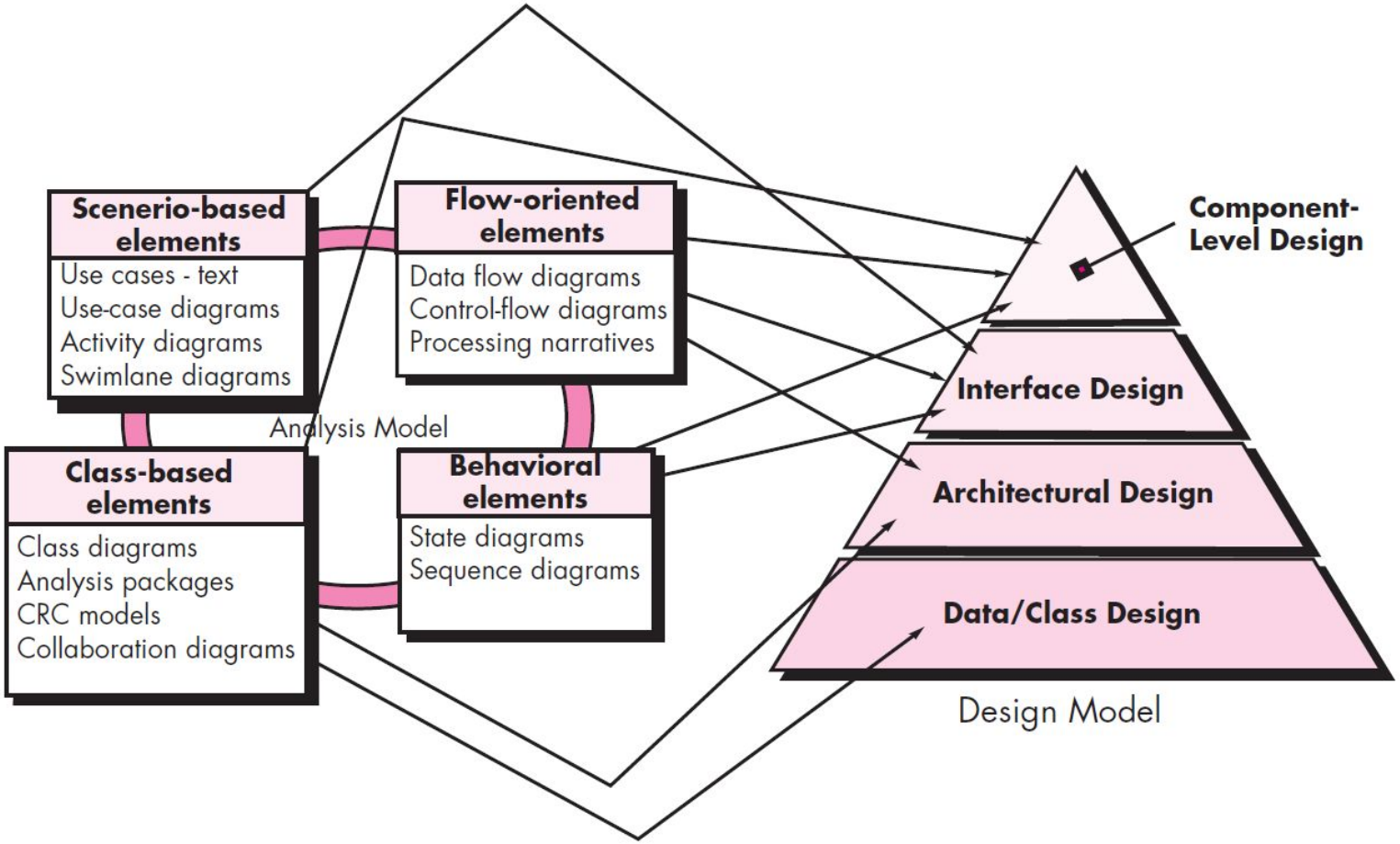
Software Design Concepts

Ashish Kumar Dwivedi

Translating the requirements model into the design model



Design Documents

- The data/class design transforms class models into design class realizations and the requisite data structures required to implement the software.
- The objects and relationships defined in the CRC diagram and the detailed data content depicted by class attributes and other notation provide the basis for the data design action.
- The **architectural design** defines the relationship between major structural elements of the software, the architectural styles and design patterns that can be used to achieve the requirements defined for the system.
- The **interface design** describes how the software communicates with systems that interoperate with it, and with humans who use it.
- An **interface** implies a flow of information (e.g., data and/or control) and a specific type of behavior. Therefore, usage scenarios and behavioral models provide much of the information required for interface design.
- The **component-level** design transforms structural elements of the software architecture into a procedural description of software components. Information obtained from the class-based models, flow models, and behavioral models serve as the basis for component design.

Design Concepts

- Abstraction:
  - **A *procedural abstraction*** refers to a sequence of instructions that have a specific and limited function. The name of a procedural abstraction implies these functions, but specific details are suppressed. An example of a procedural abstraction would be the word *open* for a door. *Open* implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.)
  - **A *data abstraction*** is a named collection of data that describes a data object. In the context of the procedural abstraction *open,* we can define a data abstraction called **door.** Like any data object, the data abstraction for **door** would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions).

Design Concepts

- Architecture:
  - The overall structure of the software and the ways in which that structure provides conceptual integrity for a system.
  - One goal of software design is to derive an architectural rendering of a system. This rendering serves as a framework from which more detailed design activities are conducted.
  - Architecture Styles, Architectural Patterns, Architectural Views
- Separation of Concern:
  - *Separation of concerns* is a design concept that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently.
  - For two problems, *p*1 and *p*2, if the perceived complexity of *p*1 is greater than the perceived complexity of *p*2, it follows that the effort required to solve *p*1 is greater than the effort required to solve *p*2.

Design Concepts

- Modularity:
  - Modularity is the most common manifestation of separation of concerns.
- Information Hiding
  - The concept of modularity leads you to a fundamental question: “How do I decompose a software solution to obtain the best set of modules?”
  - Modules should be specified and designed so that information (algorithms and data) contained within a module is inaccessible to other modules that have no need for such information.
  - Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure
- Functional Independence
  - The concept of functional independence is a direct outgrowth of separation of concerns, modularity, and the concepts of abstraction and information hiding.
  - Functional independence is achieved by developing modules with “single-minded” function
  - Independent modules are easier to maintain (and test)

Design Concepts

- Refinement:
  - Stepwise refinement is a top-down design strategy, A program is developed by successively refining levels of procedural detail.
  - Refinement is actually a process of *elaboration.*
  - Abstraction and refinement are complementary concepts
  - Refinement helps you to reveal low-level details as design progresses
- Aspects
  - As design begins, requirements are refined into a modular design representation. Consider two requirements, *A* and *B.* Requirement *A* *crosscuts* requirement *B* “if a software decomposition [refinement] has been chosen in which *B* cannot be satisfied without taking *A* into account”
- Refactoring
  - An important design activity suggested for many agile methods
  - “Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure.”

Bottom-up vs Top-down approach

- Object-oriented programming is often referred to as a bottom-up approach because it focuses on the creation and manipulation of small, reusable objects or "building blocks" that can be combined to create larger, more complex systems.
- This approach is in contrast to a top-down approach, which focuses on breaking larger systems down into smaller, simpler parts. The bottom-up approach is often used in the development of software applications, where the focus is on creating small, modular pieces of code that can be easily reused and combined to create more complex functionality.

Top-down vs Bottom-up approach

Difference between top-down and bottom-up Approach

Basis	Top-Down Approach	Bottom-Up Approach
Approach	Top-Down Approach is Theory-driven.	Bottom-Up Approach is Data-Driven.
Significance	Emphasis is on doing things (algorithms).	Emphasis is on data rather than procedure.
Focus	Large programs are divided into smaller programs which is known as decomposition.	Programs are divided into what are known as objects is called Composition.
Interaction	Communication is less among the modules.	Communication is a key among the modules.
Areas	Widely used in debugging, module documentation, etc.	Widely used in testing.
Language	The top-down approach is mainly used by Structured programming languages like C, Fortran, etc.	The bottom-up approach is used by Object-Oriented programming languages like C++, C#, Java, etc.
Redundancy	May contains redundancy as we break up the problem into smaller fragments, then build that section separately.	This approach contains less redundancy if the data encapsulation and data hiding are being used.

Design Concepts

The scene: Vinod's cubicle, as design modeling begins.

The players: Vinod, Jamie, and Ed—members of the *SafeHome* software engineering team. Also, Shakira, a new member of the team.

The conversation:

[All four team members have just returned from a morning seminar entitiled "Applying Basic Design Concepts," offered by a local computer science professor.]

Vinod: Did you get anything out of the seminar?

Ed: Knew most of the stuff, but it's not a bad idea to hear it again, I suppose.

Jamie: When I was an undergrad CS major, I never really understood why information hiding was as important as they say it is.

Vinod: Because . . . bottom line . . . it's a technique for reducing error propagation in a program. Actually, functional independence also accomplishes the same thing.

Shakira: I wasn't a CS grad, so a lot of the stuff the instructor mentioned is new to me. I can generate good code and fast. I don't see why this stuff is so important.

Jamie: I've seen your work, Shak, and you know what, you do a lot of this stuff naturally . . . that's why your designs and code work.

Shakira (smiling): Well, I always do try to partition the code, keep it focused on one thing, keep interfaces simple and constrained, reuse code whenever I can . . . that sort of thing.

