Chapter 13

- **Architectural Design**

Slide Set to accompany

Software Engineering: A Practitioner’s Approach

by Roger S. Pressman

Software Architecture - Notion

- “The overall structure of the software and the ways in which that structure provides conceptual integrity for a system.” [SHA95a]
- “... the set of structures needed to reason about the software system, which comprise the software elements, the relations between them, and the properties of both elements and relations.”

  Clements, Paul; Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, Robert Nord, Judith Stafford (2010). *Documenting Software Architectures: Views and Beyond, Second Edition*. Boston: Addison-Wesley.

☛ "An architecture is the set of significant decisions about the organization of a software system, the selection of the structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaborations among those elements, the composition of these structural and behavioral elements into progressively larger subsystems, and the architectural style that guides this organization - these elements and their interfaces, their collaborations, and their composition." [Jacobson 99][Kruchten 01].

Software Architecture - Definition

"Software architecture is a set of important decisions on
the structure of a software system and
the whole development process
so that the software system has
the required functions and qualities and also
the buildability to facilitate construction of the software system and
the evolvability to sustain maintenance and improvement."
[Kang 05]

[Kang 05] Sungwon Kang, *Invitation to Software Architecture – Fundamental Principles of Software Architecture Design*, HongRung Publishing Company, Revised Edition, 2015.

Table of Contents

13.1Software Architecture
13.1.1 What is 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.

☛This explains only part of
the notion of architecture
but explains its important roles.

13.1.2 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 model of how the system is structured and how its components work together” [BAS03].

Architecture also allows division of work.

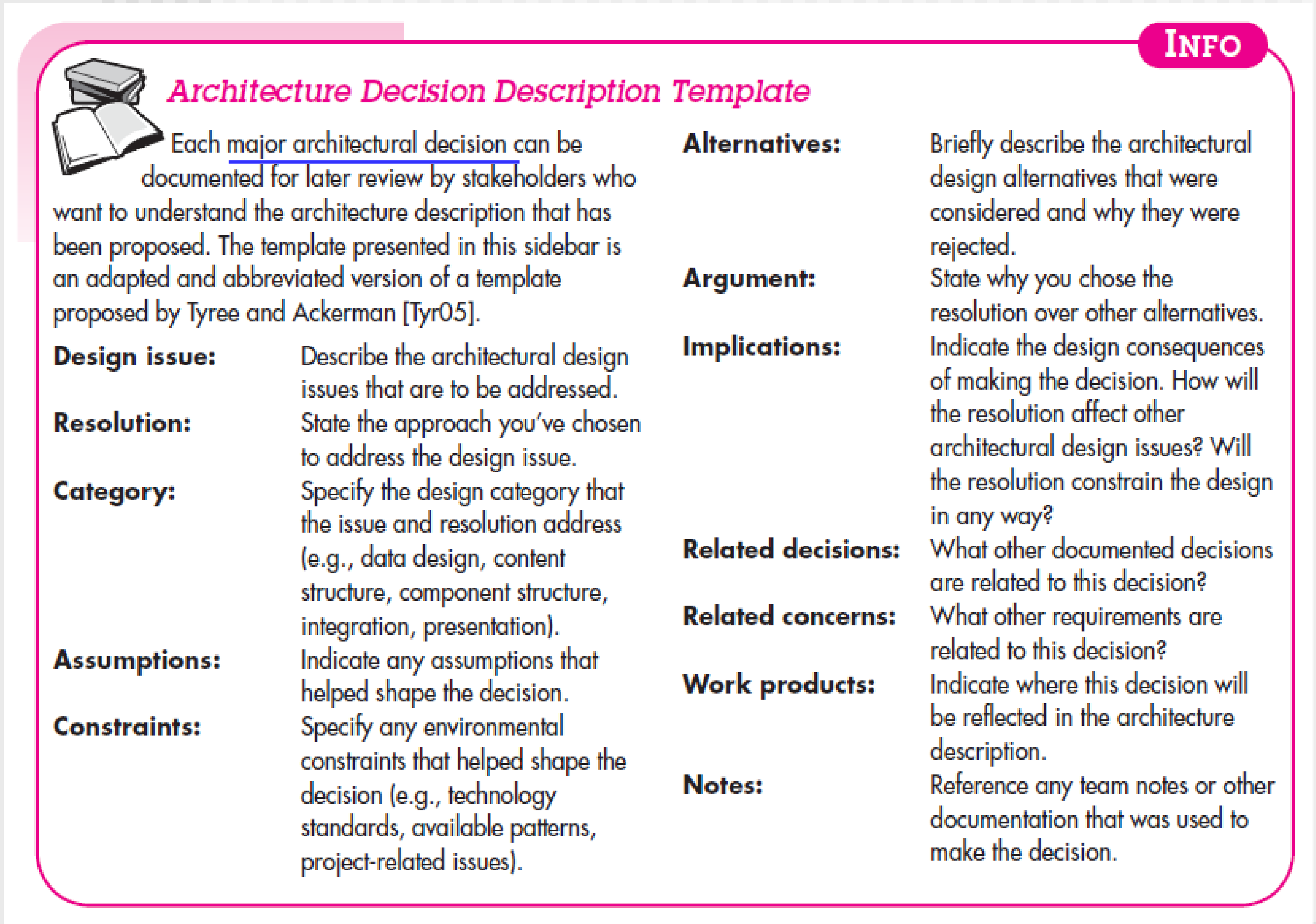
13.1.3 Architectural Descriptions

- 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
  - to encourage sound architectural design practices.
- The IEEE Standard defines an *architectural description* (AD) as a "a collection of products to document an architecture."
  - using multiple views, where each *view* is "a representation of a whole system from the perspective of a related set of [stakeholder] concerns."

Architecture Viewpoint and Architecture View

- Conceptual Viewpoint -> Conceptual View
- Logical Viewpoint -> Logical View
  (= Component and Connector Viewpoint) (= Component and Connector View)
- Module Viewpoint -> Module View
- Runtime Viewpoint -> Runtime View
  (= Execution View) (= Execution Viewpoint)
- Deployment Viewpoint -> Deployment View
- . . .

13.1.4 Architectural Decisions



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.

13.2 Architectural Genres(Nonstandard)

- *Genre* implies a specific category within the overall software domain.
- Within each category, you encounter a number of subcategories.
  - E.g., within the genre of *buildings*, the following general styles exist: 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.

