

Chapter 9

■ Architectural Design

Slide Set to accompany

Software Engineering: A Practitioner's Approach, 7/e
by Roger S. Pressman

Slides copyright © 1996, 2001, 2005, 2009 by Roger S. Pressman

For non-profit educational use only

May be reproduced ONLY for student use at the university level when used in conjunction with *Software Engineering: A Practitioner's Approach, 7/e*. Any other reproduction or use is prohibited without the express written permission of the author.

All copyright information MUST appear if these slides are posted on a website for student use.

Why Architecture?

The architecture is not the operational software. Rather, it is a representation that enables a software engineer to:

- (1) **analyze the effectiveness of the design** in meeting its stated requirements,
- (2) **consider architectural alternatives** at a stage when making design changes is still relatively easy, and
- (3) **reduce the risks** associated with the construction of the software.

Why is Architecture Important?

- **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 mode** of how the system is structured and how its components work together” [BAS03].

Architectural Descriptions

- The IEEE Computer Society has proposed IEEE-Std-1471-2000, *Recommended Practice for Architectural Description of Software-Intensive System*, [IEE00]
 - to establish a conceptual framework and vocabulary for use during the design of software architecture,
 - to provide detailed guidelines for representing an architectural description, and
 - to encourage sound architectural design practices.
- The IEEE Standard defines an *architectural description* (AD) as a “a collection of products to document an architecture.”
 - The description itself is represented using multiple views, where each *view* is “a representation of a whole system from the perspective of a related set of [stakeholder] concerns.”

Architectural Genres

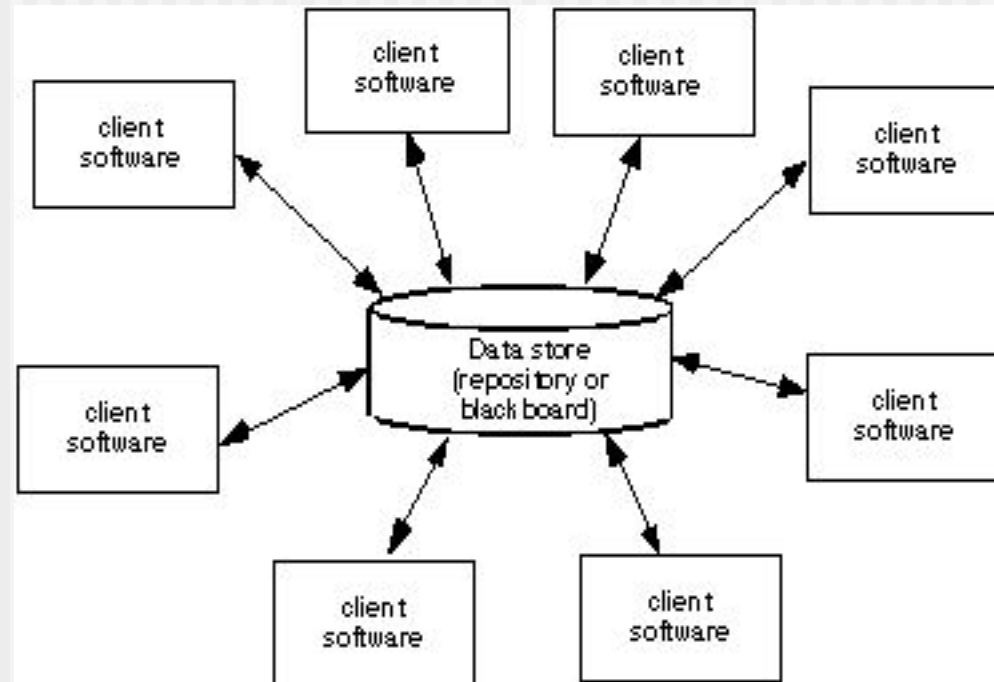
- *Genre* implies a specific category within the overall software domain.
- Within each category, you encounter a number of subcategories.
 - For example, within the genre of *buildings*, you would encounter the following general *styles*: houses, condos, apartment buildings, office buildings, industrial building, warehouses, and so on.
 - Within each general style, more specific styles might apply. Each style would have a structure that can be described using a set of predictable patterns.

Architectural Styles

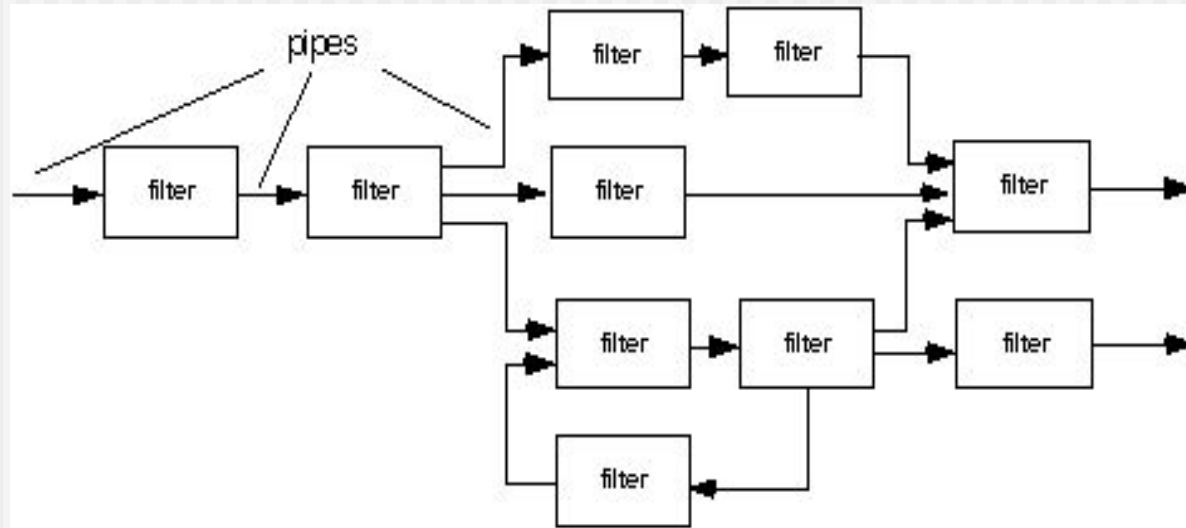
Each style describes a system category that encompasses: (1) a **set of components** (e.g., a database, computational modules) that perform a function required by a system, (2) a **set of connectors** that enable “communication, coordination and cooperation” among components, (3) **constraints** that 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.