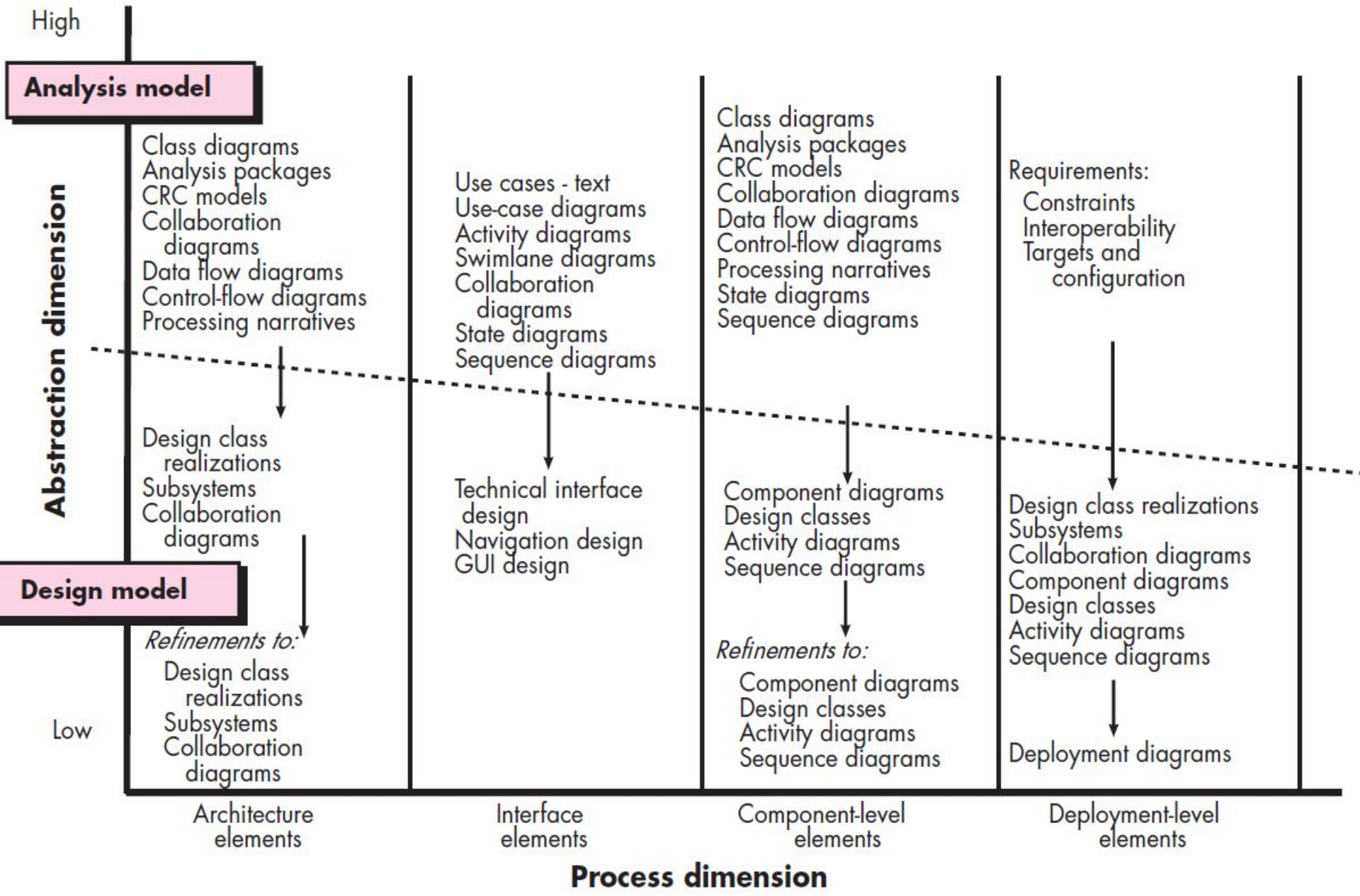
Ed: Modularity, functional independence, hiding, patterns . . . see.

Jamie: I still remember the very first programming course I took . . . they taught us to refine the code iteratively.

Vinod: Same thing can be applied to design, you know.

Vinod: The only concepts I hadn't heard of before were "aspects" and "refactoring."

FIGURE 8.4 Dimensions of the design model

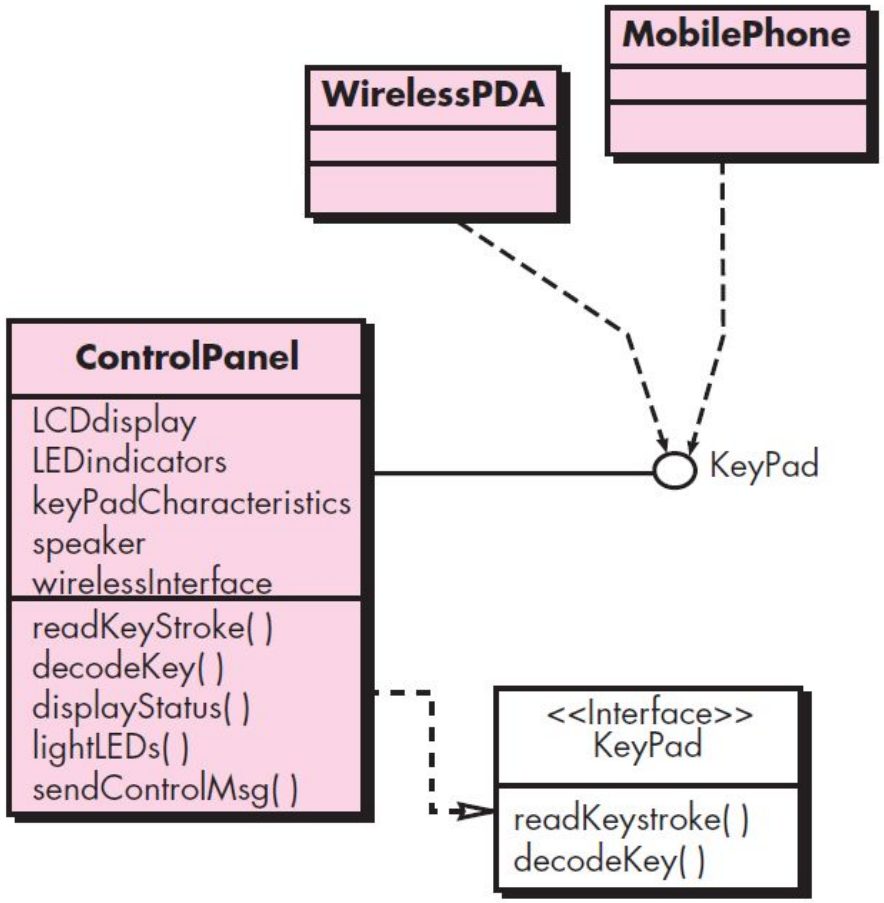


Design Elements

- Data Design Elements:
  - Data structure is an essential part of design
  - At the application level, the translation of a data model (derived as part of requirements engineering) into a database is pivotal to achieving the business objectives of a system.
  - At the business level, the collection of information stored in disparate databases and reorganized into a “data warehouse” enables data mining
- Architectural Design Elements:
  - The architectural model is derived from three sources:
  - (1) information about the application domain for the software to be built;
  - (2) specific requirements model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand; and
  - (3) the availability of architectural styles
- Interface Design Elements:
  - UI
  - External Interface to other systems, devices, networks
  - Internal Interface

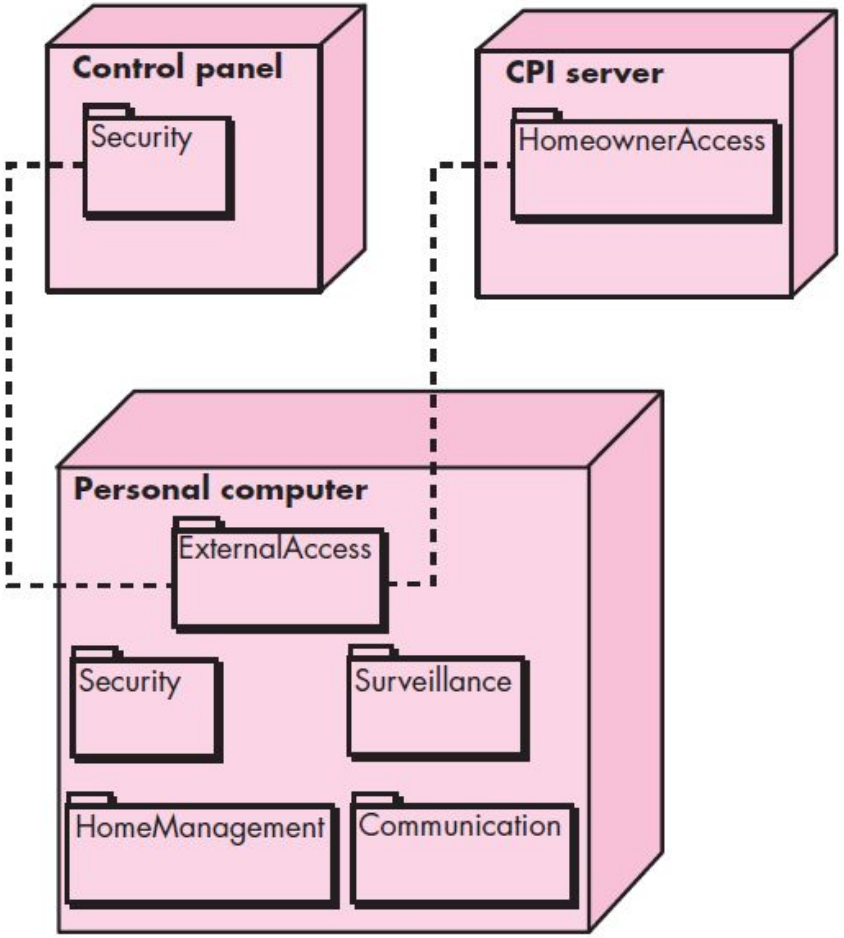
Interface design

The behavior of **ControlPanel** will be implemented by realizing **KeyPad** operations



Design Elements

- Component-Level Design Elements:
- Deployment-Level Design Elements:



Description

- The personal computer houses subsystems that implement security, surveillance, home management, and communications features.
- In addition, an external access subsystem has been designed to manage all attempts to access the *SafeHome* system from an external source.
- Software design commences as the first iteration of requirements engineering comes to a conclusion.
- The intent of software design is to apply a set of principles, concepts, and practices that lead to the development of a high-quality system or product.
- The goal of design is to create a model of software that will implement all customer requirements correctly and bring delight to those who use it.