13.3 Architectural Styles

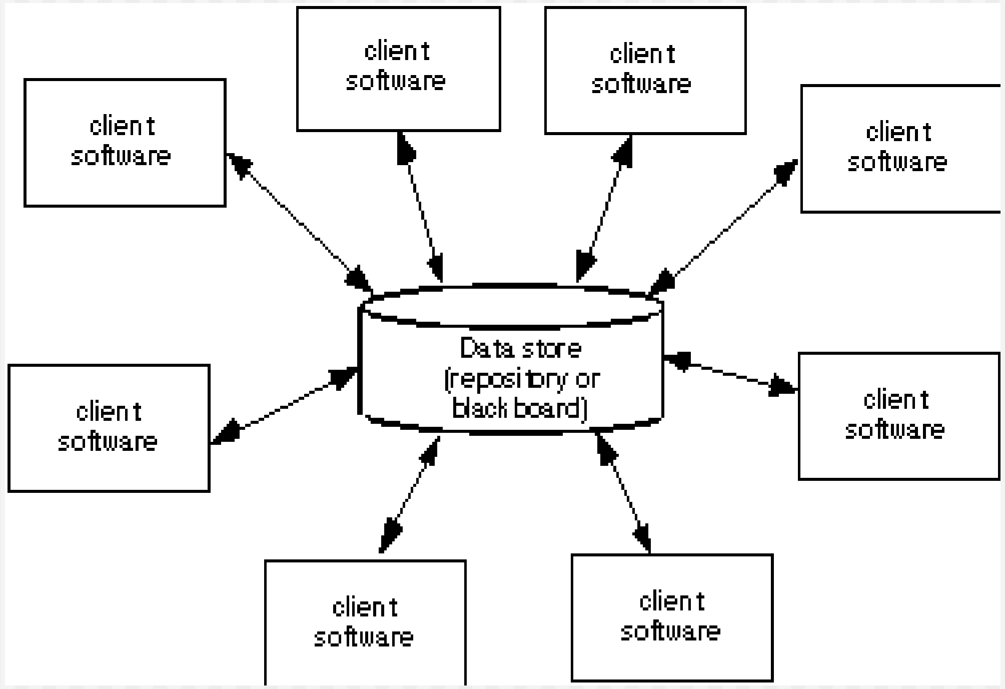
[Shaw 96] defined architecture style as follows.
"… More specifically, an architectural style defines
a vocabulary of components and connector types, and
a set of constraints on how they can be combined …"

Each style describes :
(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
(4) semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

13.3.1 A Brief Taxonomy of Architectural Styles

- Data-centered architecture style
- Data flow architecture style
- Call and return architecture style
- Object-oriented architecture style
- Layered architecture style
- . . .