- Data-centered architectures
- Data flow architectures
- Call and return architectures
- Object-oriented architectures
- Layered architectures

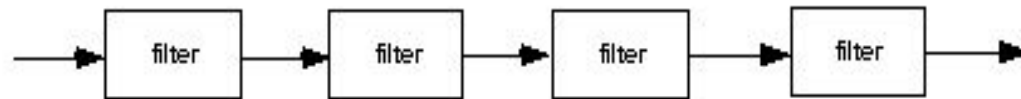
Data-Centered Architecture



Data Flow Architecture

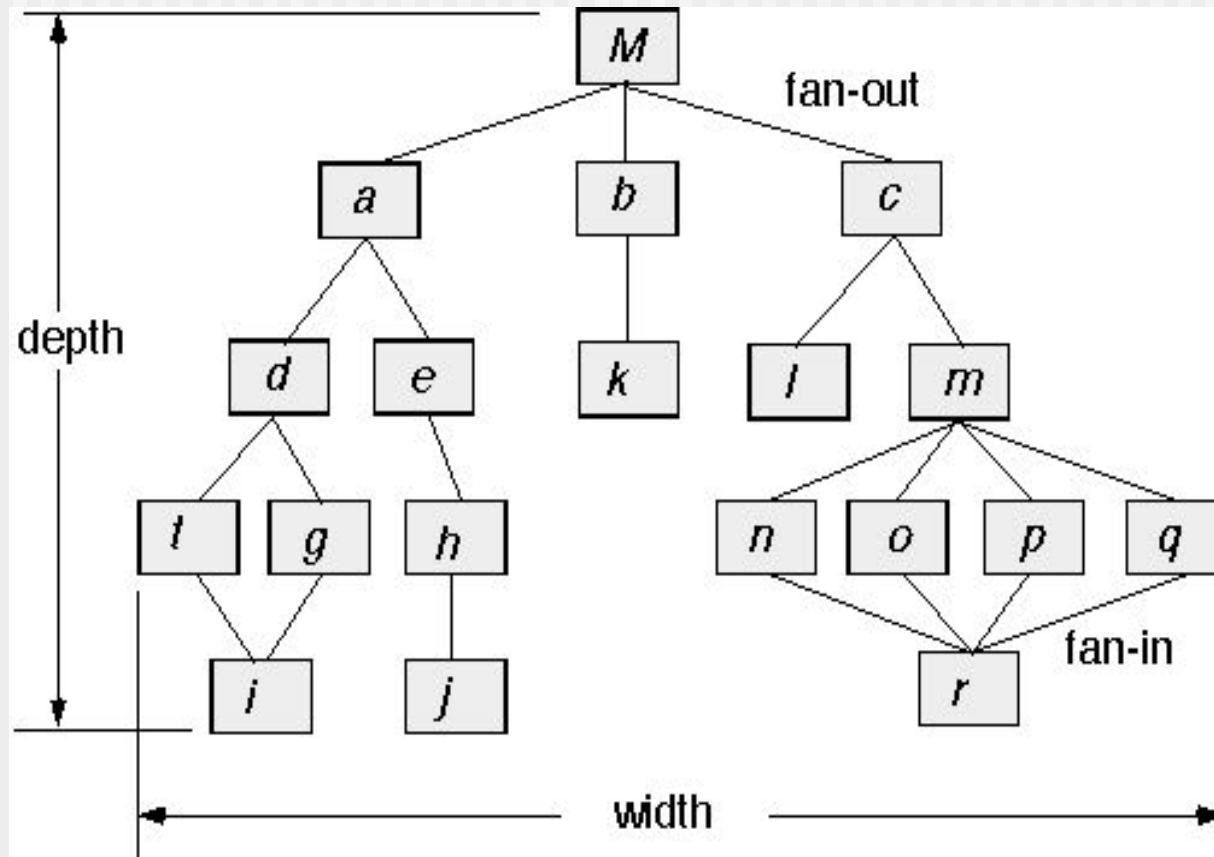


(a) pipes and filters

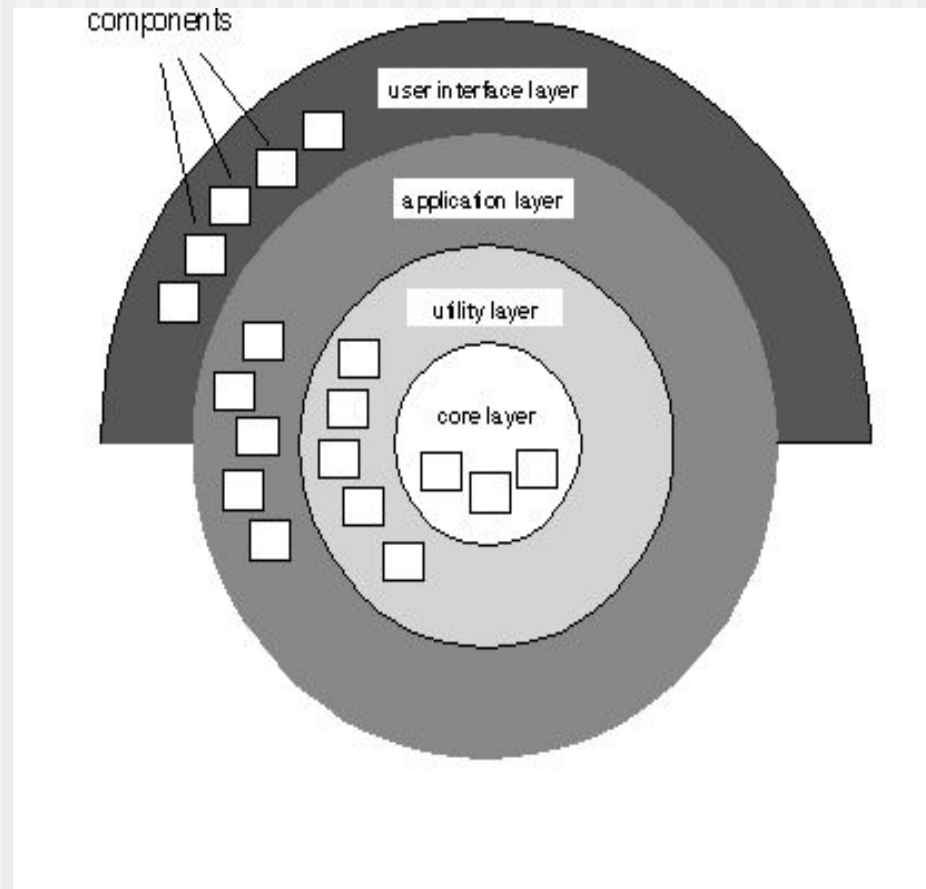


(b) batch sequential

Call and Return Architecture



Layered Architecture



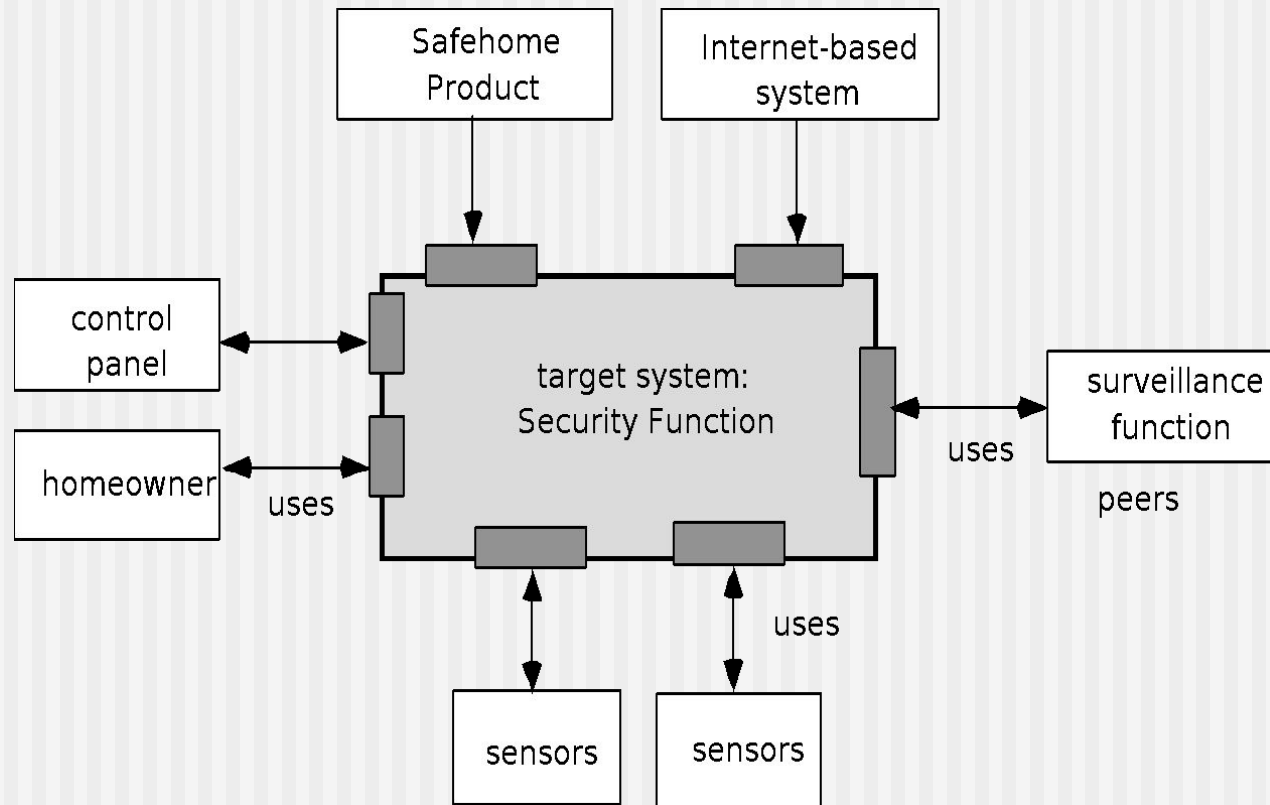
Architectural Patterns

- **Concurrency**—applications must handle multiple tasks in a manner that simulates parallelism
 - *operating system process management* pattern
 - *task scheduler* pattern
- **Persistence**—Data persists if it survives past the execution of the process that created it. Two patterns are common:
 - a *database management system* pattern that applies the storage and retrieval capability of a DBMS to the application architecture