Architectural Design

- **Definition 1**: The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.
- **Definition 2:** There is a distinct difference between the terms architecture and design. A *design* is an instance of an *architecture* like an object being an instance of a class. For example, consider the client-server architecture. I can design a network-centric software system in many ways from this architecture using either the Java platform (Java EE) or Microsoft platform (.NET framework).

Importance of Architecture

- Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.
- The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.
- Architecture “constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together”.

INFO

Architecture Decision Description Template

Each major architectural decision can be documented for later review by stakeholders who want to understand the architecture description that has been proposed. The template presented in this sidebar is an adapted and abbreviated version of a template proposed by Tyree and Ackerman [Tyr05].

Design issue: Describe the architectural design issues that are to be addressed.

Resolution: State the approach you've chosen to address the design issue.

Category: Specify the design category that the issue and resolution address (e.g., data design, content structure, component structure, integration, presentation).

Assumptions: Indicate any assumptions that helped shape the decision.

Constraints: Specify any environmental constraints that helped shape the decision (e.g., technology standards, available patterns, project-related issues).

Alternatives: Briefly describe the architectural design alternatives that were considered and why they were rejected.

Argument: State why you chose the resolution over other alternatives.

Implications: Indicate the design consequences of making the decision. How will the resolution affect other architectural design issues? Will the resolution constrain the design in any way?

Related decisions: What other documented decisions are related to this decision?

Related concerns: What other requirements are related to this decision?

Work products: Indicate where this decision will be reflected in the architecture description.

Notes: Reference any team notes or other documentation that was used to make the decision.

Architectural Styles

- The software that is built for computer-based systems also exhibits one of many architectural styles.
- An architectural style is a transformation that is imposed on the design of an entire system.
- Each style describes a system category that encompasses:
  - (1) a set of components (e.g., a database, computational modules) that perform a function required by a system;
  - (2) a set of connectors that enable “communication, coordination and cooperation” among components;
  - (3) constraints that define how components can be integrated to form the system; and
  - (4) semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts

Architectural Pattern

- An architectural style is a transformation that is imposed on the design of an entire system.
- A pattern differs from a style in several fundamental ways:
  - (1) the scope of a pattern is less broad, focusing on one aspect of the architecture rather than the architecture in its entirety;
  - (2) a pattern imposes a rule on the architecture, describing how the software will handle some aspect of its functionality at the infrastructure level (e.g., concurrency)
  - (3) architectural patterns tend to address specific behavioral issues within the context of the architecture (e.g., how real-time applications handle synchronization or interrupts)

Canonical Architectural Structures

In essence, software architecture represents a structure in which some collection of entities (often called *components*) is connected by a set of defined relationships (often called *connectors*). Both components and connectors are associated with a set of *properties* that allow the designer to differentiate the types of components and connectors that can be used. But what kinds of structures (components, connectors, and properties) can be used to describe an architecture? Bass and Kazman [Bas03] suggest five canonical or foundation architectural structures:

Functional structure. Components represent function or processing entities. Connectors represent interfaces that provide the ability to "use" or "pass data to" a component. Properties describe the nature of the components and the organization of the interfaces.

Implementation structure. "Components can be packages, classes, objects, procedures, functions, methods, etc., all of which are vehicles for packaging functionality at various levels of abstraction" [Bas03]. Connectors include the ability to pass data and control, share data, "use", and "is-an-instance-of." Properties focus on quality characteristics (e.g., maintainability, reusability) that result when the structure is implemented.

Concurrency structure. Components represent "units of concurrency" that are organized as parallel tasks or threads. "Relations [connectors] include synchronizes-with, is-higher-priority-than, sends-data-to, can't-run-without, and can't-run-with. Properties relevant to this structure include priority, preemptability, and execution time" [Bas03].

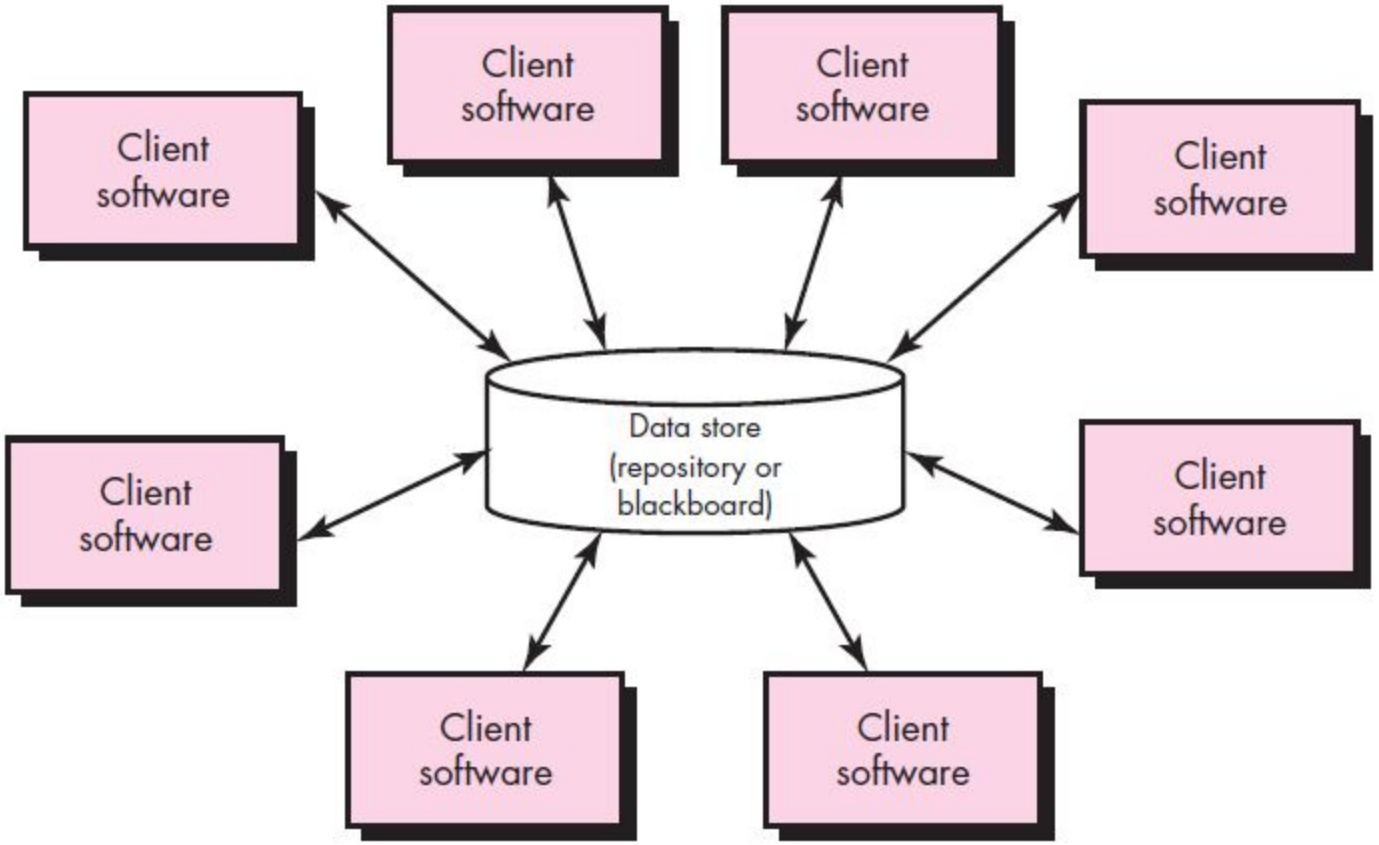
Physical structure. This structure is similar to the deployment model developed as part of design. The components are the physical hardware on which software resides. Connectors are the interfaces between hardware components, and properties address capacity, bandwidth, performance, and other attributes.

Developmental structure. This structure defines the components, work products, and other information sources that are required as software engineering proceeds. Connectors represent the relationships among work products, and properties identify the characteristics of each item.

Each of these structures presents a different view of software architecture, exposing information that is useful to the software team as modeling and construction proceed.

