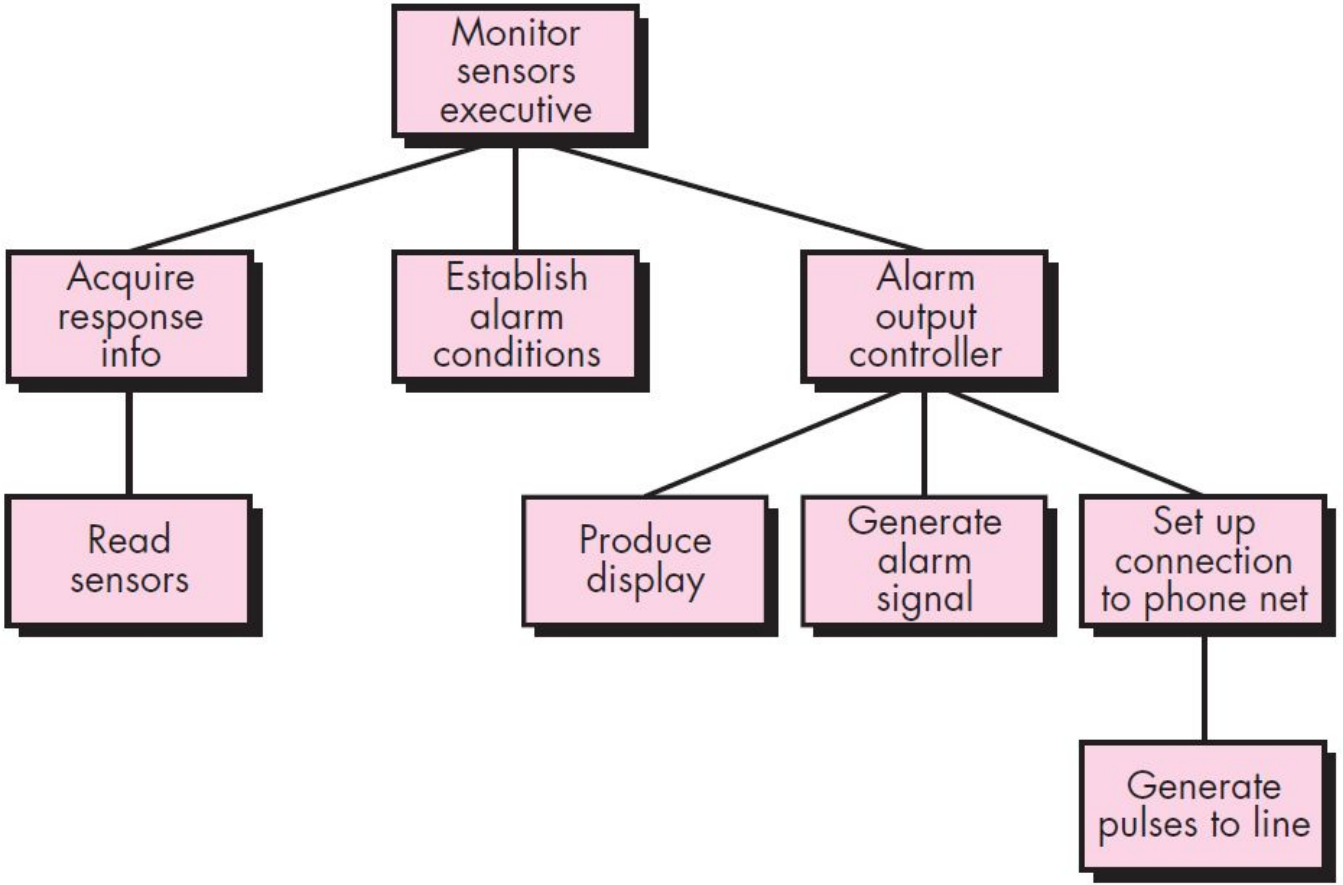


FIGURE 9.17

Refined program structure for *monitor sensors*



Refining the Design

Section 9.6.2: From Raw to Polished

Evaluate Quality: Once mapped, apply design concepts.

Coupling: Are connections too tight?

Cohesion: Does each module do only one thing?

Optimize: Merge tiny modules or split complex ones.

Define Data Structures: Formalize the exact data passed between these new modules.