 - an *application level persistence* pattern that builds persistence features into the application architecture
- **Distribution**— the manner in which systems or components within systems communicate with one another in a distributed environment
 - A *broker* acts as a ‘middle-man’ between the client component and a server component.

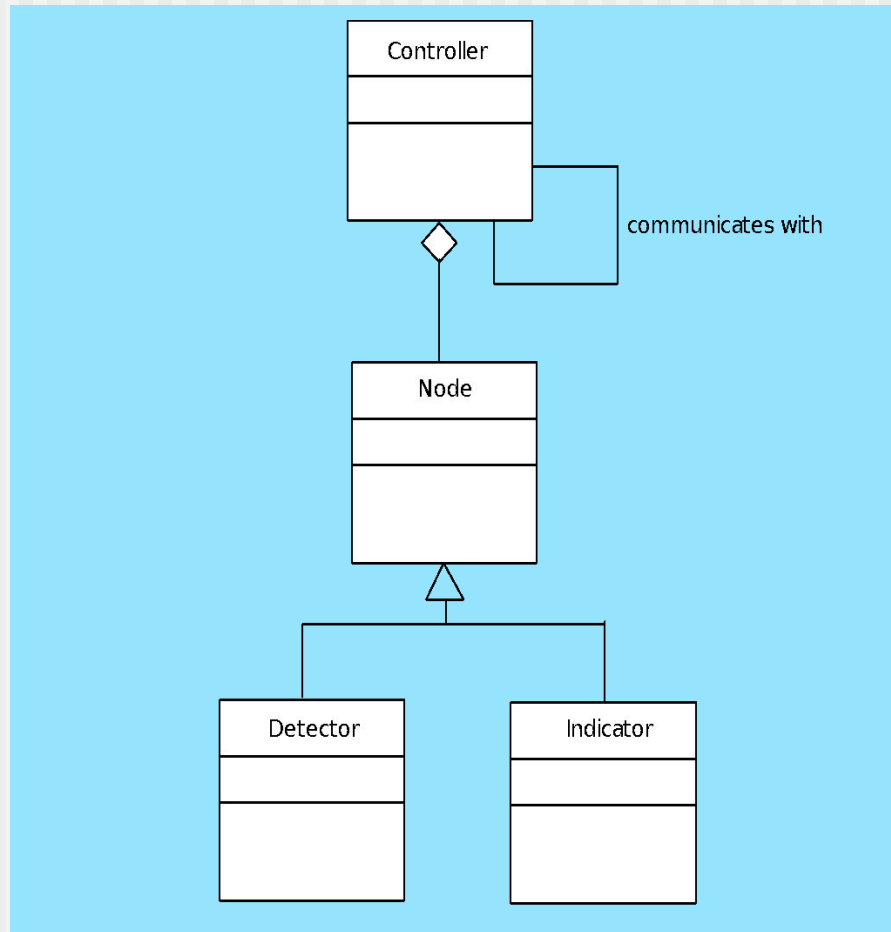
Architectural Design

- The software must be placed into context
 - the design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction
- A set of architectural archetypes should be identified
 - An *archetype* is an abstraction (similar to a class) that represents one element of system behavior
- The designer specifies the structure of the system by defining and refining software components that implement each archetype