Taxonomy of Architectural Style

- Data-centered Architecture

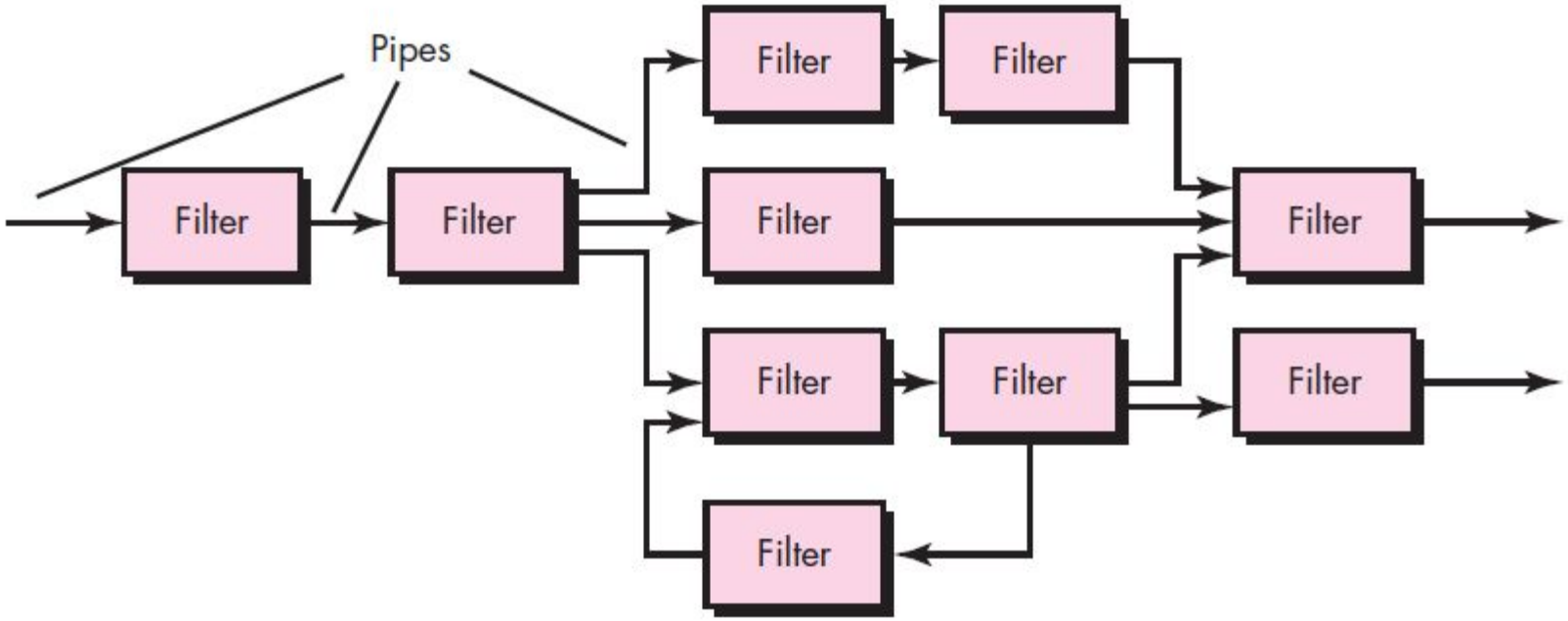


Data-centered Architecture and Data-Flow

- A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store.
- Client software accesses a central repository.
- Data-Flow is applied when input data are to be transformed through a series of computational.
- The filter does not require knowledge of the workings of its neighboring filters.

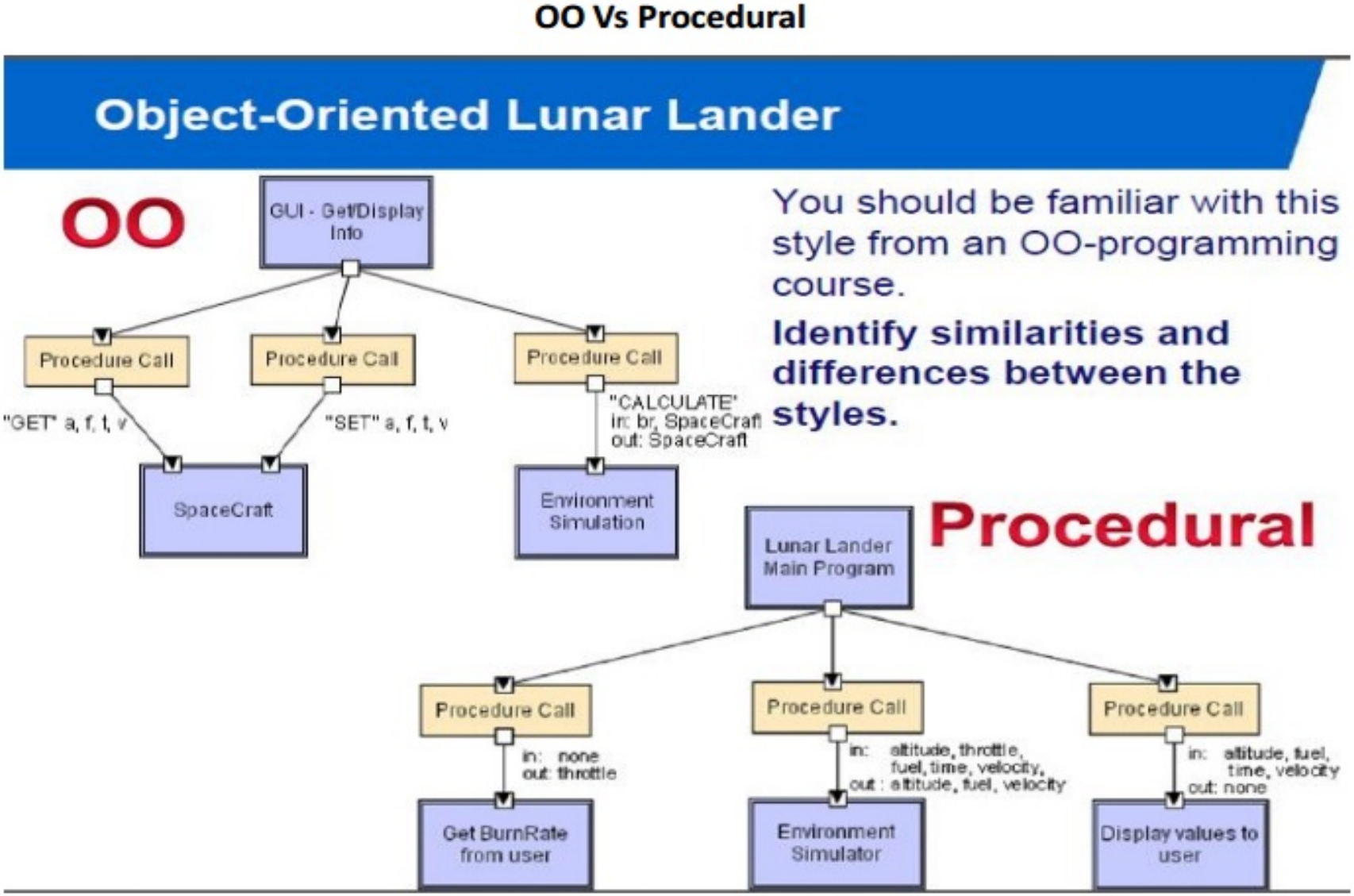
Data-flow Architecture

- A pipe-and-filter pattern has a set of components, called *filters*, connected by *pipes* that transmit data from one component to the next.