The Architect's Toolkit

Refinement, Assessment, and Systematic Derivation

Wrapping up the transition from concept to concrete engineering.

Summary of Core Concepts

What We Learned Today

Refinement: The process of populating abstract styles with specific, named components and demonstrating concrete instantiations (e.g., "The Red Roof House").

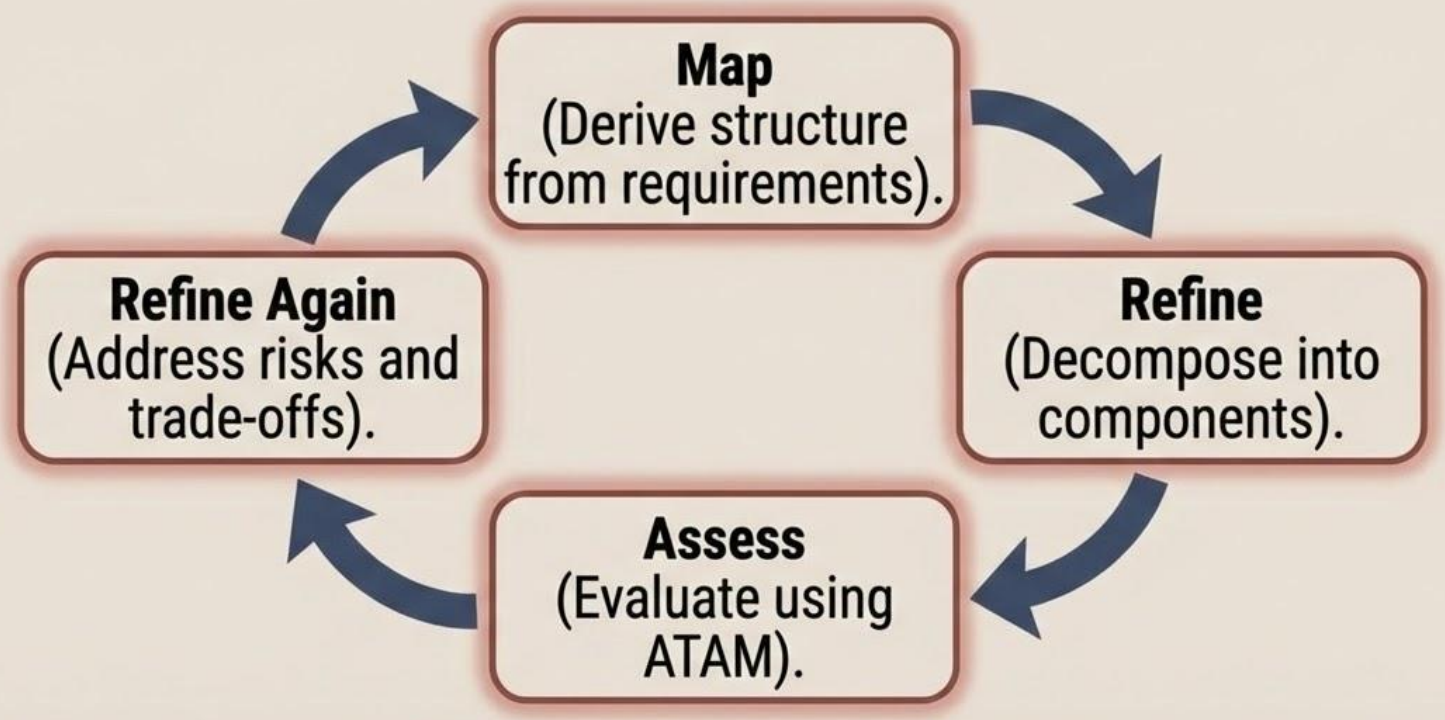
Assessment (ATAM): A structured method to evaluate design quality.

Key Insight: Architecture is about managing trade-offs (e.g., Performance vs. Modifiability).

Transform Mapping: A systematic, requirements-driven technique to derive a Call-and-Return architecture directly from Data Flow Diagrams (DFDs).

The Architectural Lifecycle

It Is Not a Linear Path



Final Thought

Architecture as Engineering

> Architecture is not a one-time sketch. It is an engineered artifact that must be constructed methodically, evaluated rigorously, and refined continuously. The techniques we learned today—ATAM and Transform Mapping—are the engineer's tools for that construction and evaluation.