Architectural Context

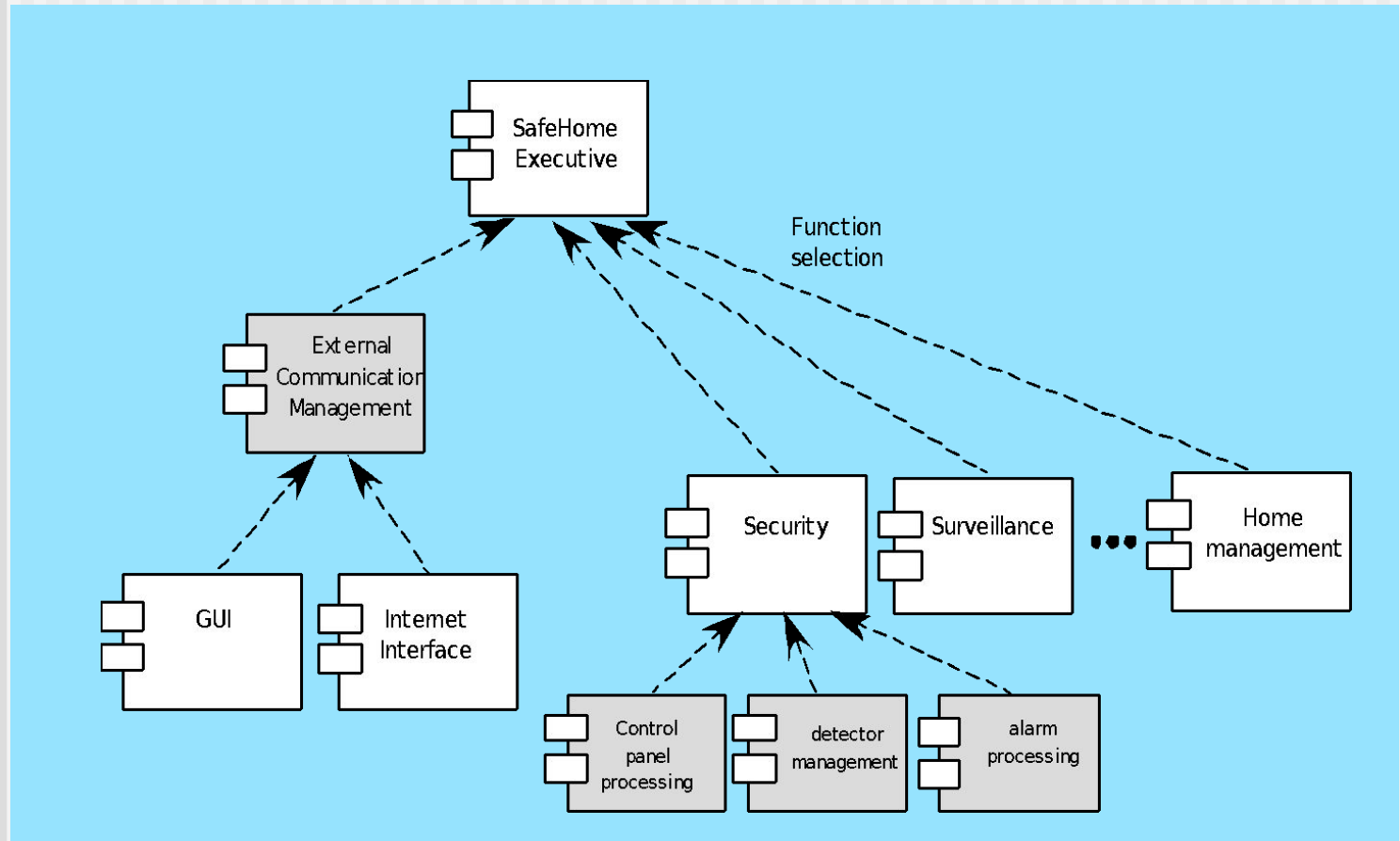


Archetypes

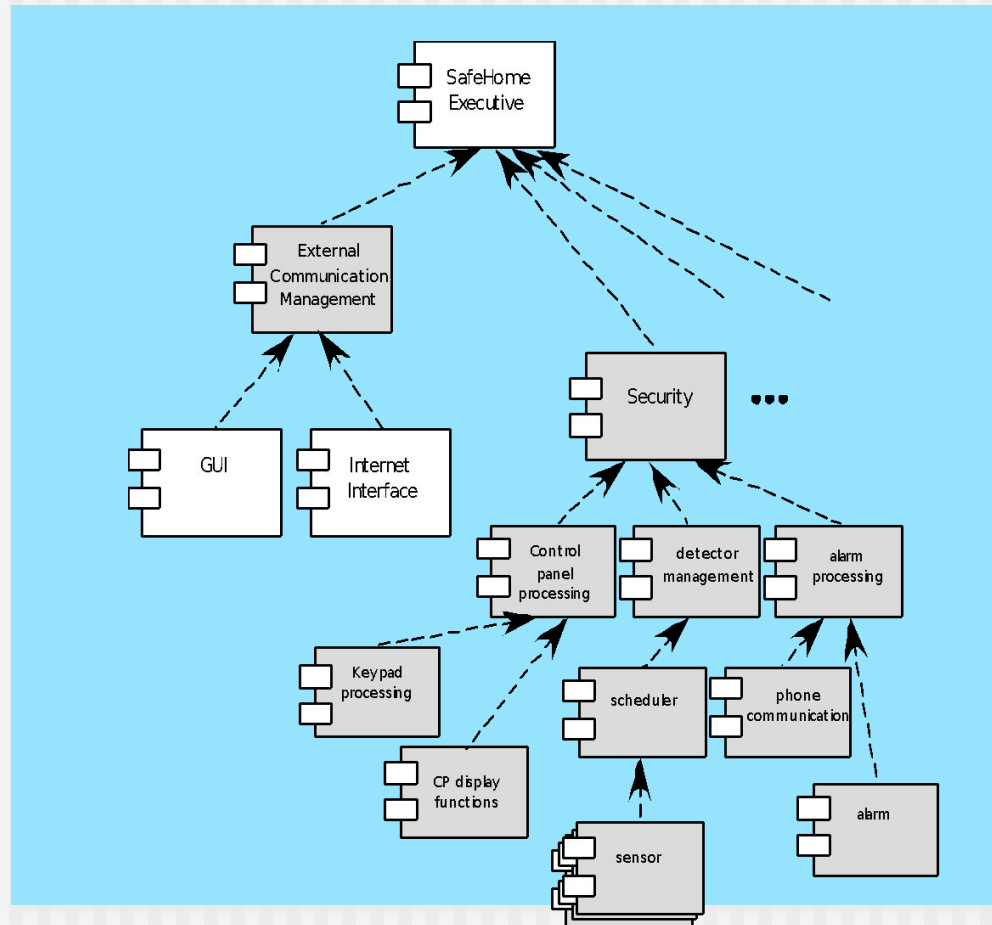


These slides are designed to accompany *Software Engineering for Safe Home Security Function, archetypes* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Component Structure



Refined Component Structure



These slides are designed to accompany *Software Engineering: A Practitioner's Approach*, 7/e (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Analyzing Architectural Design

1. Collect scenarios.
2. Elicit requirements, constraints, and environment description.
3. Describe the architectural styles/patterns that have been chosen to address the scenarios and requirements:
 - module view
 - process view
 - data flow view
4. Evaluate quality attributes by considered each attribute in isolation.
5. Identify the sensitivity of quality attributes to various architectural attributes for a specific architectural style.
6. Critique candidate architectures (developed in step 3) using the sensitivity analysis conducted in step 5.