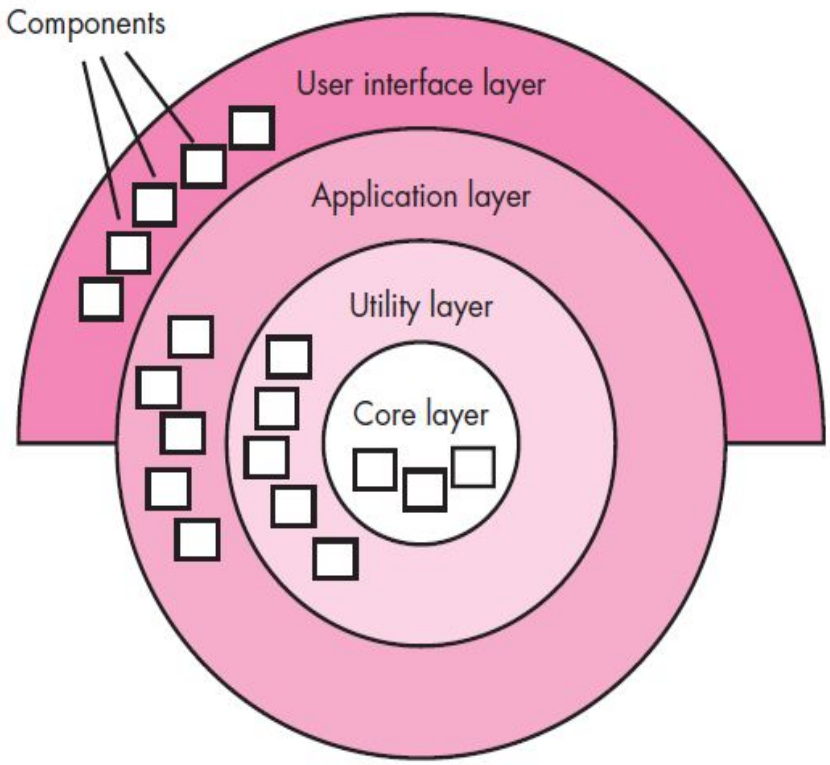


Pipes and filters

Main Program/ Subprogram and Object-Oriented architectures



Layered Architecture



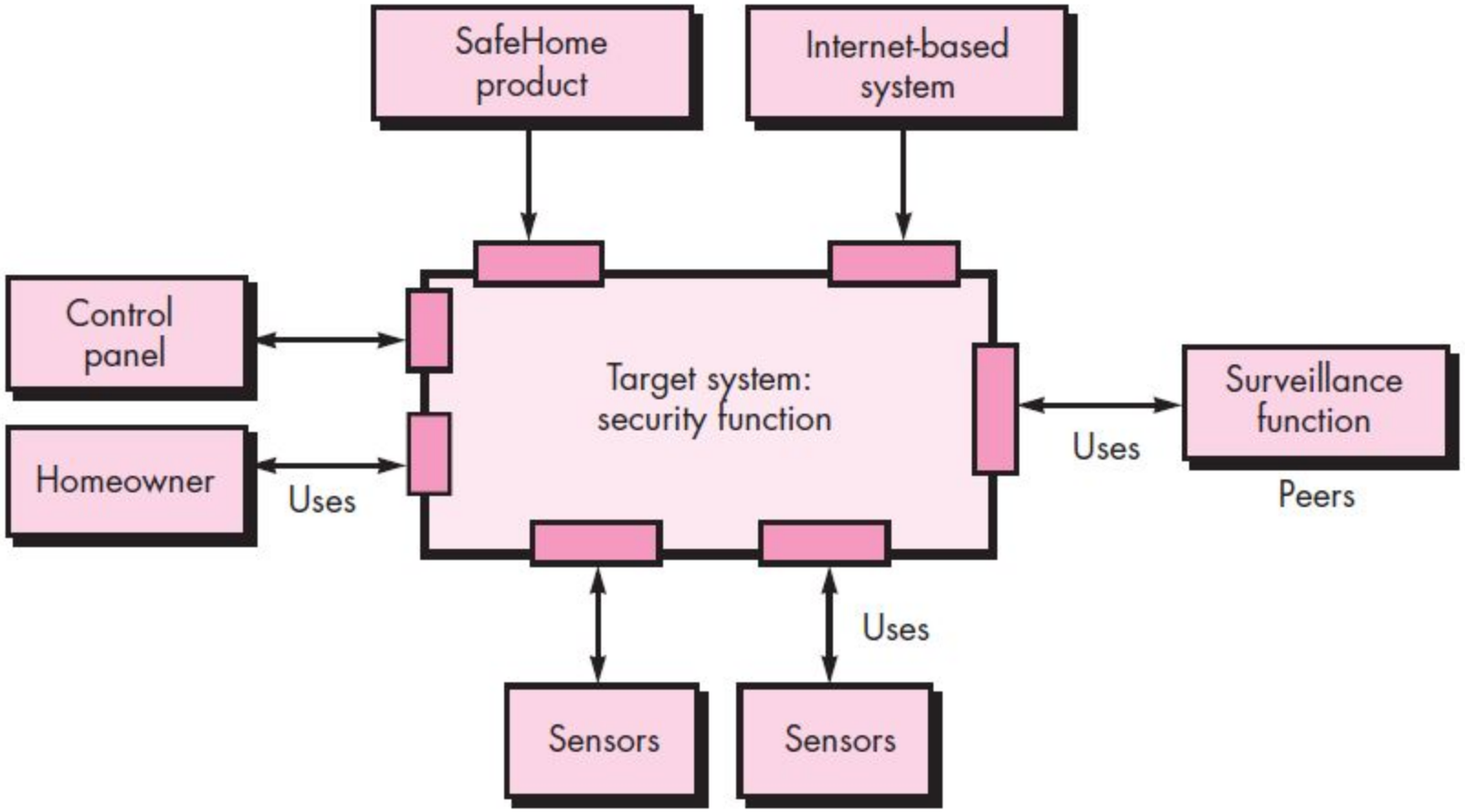
- A number of layers are defined, each accomplishing operations that progressively become closer to the machine instruction set.
- At the outer layer, components service user interface operations.
- At the inner layer, components perform operating system interfacing.
- Intermediate layers provide utility services and application software functions.

Architectural Context Diagram

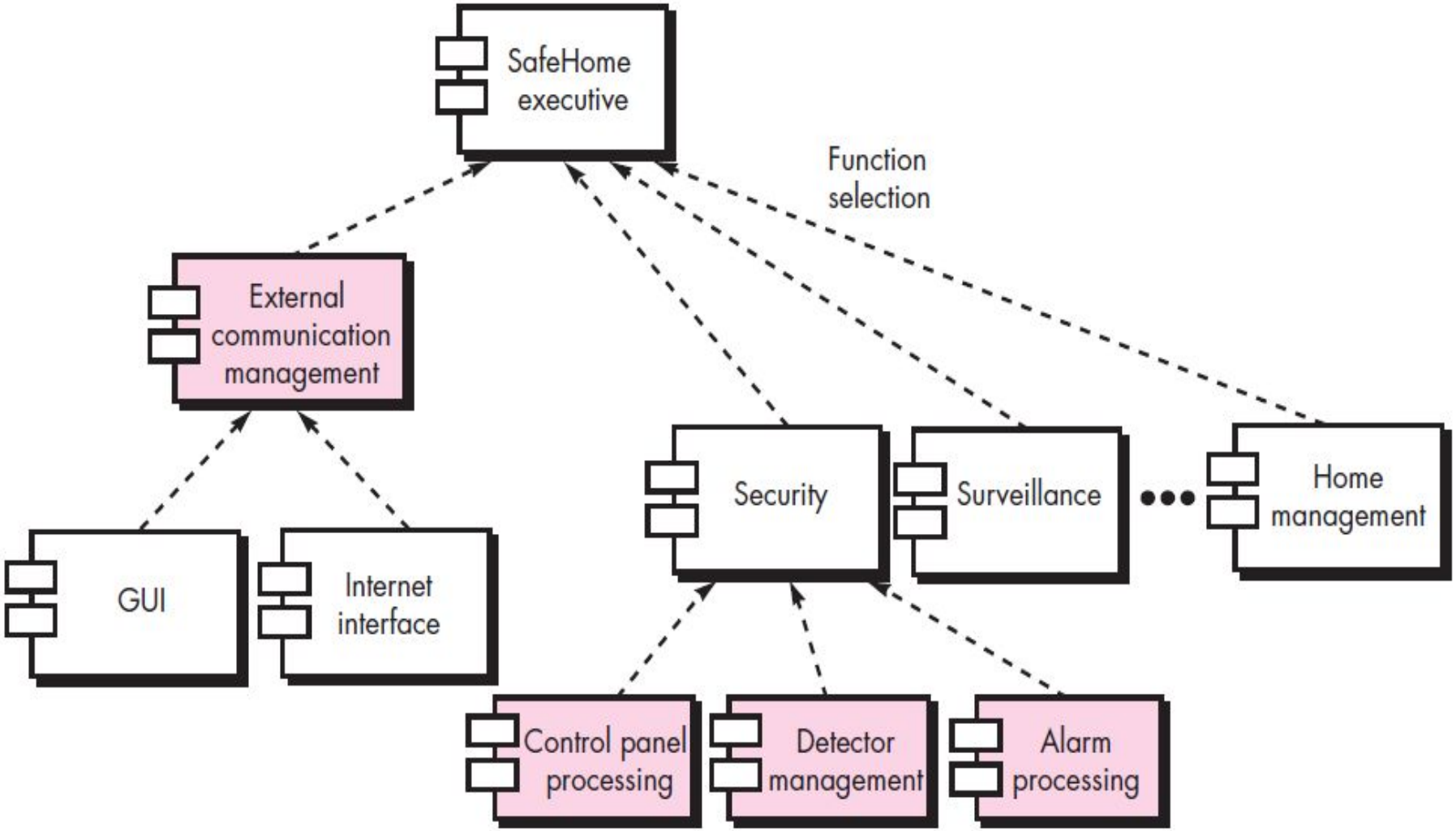
- At the architectural design level, a software architect uses an *architectural context diagram* (ACD) to model the way software interacts with entities external to its boundaries.
  - *Superordinate systems*—those systems that use the target system as part of some higher-level processing scheme.
  - *Subordinate systems*—those systems that are used by the target system and provide data or processing that are necessary to complete target system functionality.
  - *Peer-level systems*—those systems that interact on a peer-to-peer basis (i.e., information is either produced or consumed by the peers and the target system.
  - *Actors*—entities (people, devices) that interact with the target system by producing or consuming information that is necessary for requisite processing.