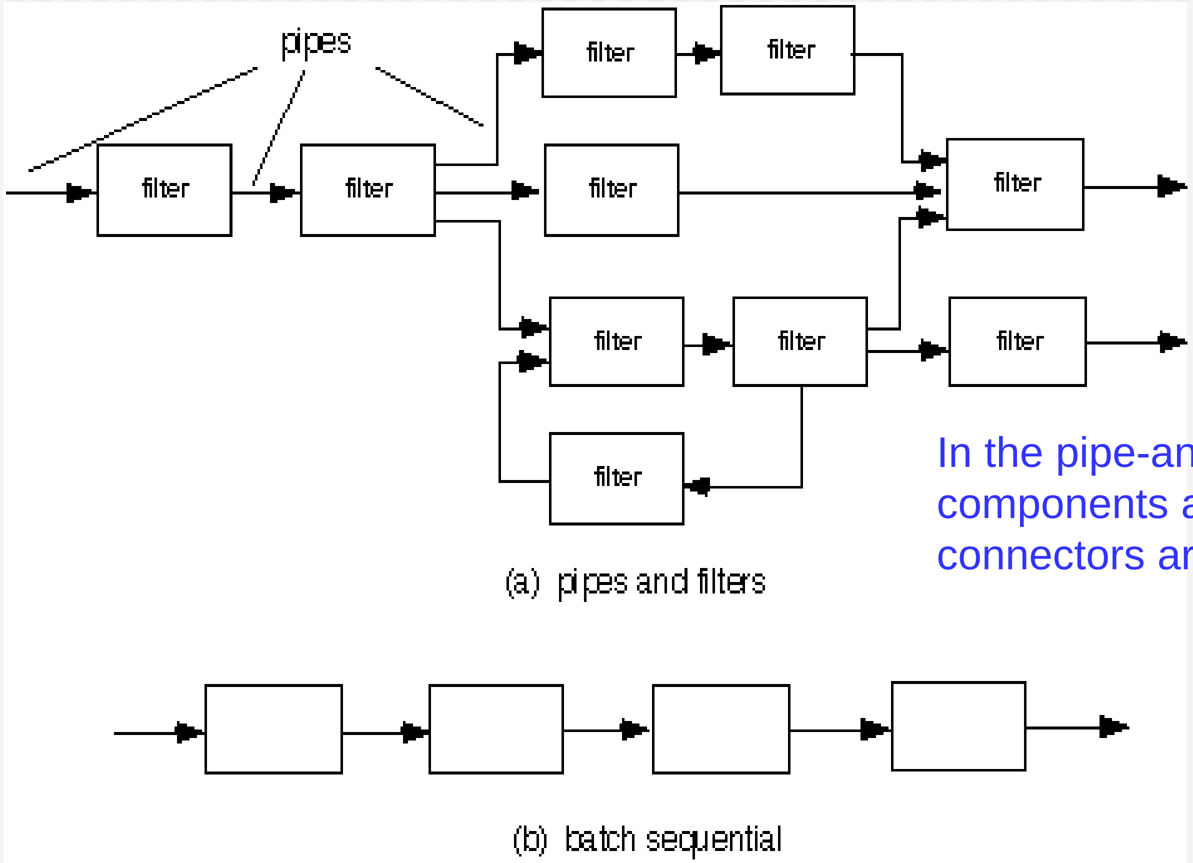
Data-Centered Architecture Style



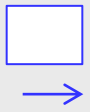
Legend

▭	: component
⛁	: data component
↔	: connector

Data Flow Architecture Style



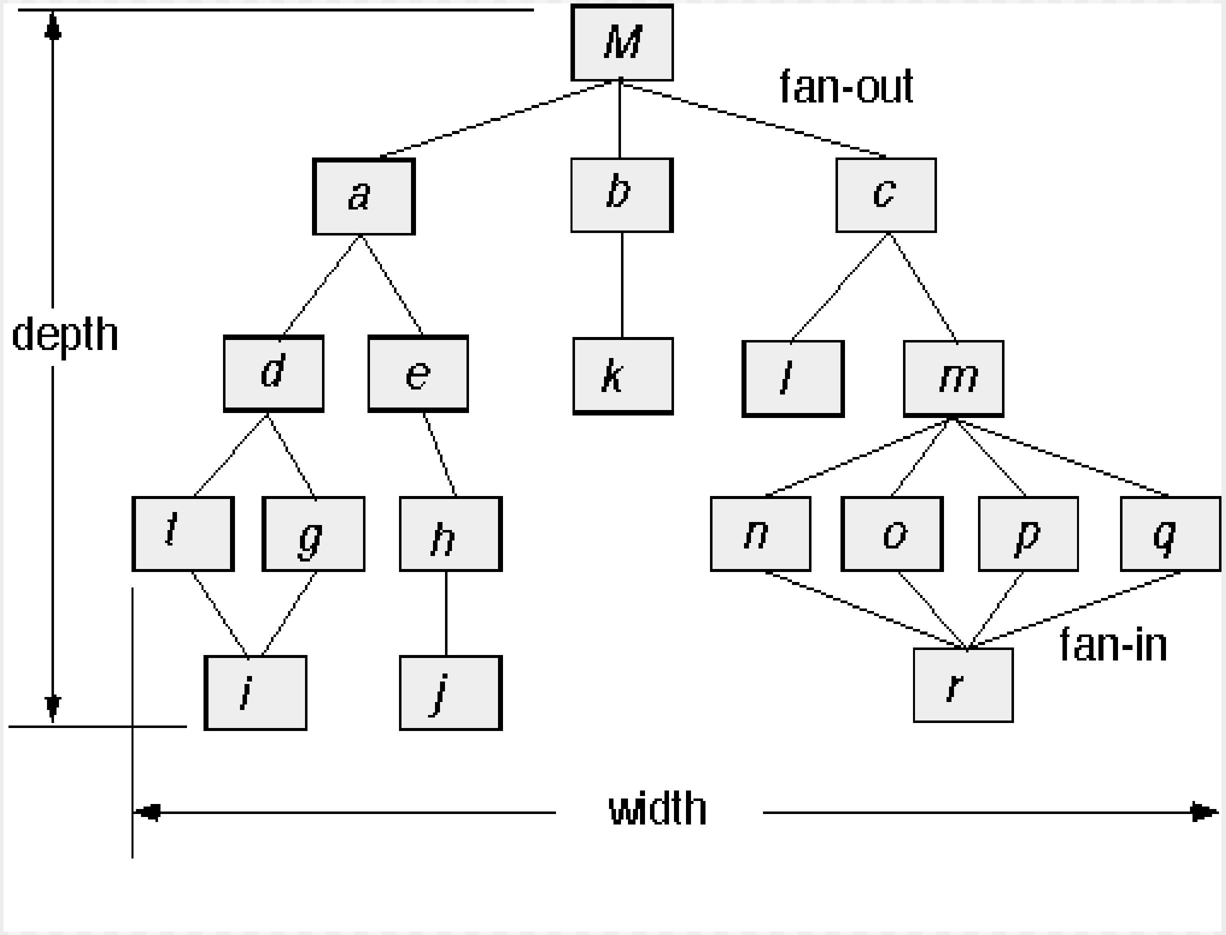
Legend


: component

: connector

In the pipe-and-filter style, components are filters and connectors are pipes.

Call and Return Architecture Style

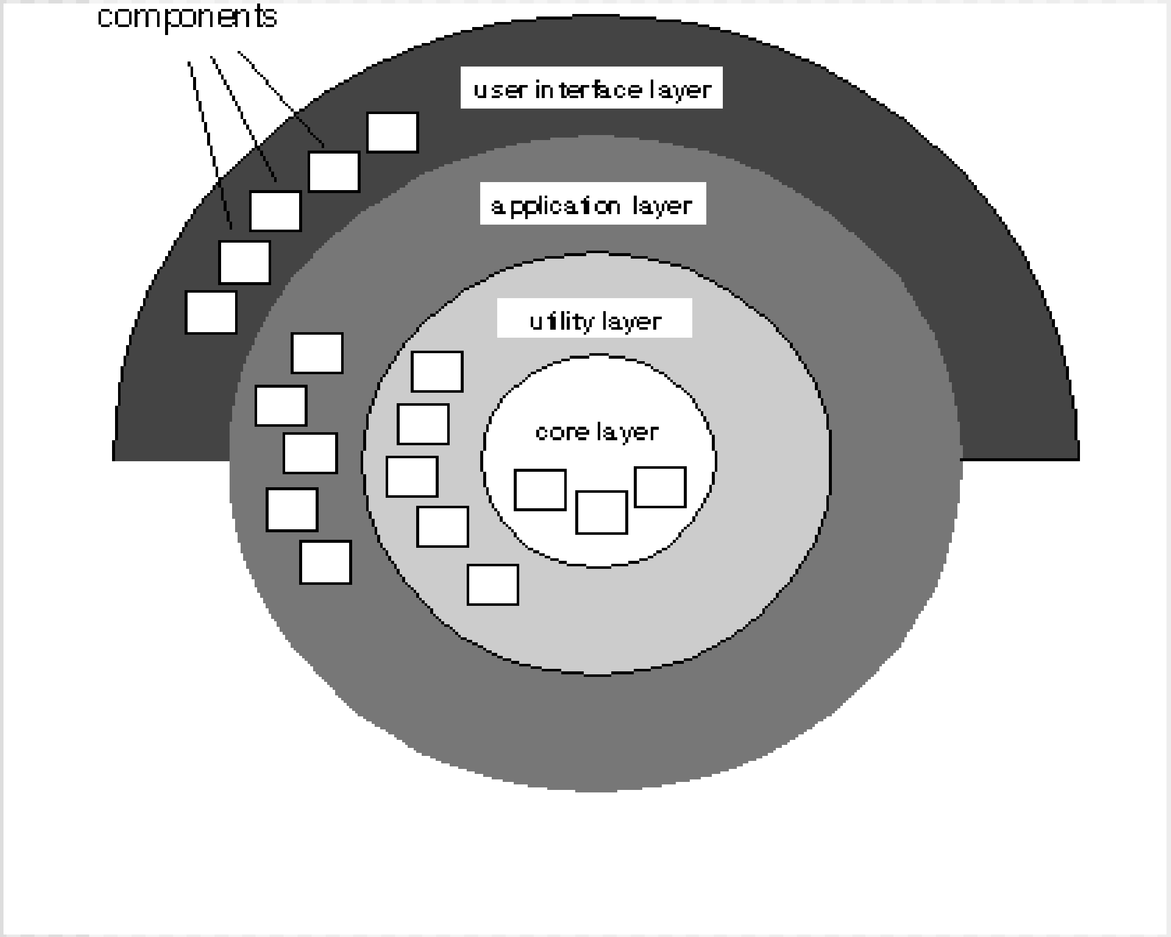


Legend

▭ : component

— : connector

Layered Architecture Style



13.3.2 Architectural Patterns(1/3)

- Definition : An architectural pattern is a named collection of architectural design decisions that are applicable to a recurring design problem parameterized to account for different software development contexts in which that problem appears. [Taylor 09]

To solve the same problem **using architectural style** requires more attention from the system's architect, and provides less direct support.

13.3.2 Architectural Patterns(2/3)

- Concurrency—applications must handle multiple tasks in a parallel manner
  - Operating system process management pattern
  - Task scheduler pattern
- Persistence—Data persists if it survives past the execution of the process that created it.
  - Database management system pattern
    - applies the storage and retrieval capability of a DBMS to the application architecture
  - Application level persistence pattern
    - 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
  - Broker pattern
    - There is a ‘middle-man’ between the client component and a server component.
- . . .