Architectural Complexity

- the overall complexity of a proposed architecture is assessed by considering the **dependencies** between components within the architecture [Zha98]
 - *Sharing dependencies* represent dependence relationships among consumers who use the same resource or producers who produce for the same consumers.
 - *Flow dependencies* represent dependence relationships between producers and consumers of resources.
 - *Constrained dependencies* represent constraints on the relative flow of control among a set of activities.

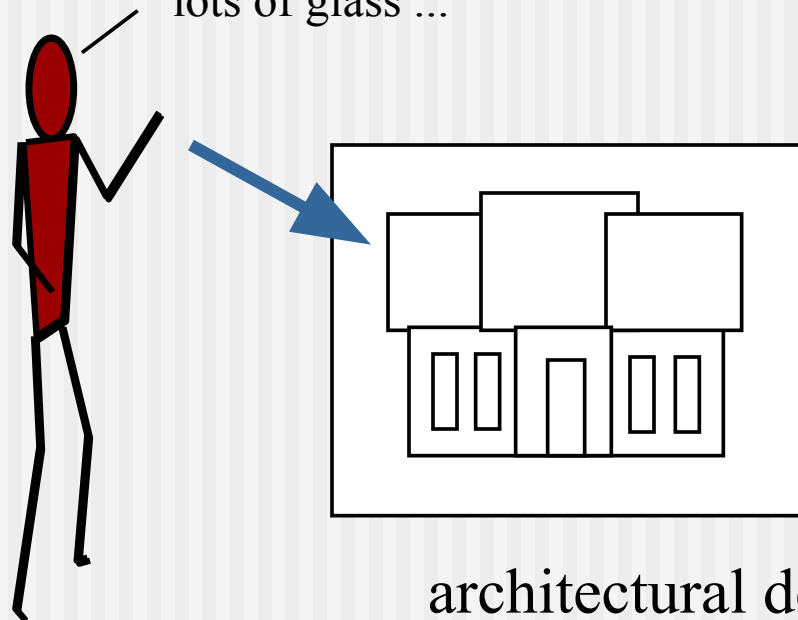
ADL

- *Architectural description language (ADL)* provides a semantics and syntax for describing a software architecture
- Provide the designer with the ability to:
 - decompose architectural components
 - compose individual components into larger architectural blocks and
 - represent interfaces (connection mechanisms) between components.

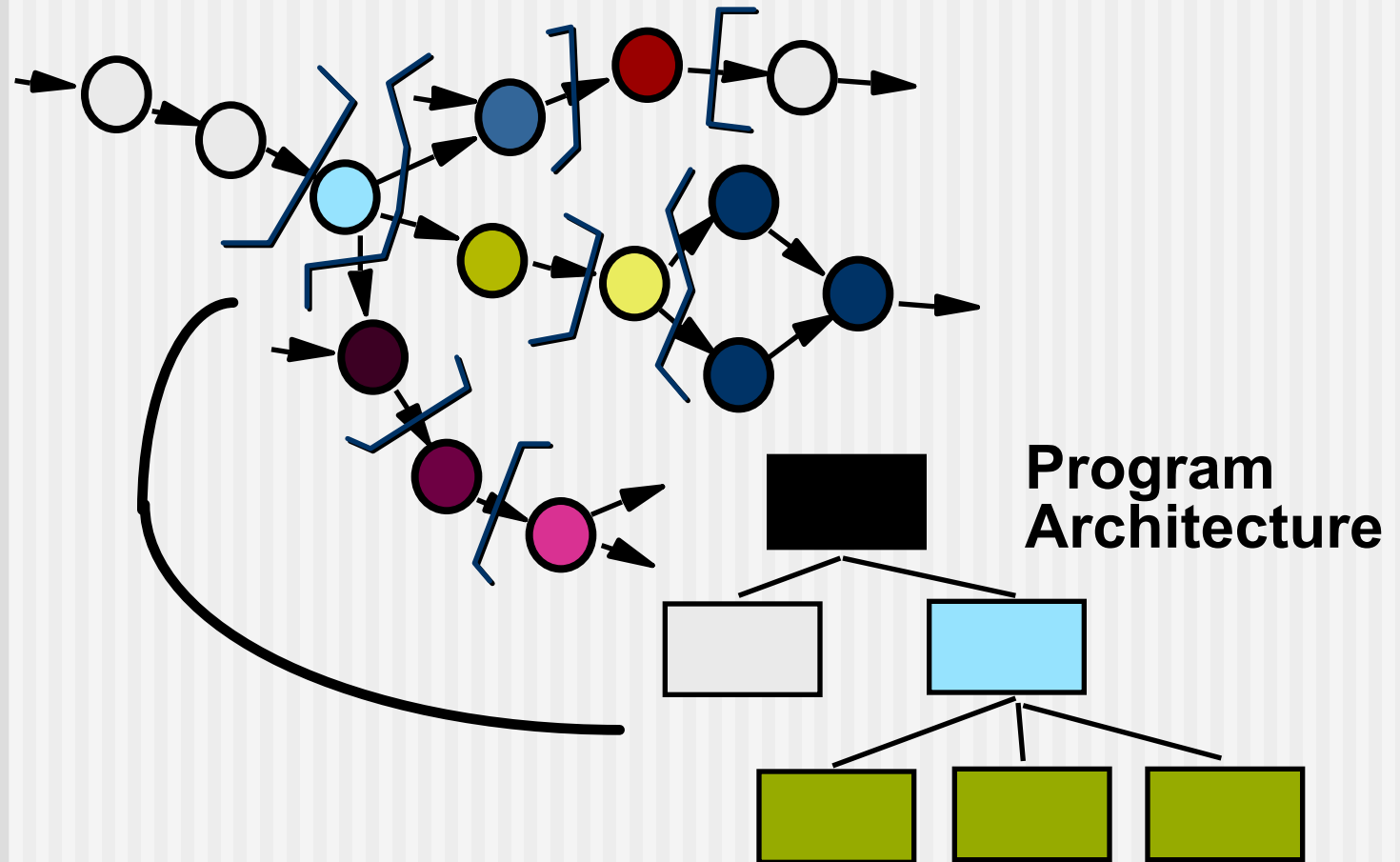
An Architectural Design Method

customer requirements

"four bedrooms, three baths,
lots of glass ..."