Architectural Context Diagram

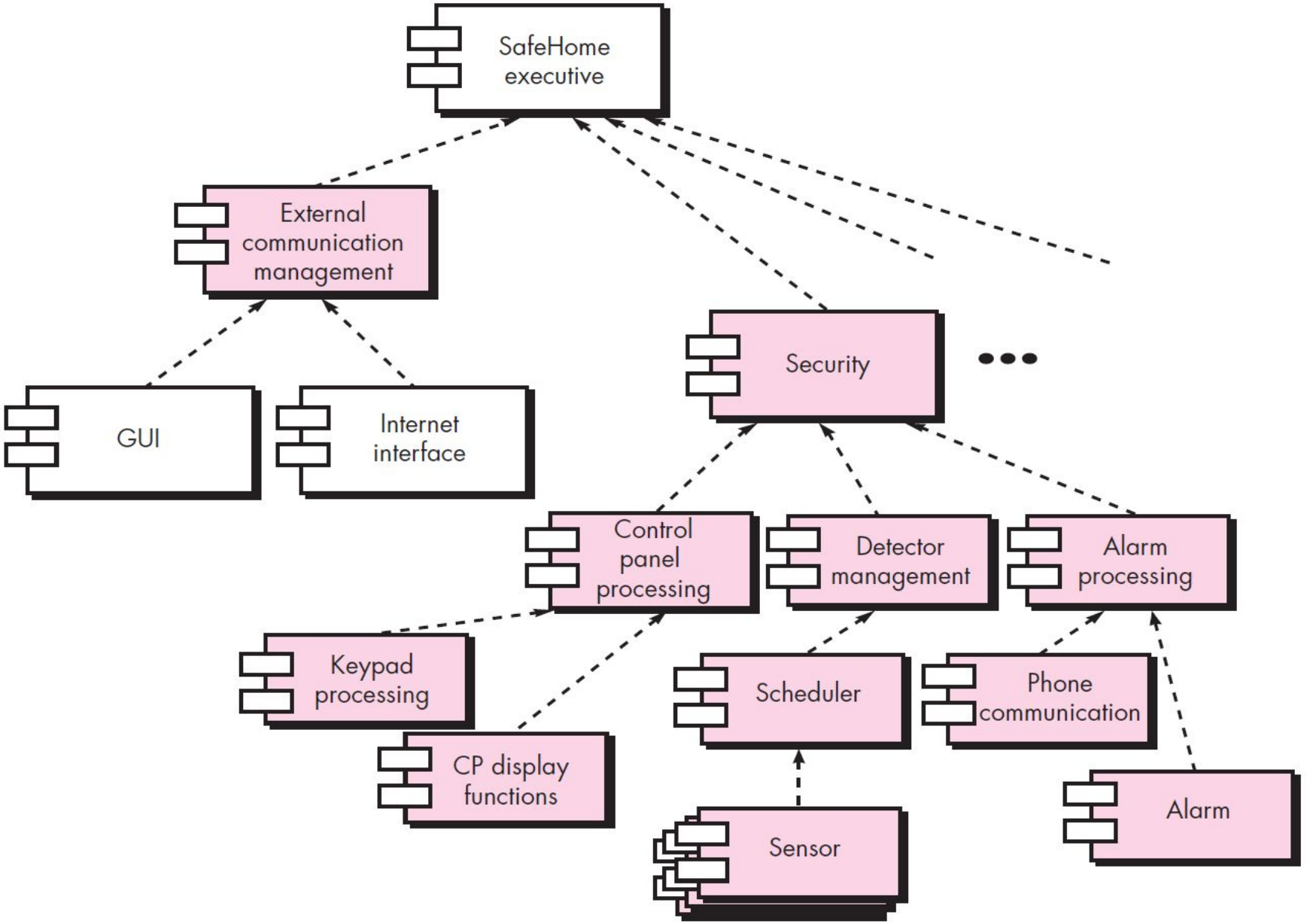
- Architectural context diagram for the *SafeHome* security function



Architectural Structure for SafeHome with top level component



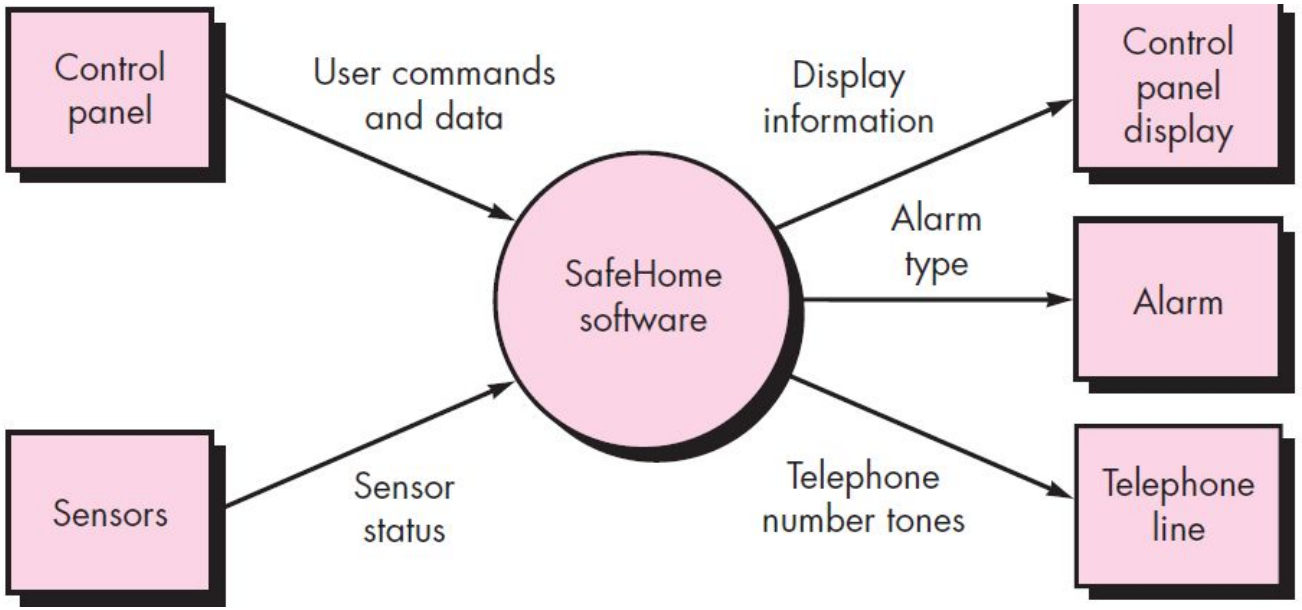
An instantiation of the security function with component elaboration



Transformation of Data-Flow into Software Architecture

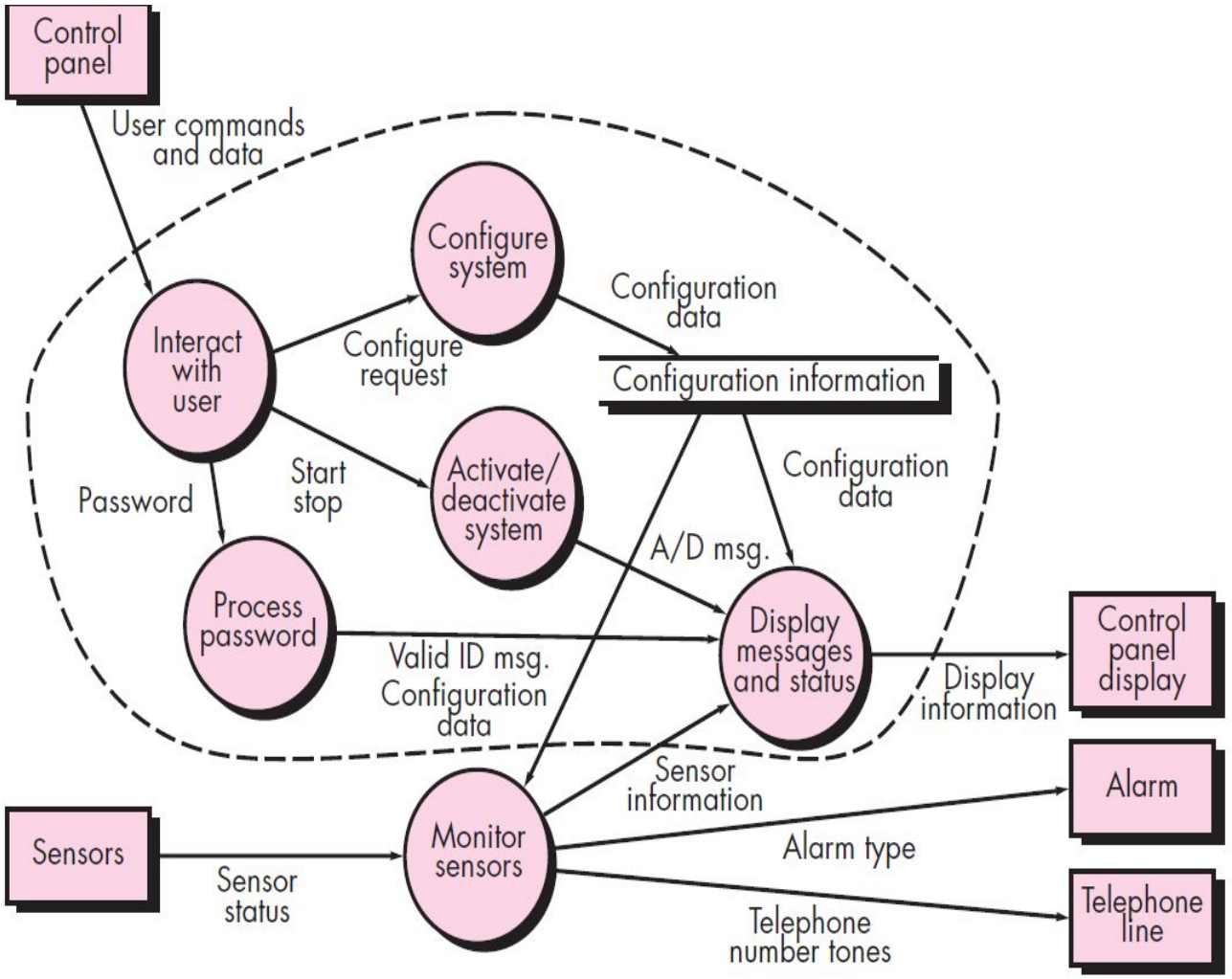
- First understand the system using context diagram