13.3.2 Architectural Patterns(3/3)

Architecture Patterns for Distributed Computing

Category	Pattern
From Mud to Structure	Domain Model, Layers, Model-View-Controller, Presentation-Abstraction-Control, Microkernel, Reflection, Pipes and Filters, Shared Repository, Blackboard, Domain Object,
Distribution Infrastructure	Messaging, Message Channel, Message Endpoint, Message Translator, Message Router, Publisher-Subscriber, Broker, Client Proxy, Requestor, Invoker, Client Request Handler, Server Request Handler
Event Demultiplexing and Dispatching	Reactor, Proactor, Acceptor-Connector, Asynchronous Completion Token
Interface Partitioning	Explicit Interface, Extension Interface, Introspective Interface, Dynamic Invocation Interface, Proxy, Business Delegate, Facade, Combined Method, Iterator , Enumeration Method, Batch Method
Component Partitioning	Encapsulated Implementation, Whole-Part, Composite, Master-Slave, Half-Object plus P:rotocol, Replicated Component Group
Application Control	Page Controller, Front Controller, Application controller, command Processor, Template View, Transform View, Firewall Proxy, authorization

Source: [Buschmann 07a]

13.3.3 (Assessing) Organization and (Further) Refinement

- Because the design process often leaves you with a number of architectural alternatives, it is important to establish a set of design criteria that can be used to assess an architectural design that is derived.
- The following questions [Bas03] provide insight into an architectural style:
  - How is control managed within the architecture?
  - How are data communicated between components?

13.4 Architectural Considerations

- Economy – The best software is uncluttered and relies on abstraction to reduce unnecessary detail.
- Visibility – Architectural decisions and the reasons for them should be obvious to software engineers who examine the model at a later time.
- Spacing – Separation of concerns in a design without introducing hidden dependencies.
- Symmetry – Architectural symmetry implies that a system is consistent and balanced in its attributes.
- Emergence – Emergent, self-organized behavior and control.

13.5 Architectural Decisions

Architectural decisions bridge requirements and architecture.

1. Determine which information items are needed for each decision.
2. Define links between each decision and appropriate requirements.
3. Provide mechanisms to change status when alternative decisions need to be evaluated.
4. Define prerequisite relationships among decisions to support traceability.
5. Link significant decisions to architectural views resulting from decisions.
6. Document and communicate all decisions as they are made.

13.6 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

ADLs

- *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.

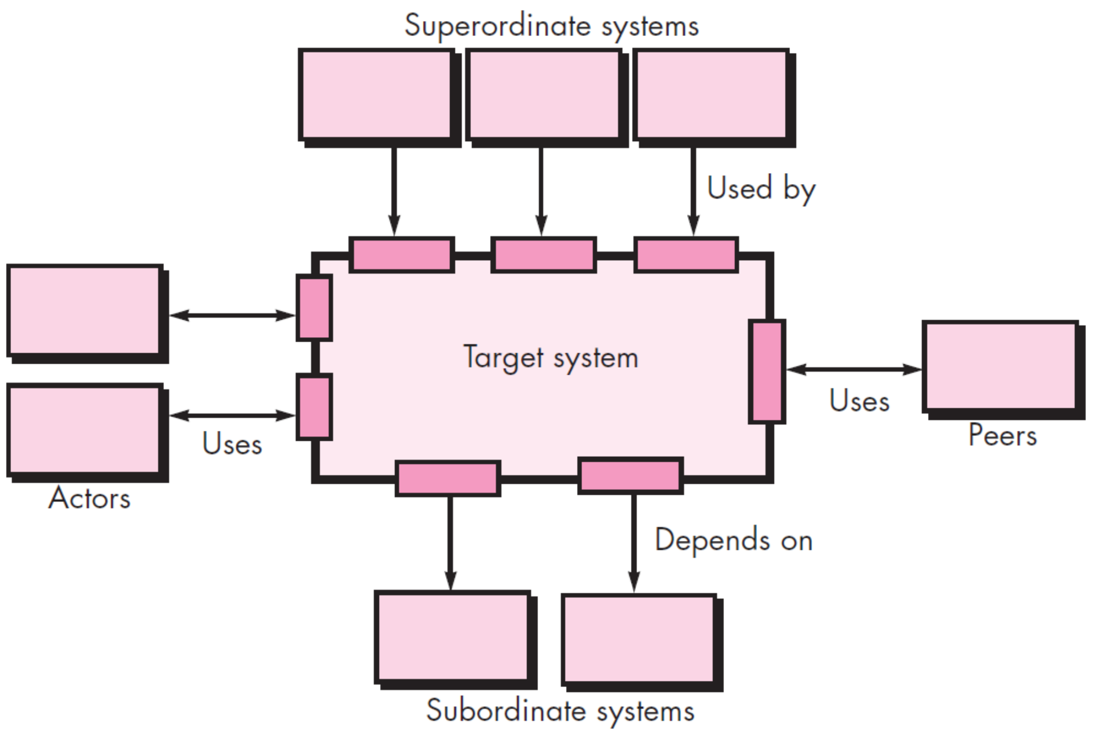
13.6.1 Representing the system in Context

Generic structure of the context diagram.

FIGURE 9.5

Architectural context diagram

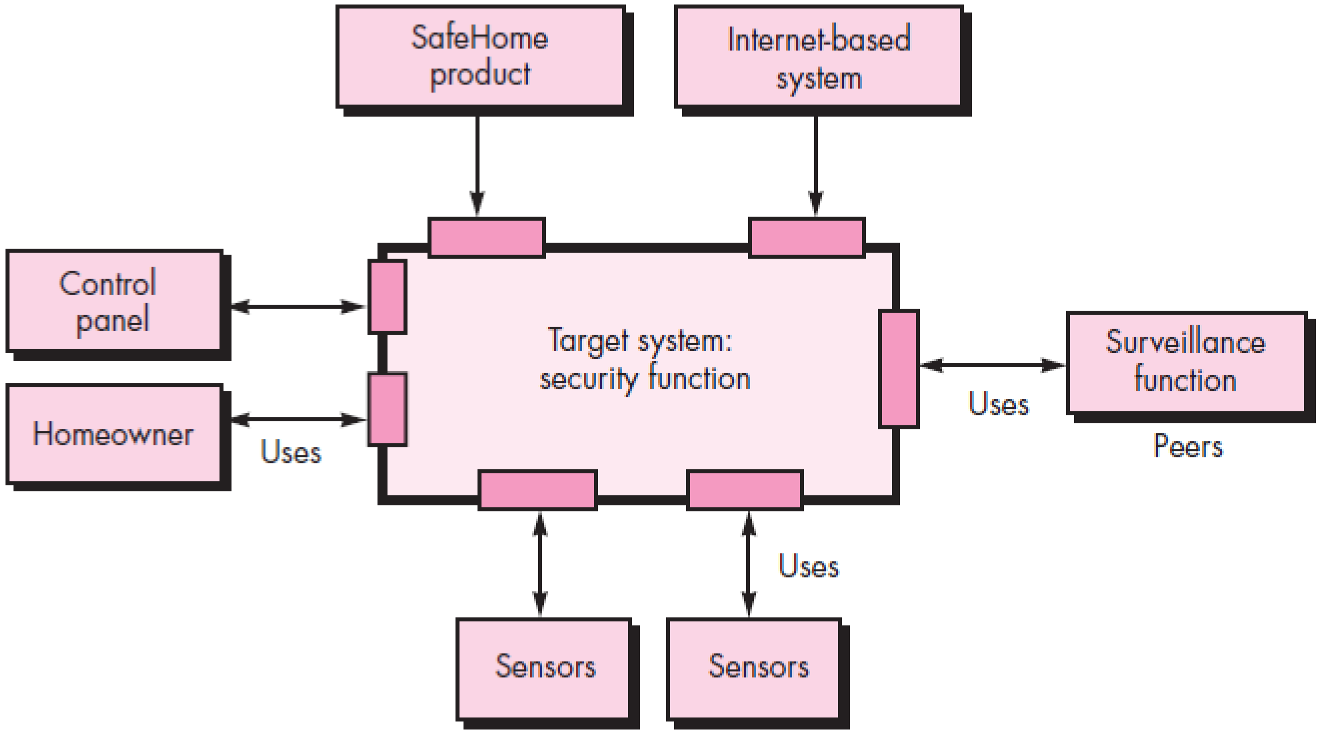
Source: Adapted from [Bos00].