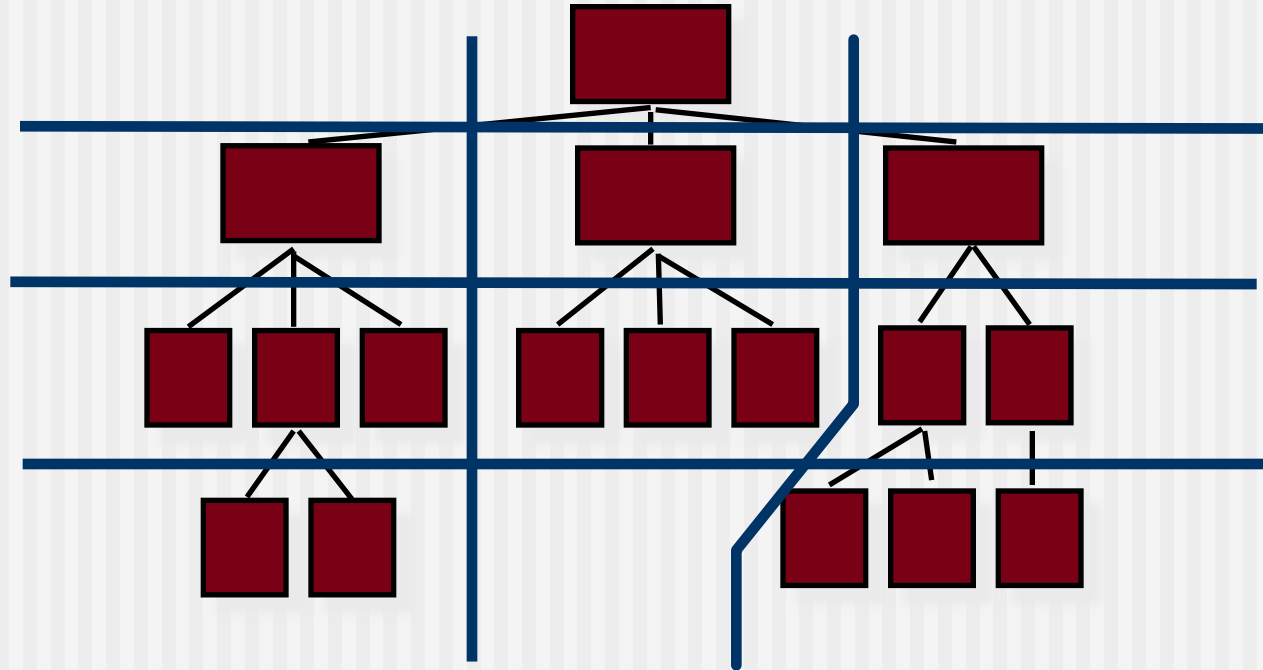


Deriving Program Architecture



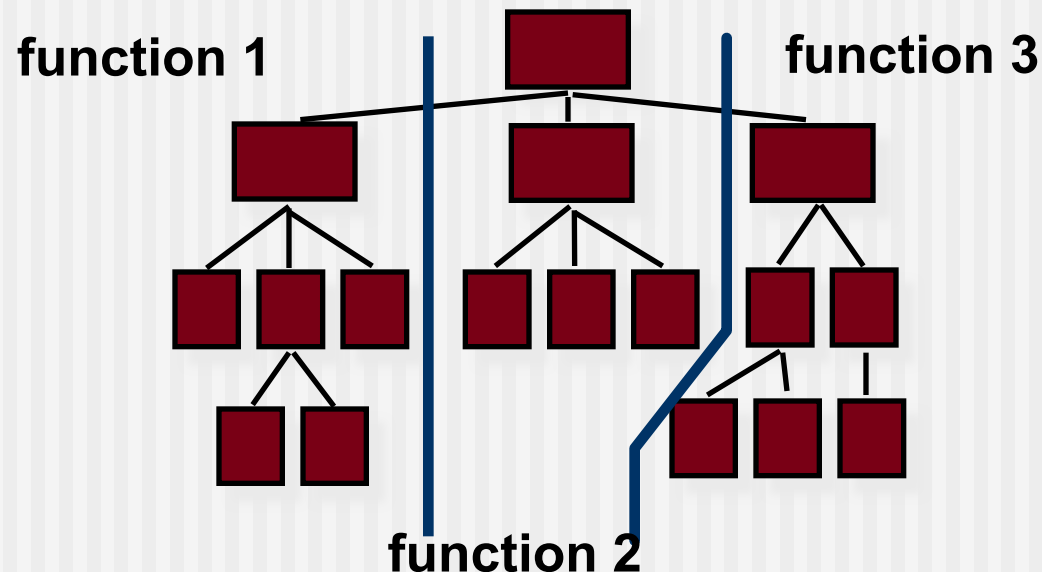
Partitioning the Architecture

- “horizontal” and “vertical” partitioning are required



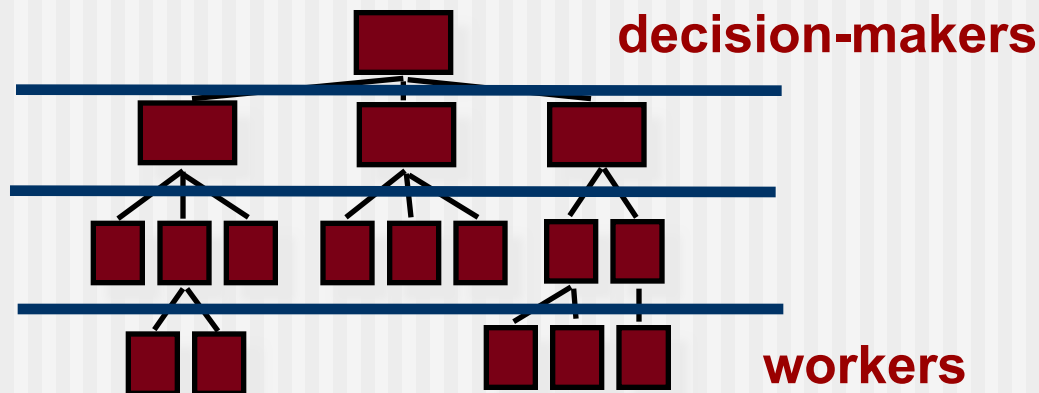
Vertical Partitioning

- define separate branches of the module hierarchy for each major function
- use control modules to coordinate communication between functions