Context-level DFD for the *SafeHome* security function



- **Step 1. Review the fundamental system model.**

Level 1 DFD for the *SafeHome* security function



- **Step 2. Review and refine data flow diagrams for the software.**

Level 2 DFD that refines the *monitor sensors* transform

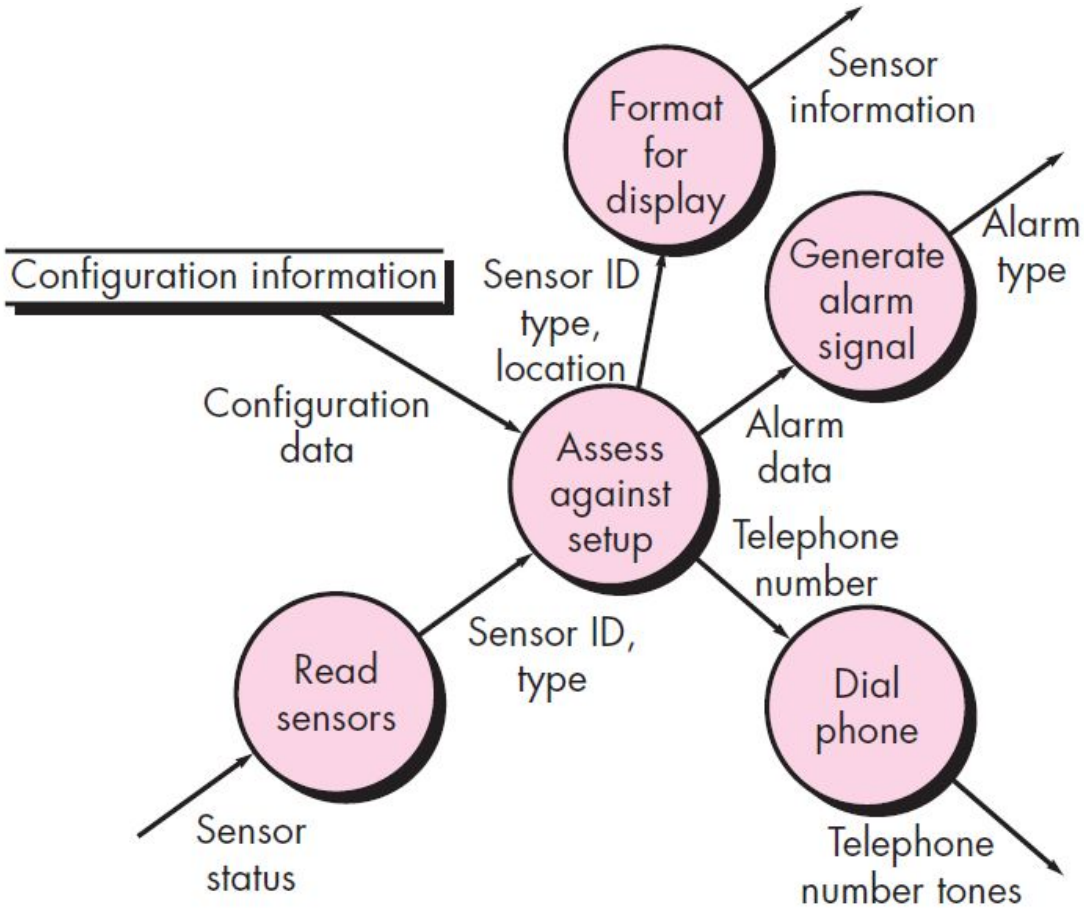
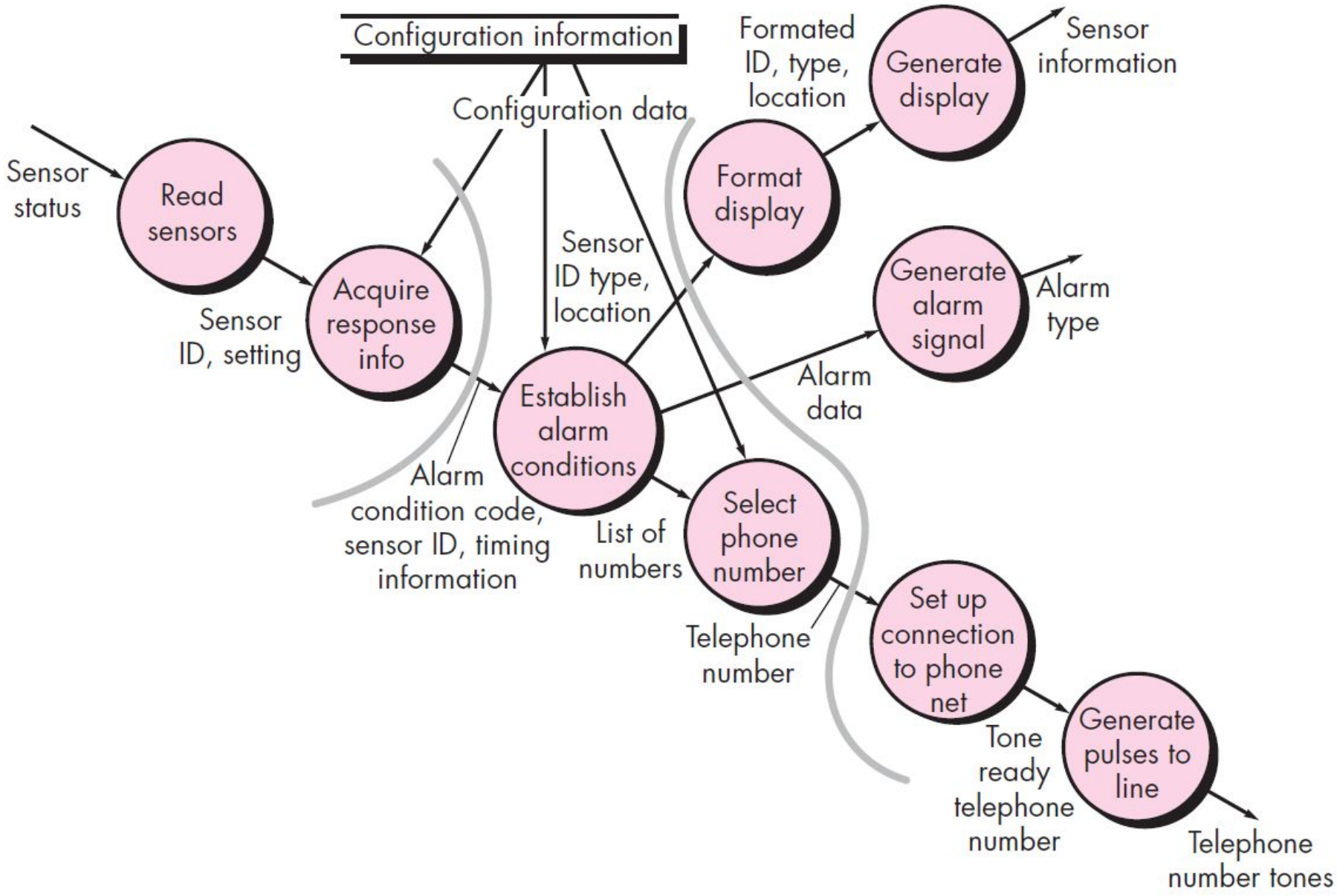


FIGURE 9.13 Level 3 DFD for *monitor sensors* with flow boundaries

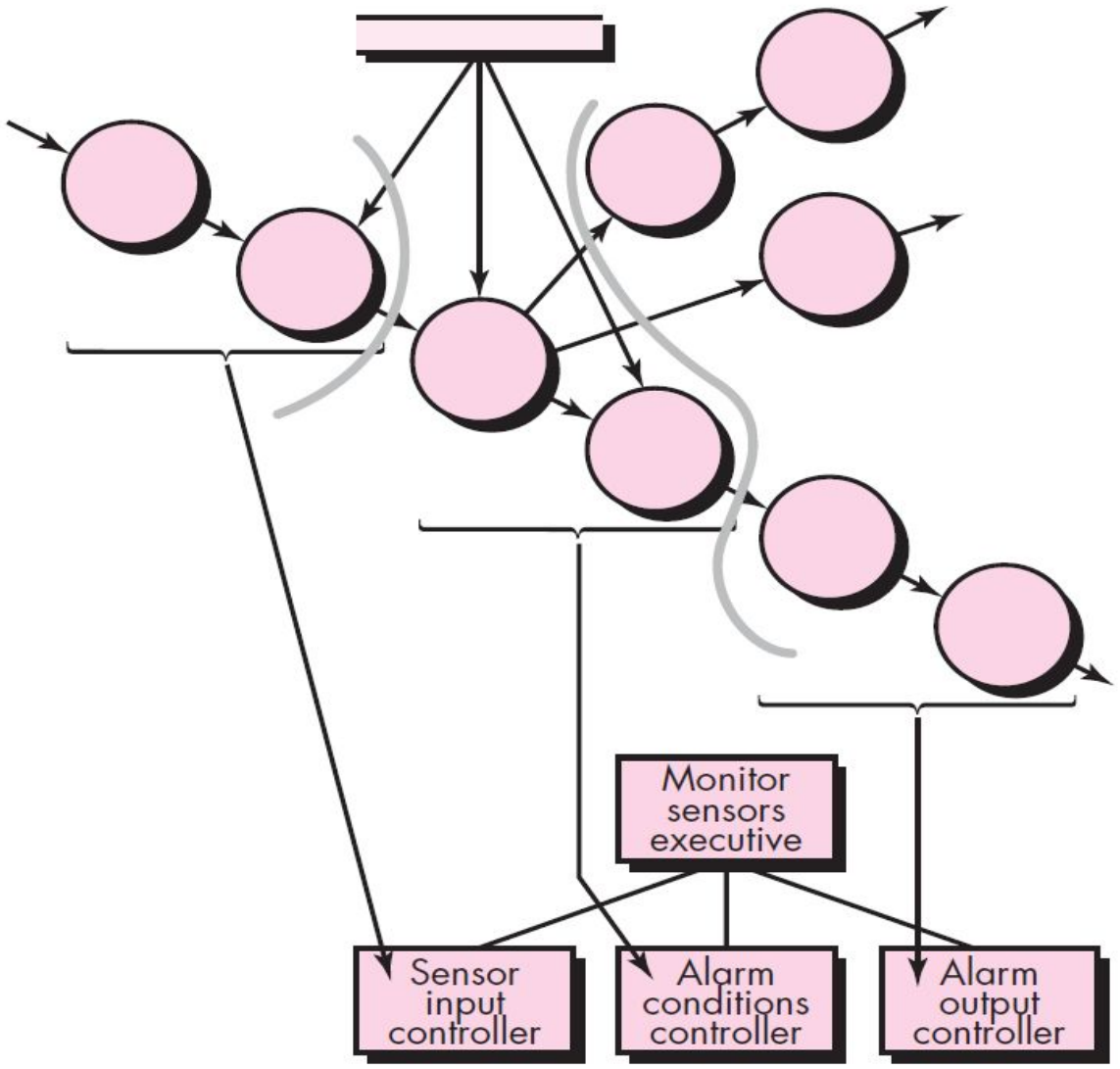


- **Step 3. Determine whether the DFD has transform or transaction flow characteristics.**
  - Evaluating the DFD (Figure 9.13), we see data entering the software along one incoming path and exiting along three outgoing paths.
  - Therefore, an overall transform characteristic will be assumed for information flow.
- **Step 4. Isolate the transform center by specifying incoming and outgoing flow boundaries.**
  - Incoming data flows along a path in which information is converted from external to internal form; outgoing flow converts internalized data to external form.
  - Incoming and outgoing flow boundaries are open to interpretation.