Architecture design begins with this (i.e., the context diagram).

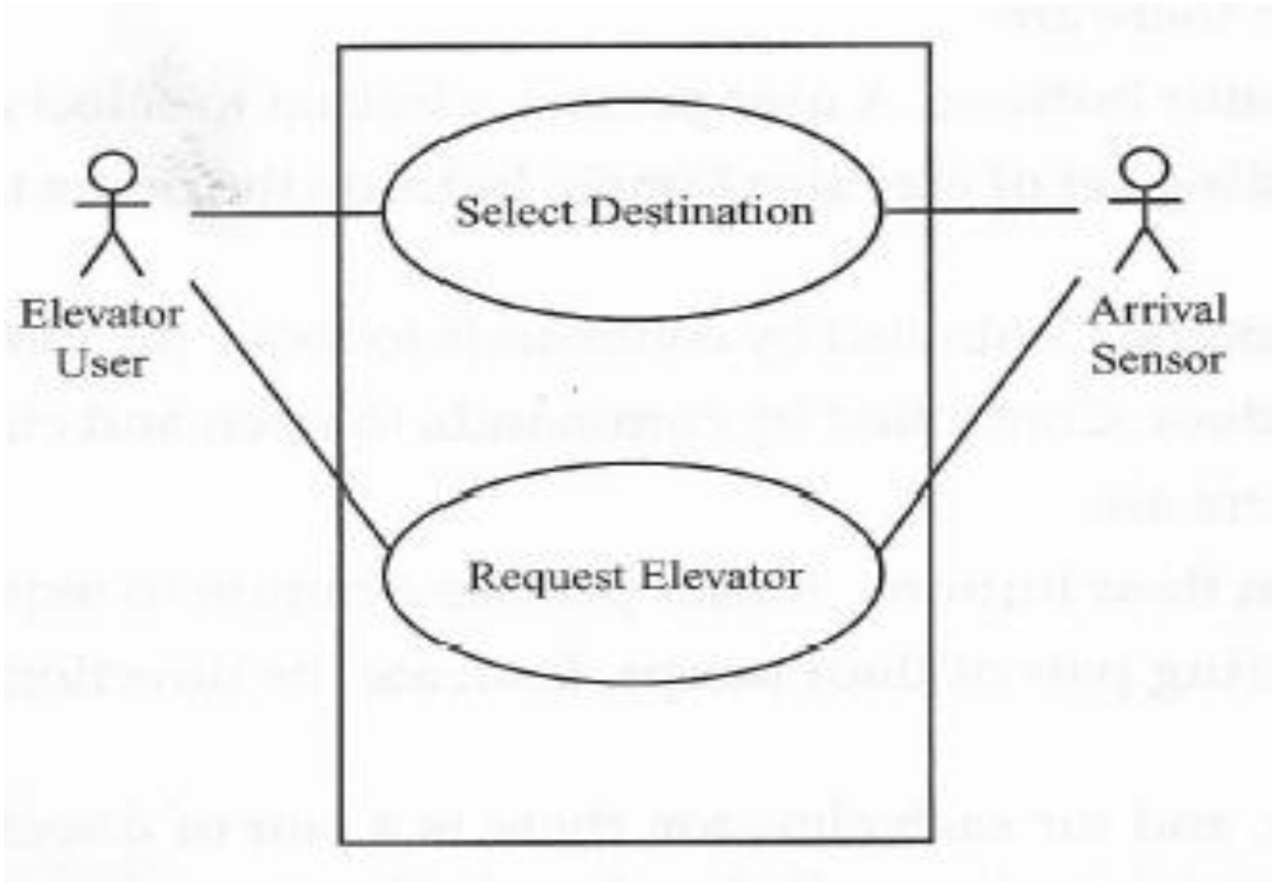
FIGURE 9.6

Architectural context diagram for the *SafeHome* security function



Example - Elevator Control System

- ECS actors and use cases



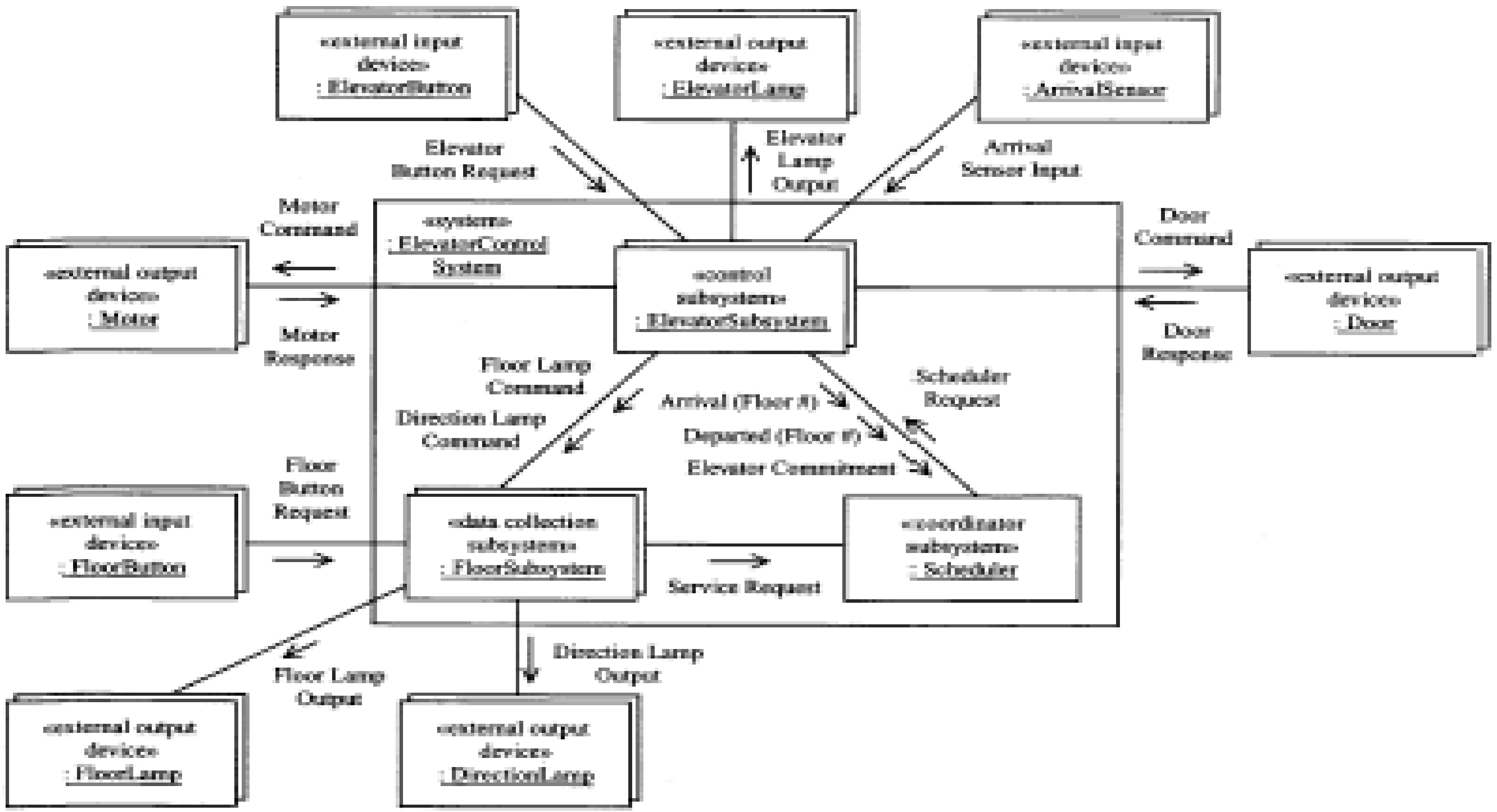
Software Architecture Design

- The system is structured into subsystems.