Horizontal Partitioning: Factoring

- design so that decision making and work are stratified
- decision making modules should reside at the top of the architecture



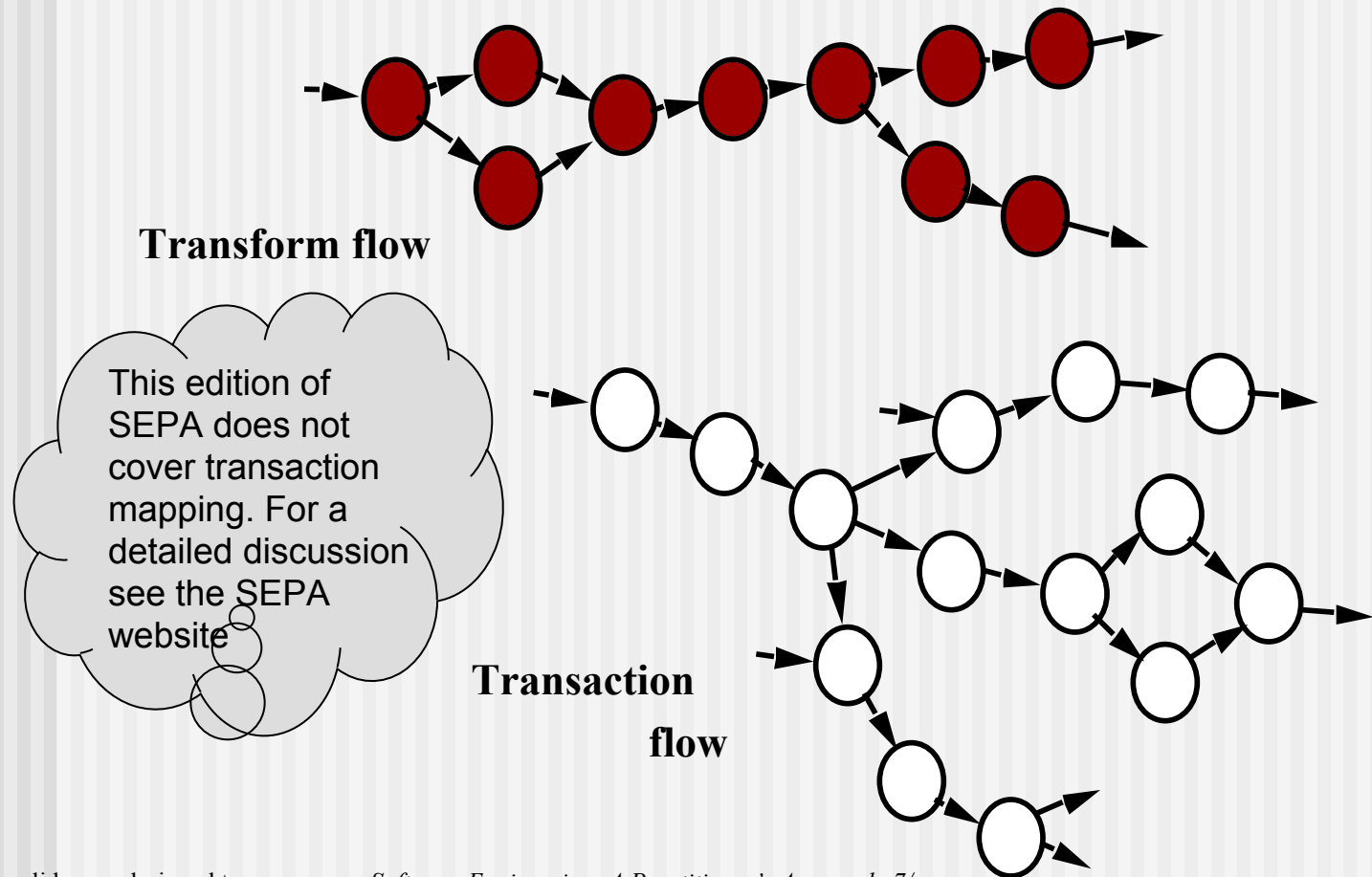
Why Partitioned Architecture?

- results in software that is easier to test
- leads to software that is easier to maintain
- results in propagation of fewer side effects
- results in software that is easier to extend





Structured Design

- **objective:** to derive a program architecture that is partitioned
- **approach:**
 - a DFD is mapped into a program architecture
 - the PSPEC and STD are used to indicate the content of each module
- **notation:** structure chart