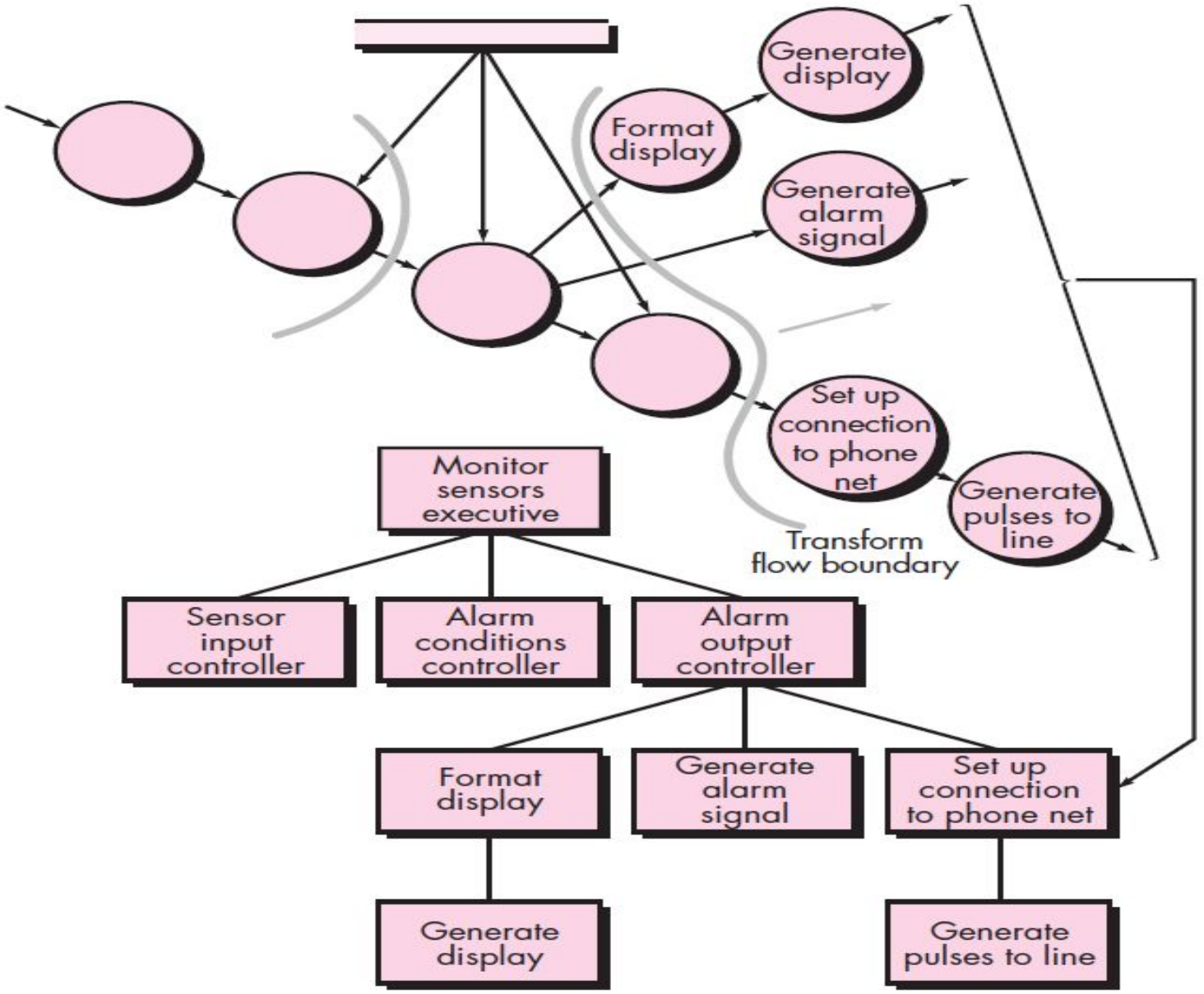
Step 5. Perform “first-level factoring.”

FIGURE 9.14

First-level factoring for *monitor sensors*

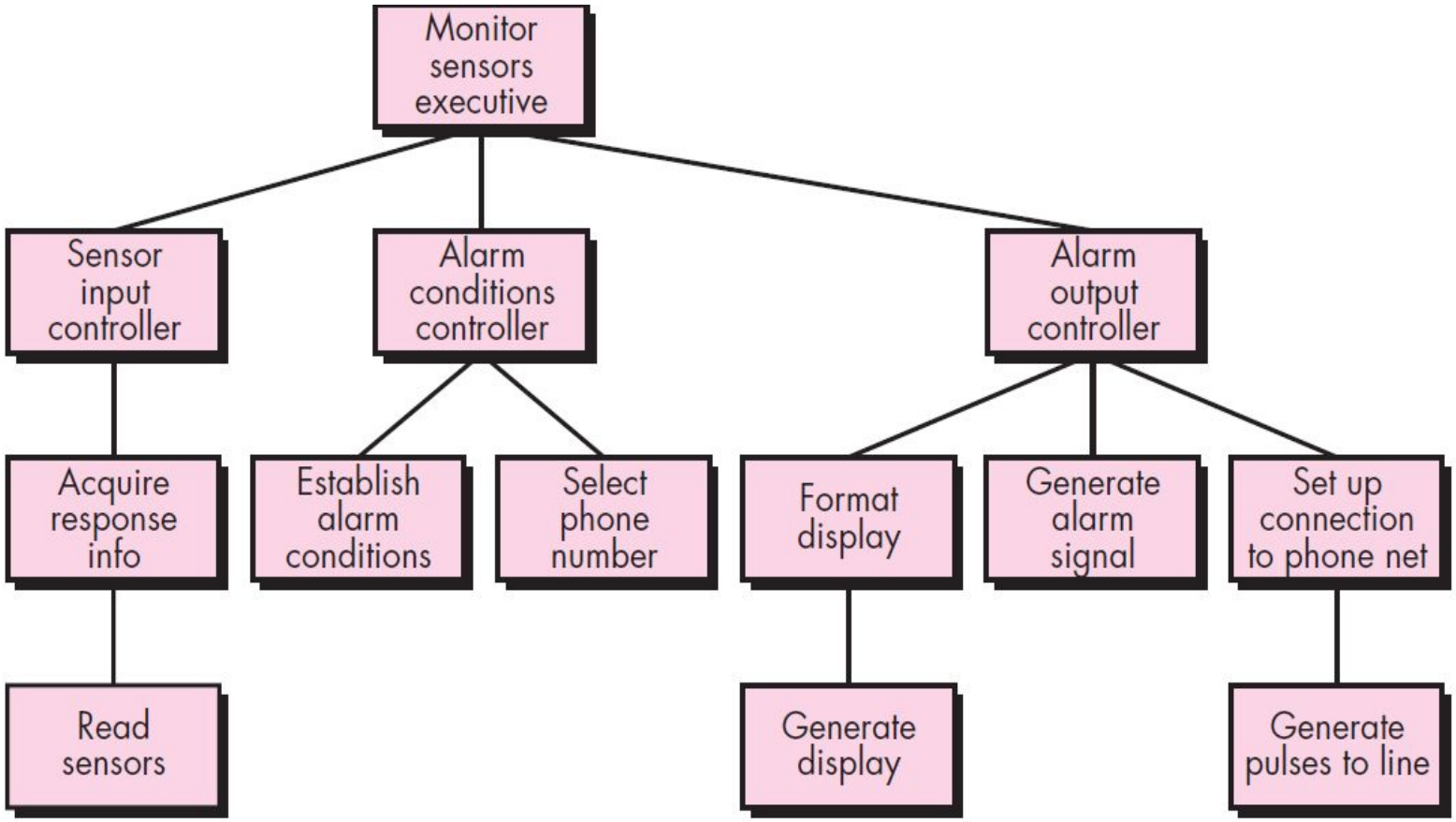


Step 6. Perform “second-level factoring.”



Step 7. Refine the first-iteration architecture using design heuristics for improved software quality.

First-iteration structure for *monitor sensors*



Quality Attributes

- *Functionality* is assessed by evaluating the feature set and capabilities of the program
- *Usability* is assessed by considering human factors, overall aesthetics, consistency, and documentation.
- *Reliability* is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure
- *Performance* is measured by considering processing speed, response time, resource consumption, throughput, and efficiency.
- *Supportability* combines the ability to extend the program (extensibility), adaptability, serviceability—these three attributes represent a more common term, *maintainability*