  - For distributed application, the geographical location takes precedence.

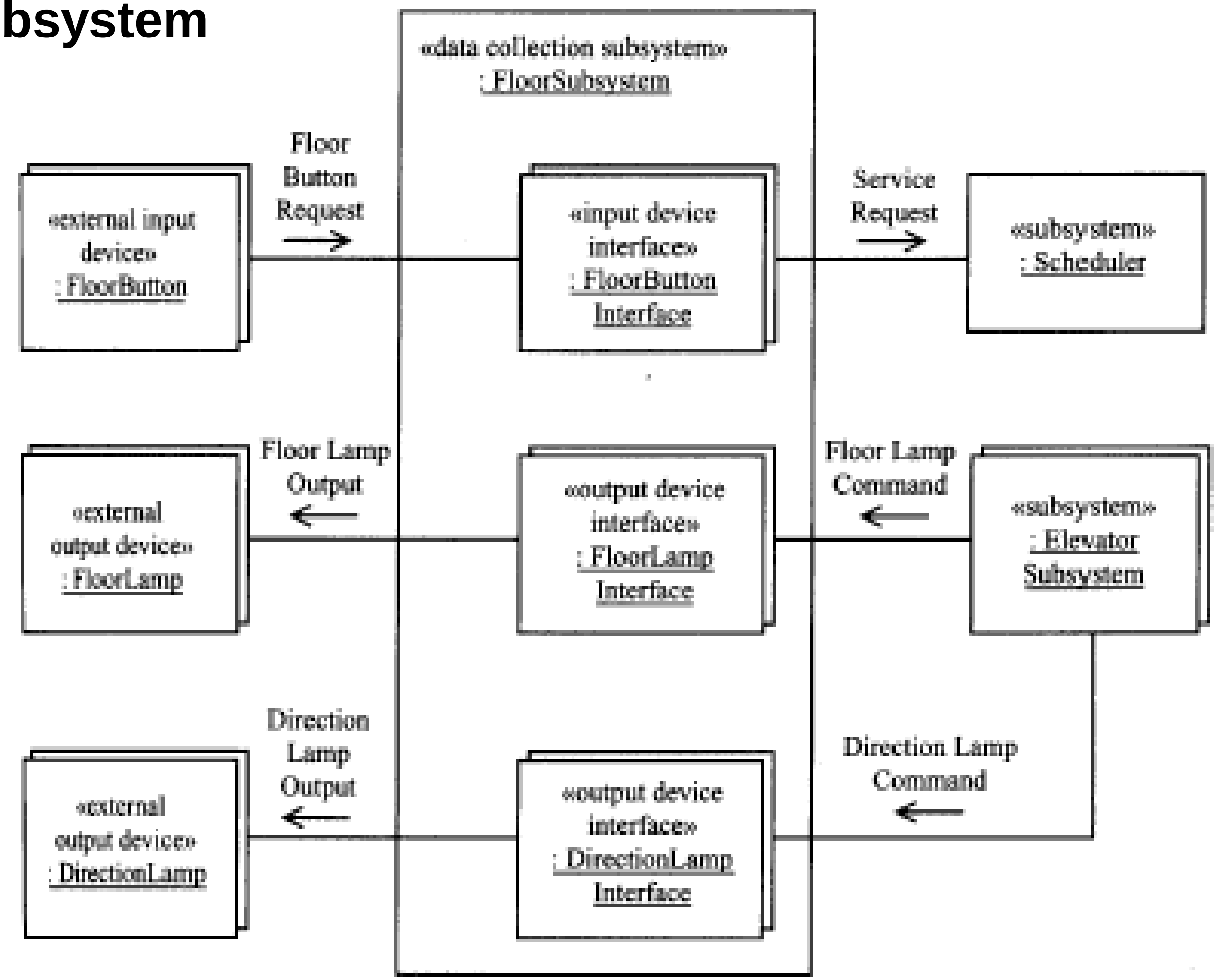
- The overall system architecture consists of
  - Multiple instances of the Elevator Subsystem (one per Elevator)
  - Multiple instances of the Floor Subsystem (one per floor)
  - One instance of the Scheduler Subsystem