Flow Characteristics



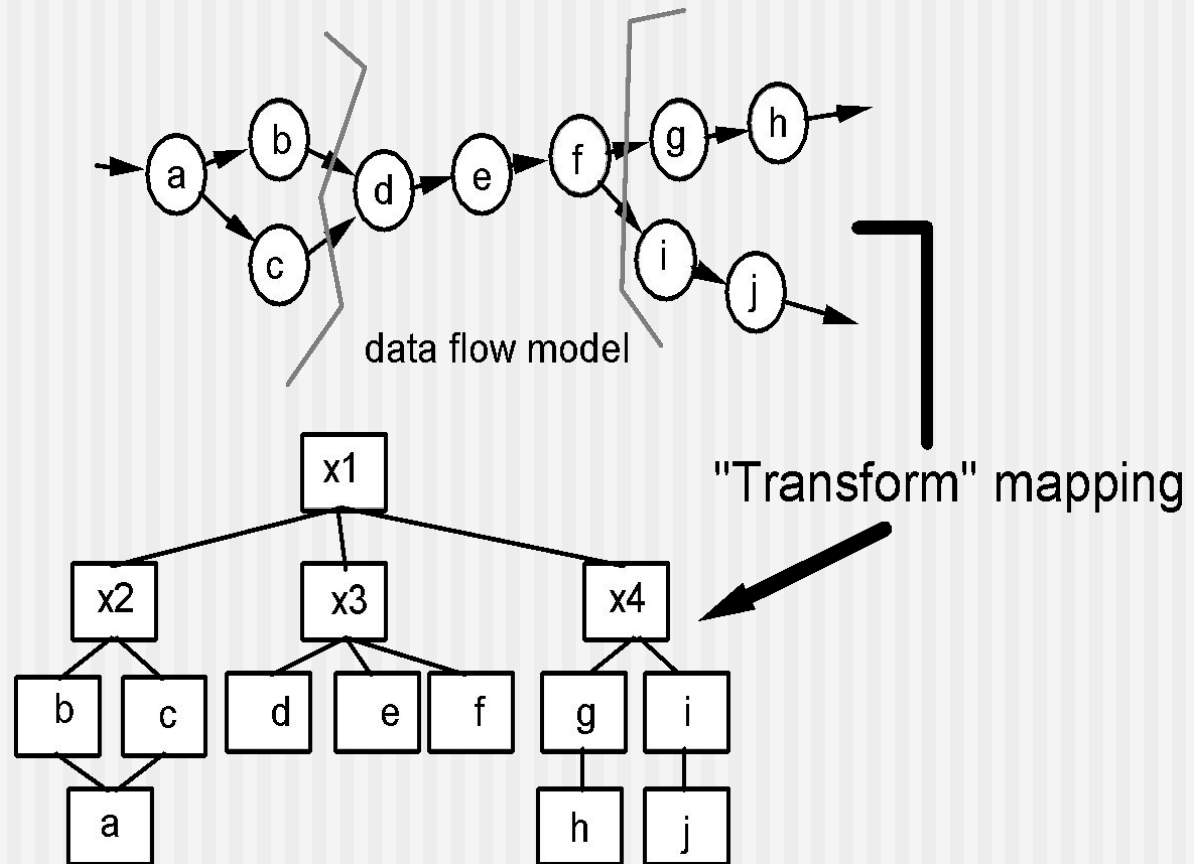
General Mapping Approach

-  isolate incoming and outgoing flow boundaries; for transaction flows, isolate the transaction center
-  working from the boundary outward, map DFD transforms into corresponding modules
-  add control modules as required
-  refine the resultant program structure using effective modularity concepts

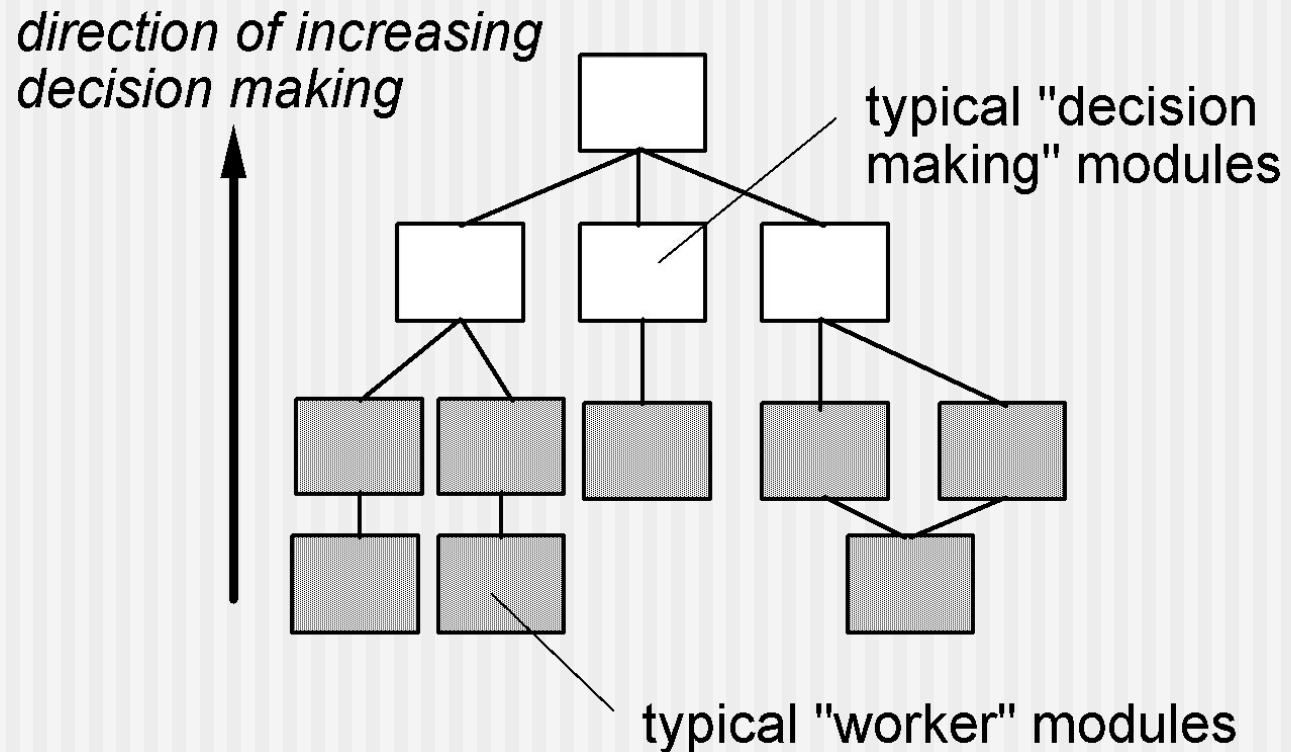
General Mapping Approach

- **Isolate the transform center by specifying incoming and outgoing flow boundaries**
- **Perform "first-level factoring."**
 - The program architecture derived using this mapping results in a top-down distribution of control.
 - *Factoring* leads to a program structure in which top-level components perform decision-making and low-level components perform most input, computation, and output work.
 - Middle-level components perform some control and do moderate amounts of work.
- **Perform "second-level factoring."**

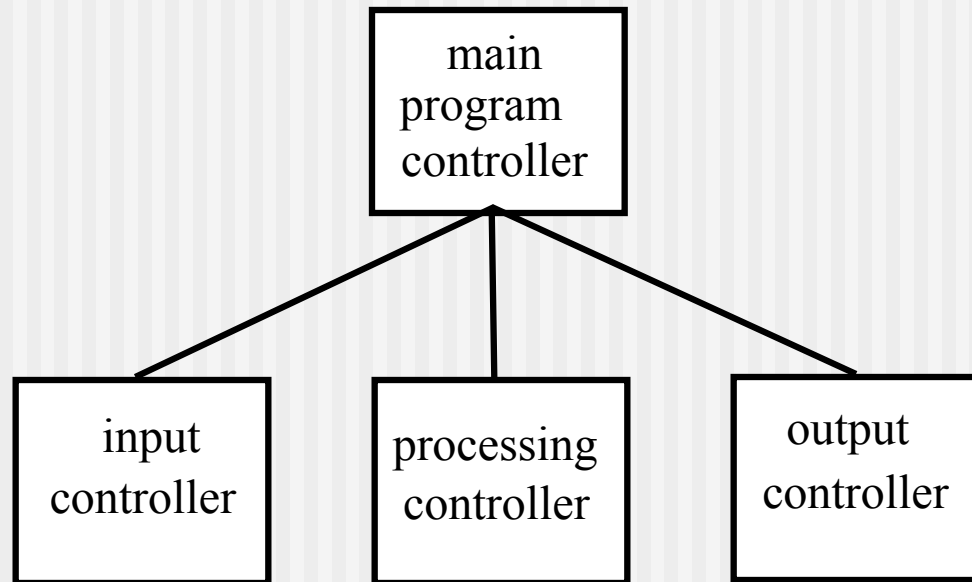
Transform Mapping



Factoring



First Level Factoring



Second Level Mapping