Logical Architecture View

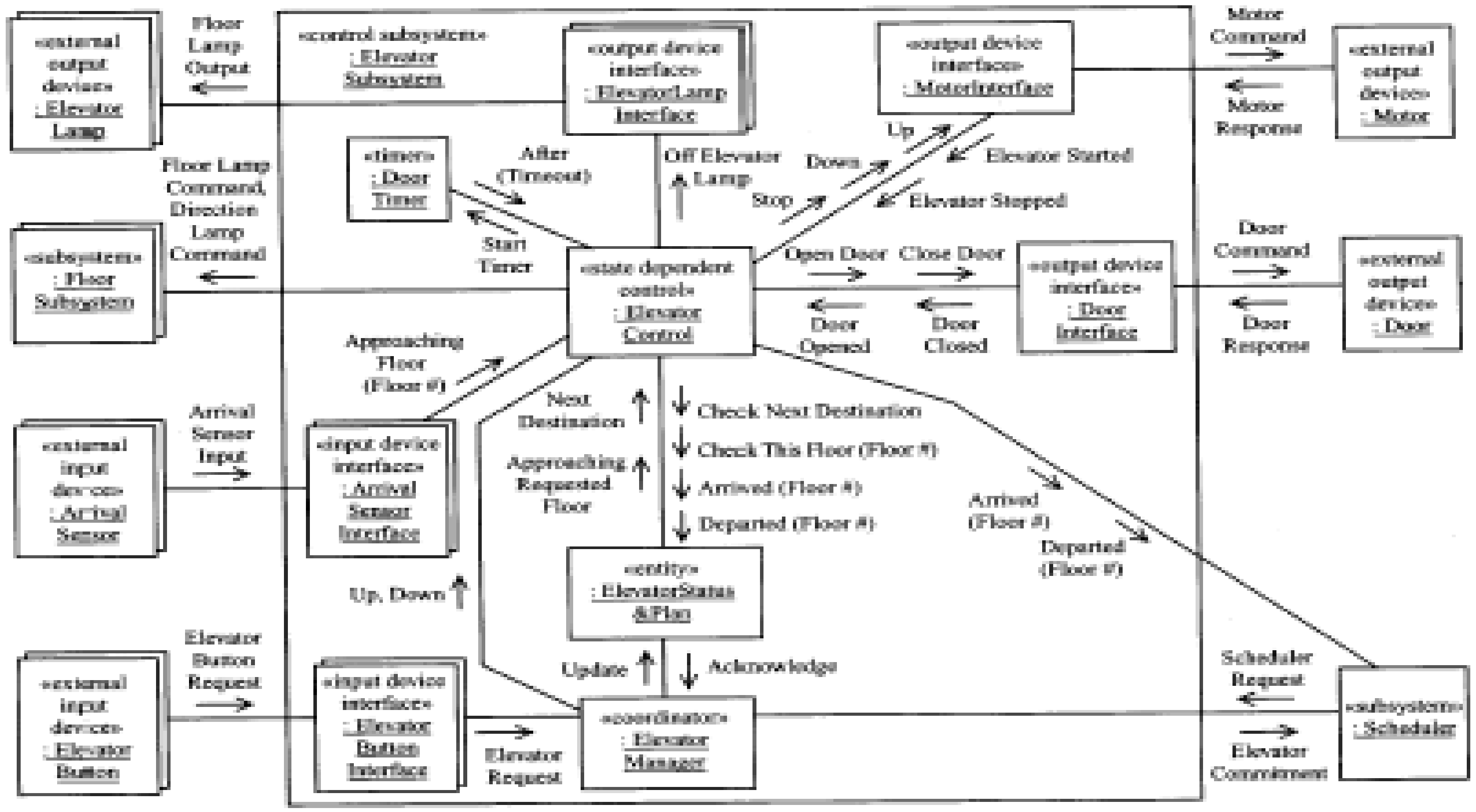
- **Overall system architecture**


«external input device» : ElevatorButton
«external output device» : ElevatorLamp
«external input device» : ArrivalSensor
Elevator Button Request
Elevator Lamp Output
Arrival Sensor Input
Motor Command
Motor Response
«system» : ElevatorControl System
«control subsystem» : ElevatorSubsystem
Door Command
Door Response
«external output device» : Motor
«external output device» : Door
Floor Lamp Command
Direction Lamp Command
Arrival (Floor #)
Departed (Floor #)
Elevator Commitment
Scheduler Request
Floor Button Request
«external input device» : FloorButton
«data collection subsystem» : FloorSubsystem
Service Request
«coordinator subsystem» : Scheduler
Floor Lamp Output
Direction Lamp Output
«external output device» : FloorLamp
«external output device» : DirectionLamp

- **Floor subsystem**


«data collection subsystem» : FloorSubsystem
«external input device» : FloorButton
Floor Button Request
«input device interface» : FloorButton Interface
Service Request
«subsystem» : Scheduler
«external output device» : FloorLamp
Floor Lamp Output
«output device interface» : FloorLamp Interface
Floor Lamp Command
«subsystem» : Elevator Subsystem
«external output device» : DirectionLamp
Direction Lamp Output
«output device interface» : DirectionLamp Interface
Direction Lamp Command

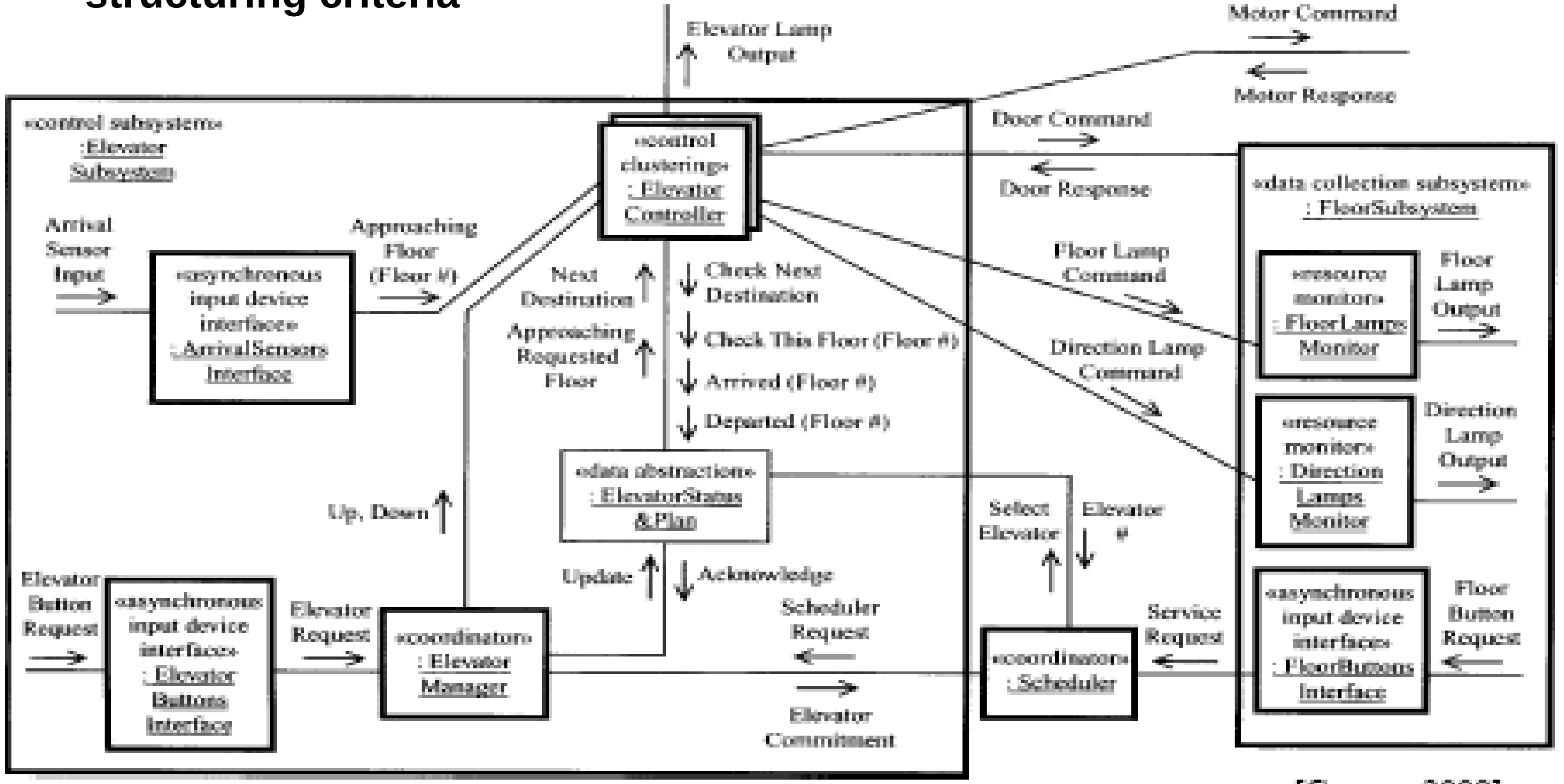
- **Elevator subsystem**



«external output device» : Elevator Lamp
Floor Lamp Output
«control subsystem» : Elevator Subsystem
«output device interface» : ElevatorLamp Interface
«output device interface» : MotorInterface
Motor Command
Motor Response
«external output device» : Motor
«timer» : Door Timer
After (Timeout)
Start Timer
Off Elevator Lamp
Up
Down
Stop
Elevator Started
Elevator Stopped
Floor Lamp Command, Direction Lamp Command
«subsystem» : Floor Subsystem
«state dependent control» : Elevator Control
Open Door
Close Door
Door Opened
Door Closed
«output device interface» : Door Interface
Door Command
Door Response
«external output device» : Door
Approaching Floor (Floor #)
Arrival Sensor Input
«external input device» : Arrival Sensor
«input device interface» : Arrival Sensor Interface
Next Destination
Approaching Requested Floor
Check Next Destination
Check This Floor (Floor #)
Arrived (Floor #)
Departed (Floor #)
Arrived (Floor #)
Departed (Floor #)
«entity» : ElevatorStatus &Plan
Up, Down
Update
Acknowledge
Elevator Button Request
«external input device» : Elevator Button
«input device interface» : Elevator Button Interface
Elevator Request
«coordinator» : Elevator Manager
Scheduler Request
Elevator Commitment
«subsystem» : Scheduler

Concurrent Task Architecture Design: Task Architecture View

- **Analyze all objects on the collaboration diagrams and apply task structuring criteria**



«control subsystem» :Elevator Subsystem
Arrival Sensor Input
«asynchronous input device interface» : ArrivalSensors Interface
Approaching Floor (Floor #)
«control clustering» : Elevator Controller
Elevator Lamp Output
Motor Command
Motor Response
Door Command
Door Response
Next Destination
Approaching Requested Floor
Check Next Destination
Check This Floor (Floor #)
Arrived (Floor #)
Departed (Floor #)
«data abstraction» : ElevatorStatus &Plan
Up, Down
Update
Acknowledge
Elevator Button Request
«asynchronous input device interface» : Elevator Buttons Interface
Elevator Request
«coordinator» : Elevator Manager
Scheduler Request
Elevator Commitment
Floor Lamp Command
Direction Lamp Command
Select Elevator
Elevator #
«coordinator» : Scheduler
Service Request
«data collection subsystem» : FloorSubsystem
«resource monitor» : FloorLamps Monitor
Floor Lamp Output
«resource monitor» : Direction Lamps Monitor
Direction Lamp Output
«asynchronous input device interface» : FloorButtons Interface
Floor Button Request

[Gomaa, 2000]

13.7 Assessing Alternative Architectural Designs

An Architecture Trade-Off Analysis Method (ATAM)

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.
   <= Brainstorm questions to ask for analysis of quality attributes
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.

Architecture Analysis

Architecture Reviews

- Assess the ability of the software architecture to meet the systems quality requirements and identify potential risks

  - Can reduce project costs by detecting design problems early

  - Can use experience-based reviews, prototype evaluation, and scenario reviews, and checklists

    (≈ questions)

Architectural Complexity

- Assessed by considering the dependencies between components [Zha98]
  - Flow dependencies
    - dependence relationships between producers and consumers of resources.
  - Sharing dependencies
    - dependence relationships among consumers who use the same resource or producers who produce for the same consumers.
  - Constrained dependencies
    - constraints on the flow of control among a set of activities.

13.8 Lessons Learned

- To justify design decisions, various decision analysis and resolution(DAR) methods can be used:
  - **Chain of causes**: a form of root cause analysis in which the team defines an architectural goal and then list the related actions
  - **Ishikawa fishbone**: a graphical technique that identifies the many possible actions to achieve a goal
  - **Mind mapping or spider diagrams**: represent words, concepts, tasks, or software engineering artifact arranged around central keyword, constraint, or requirement

13.9 Pattern-Based Architecture Review

1. Identify and discuss the quality attributes by walking through the use cases.
2. Discuss a diagram of system’s architecture in relation to its requirements.
3. Identify the architecture patterns used and match the system’s structure to the patterns’ structure.
4. Use existing documentation and use cases to determine each pattern’s effect on quality attributes.
5. Identify all quality issues raised by architecture patterns used in the design.
6. Develop a short summary of issues uncovered during the meeting and make revisions to the walking skeleton.

13.10 Architectural Conformance Checking

- Conformance of architecture to requirements
  - Functional requirements
  - Non-functional requirements
  - Constraints
- Conformance of implementation to architecture

Jeff Bezos Mandate (around 2002)

1. *"All teams will henceforth expose their data and functionality through service interfaces."*
2. *"Teams must communicate with each other through these interfaces."*
3. *"There will be no other form of interprocess communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network."*
4. *"It doesn't matter what [API protocol] technology you use."*
5. *"Service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions."*
6. *"Anyone who doesn't do this will be fired."*
7. *"Thank you; have a nice day! "*

☛ Known to have enabled Amazon to be a successful platform company.
☛ Checking violations of this mandate ensures implementation's conformance to architecture.

13.11 Agility and Architecture

- To avoid rework, user stories are used to create and evolve an architectural model (walking skeleton) before coding
- Hybrid models which allow software architects contributing users stories to the evolving storyboard
- Well run agile projects include delivery of work products during each sprint
- Reviewing code emerging from the sprint can be a useful form of architectural review

☛ In an agile development process (which we didn't cover in this course), documentation is deemphasized and working software and face-to-face communication is emphasized.

☛ But still an architecture is designed and evolved.