Chapter 10
Architectural Design

Moonzoo Kim
CS Division of EECS Dept.
KAIST

KAIST

Overview of Ch 10. Creating an Architectural Design

- 10.1 Software Architecture
  - What Is Architecture
  - Why Is Architecture Important
- 10.2 Data Design
  - Data Design at the Architectural Level
  - Data Design at the Component Level
- 10.3 Architectural Styles and patterns
  - A Brief Taxonomy of Architectural Styles
  - Architectural Patterns
  - Organization and Refinement
- 10.4 Architectural Design
  - Representing the System in Context
  - Defining Archetypes
  - Refining the Architecture into Components
  - Describing Instantiations of the System
- 10.5 Assessing Alternative Architectural Designs
  - An Architecture Trade-off Analysis Method
  - Architectural Complexity
  - Architectural Description Language
- 10.6 Mapping Data Flow into a SW architecture

KAIST

What is Software Architecture

- A Software Architecture provides a fundamental description of a system, detailing
  - the components that make up the system
  - the significant collaborations between those components, including the data and control flows of the system
- A Software Architecture attempts to provide a sound basis for analysis, decision making, and risk assessment of both design and performance
- Architecture is an asset that constitutes tangible value to the organization that has created it

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

Data Design

- The data design action translates data objects defined as part of the analysis model into
  - A database architecture at the application level (when necessary)
  - Data structures at the software component level
- At the architectural level …
  - Design of one or more databases to support the application architecture
  - Design of methods for 'mining' the content of multiple databases
    - Navigate through existing databases in an attempt to extract appropriate business-level information
    - Design of a data warehouse (an additional layer to the data architecture)—a large, independent database that has access to the data that are stored in databases that serve the set of applications required by a business

KAIST

Data Design

- At the component level …
  - refine data objects and develop a set of data abstractions
  - implement data object attributes as one or more data structures
  - review data structures to ensure that appropriate relationships have been established
  - simplify data structures as required

KAIST

Data Design—Component Level

1. The systematic analysis principles applied to function and behavior should also be applied to data.
2. All data structures and the operations to be performed on each should be identified.
3. A mechanism for defining the content of each data object should be established and used to define both data and the operations applied to it.
4. Low level data design decisions should be deferred until late in the design process.
5. The representation of data structure should be known only to those modules that must make direct use of the data contained within the structure (*information hiding*).
6. A library of useful data structures and the operations that may be applied to them should be developed.
7. A software design and programming language should support the specification and realization of abstract data types.

KAIST

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,

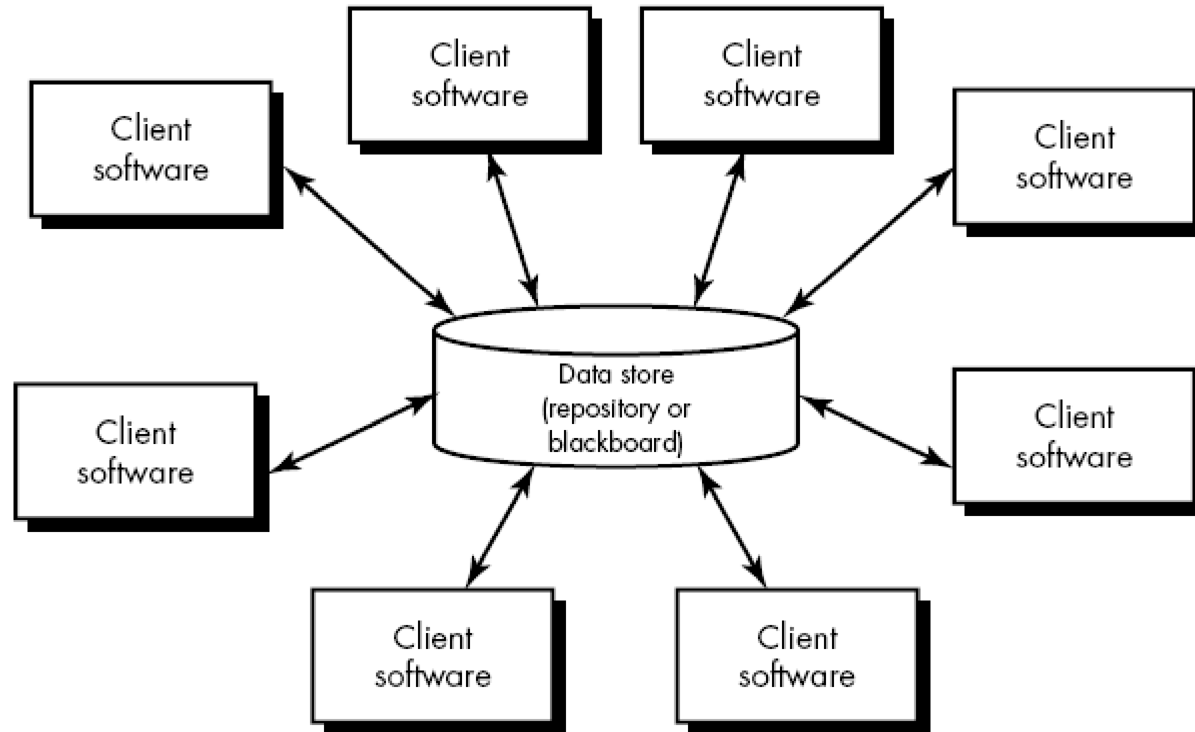
(3) constraints that define how components can be integrated to form the system

(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 Styles

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

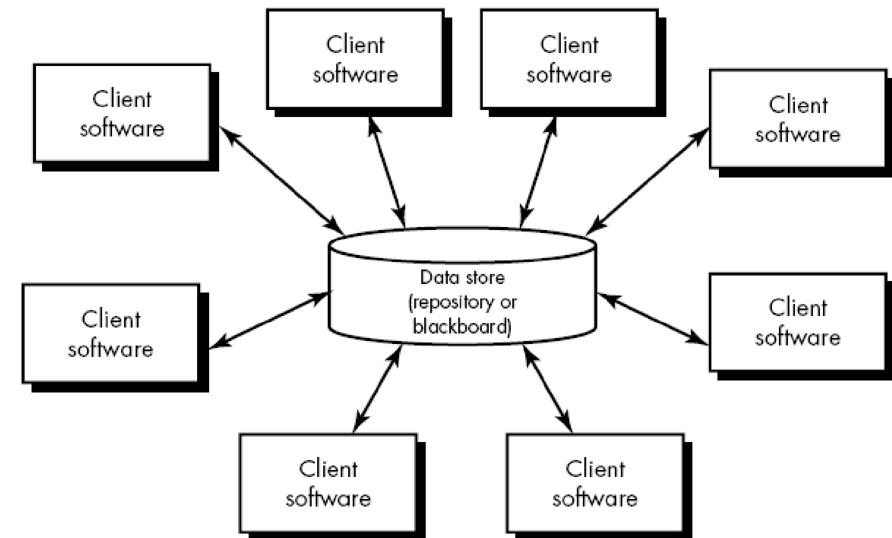
Data-Centered Architecture



KAIST

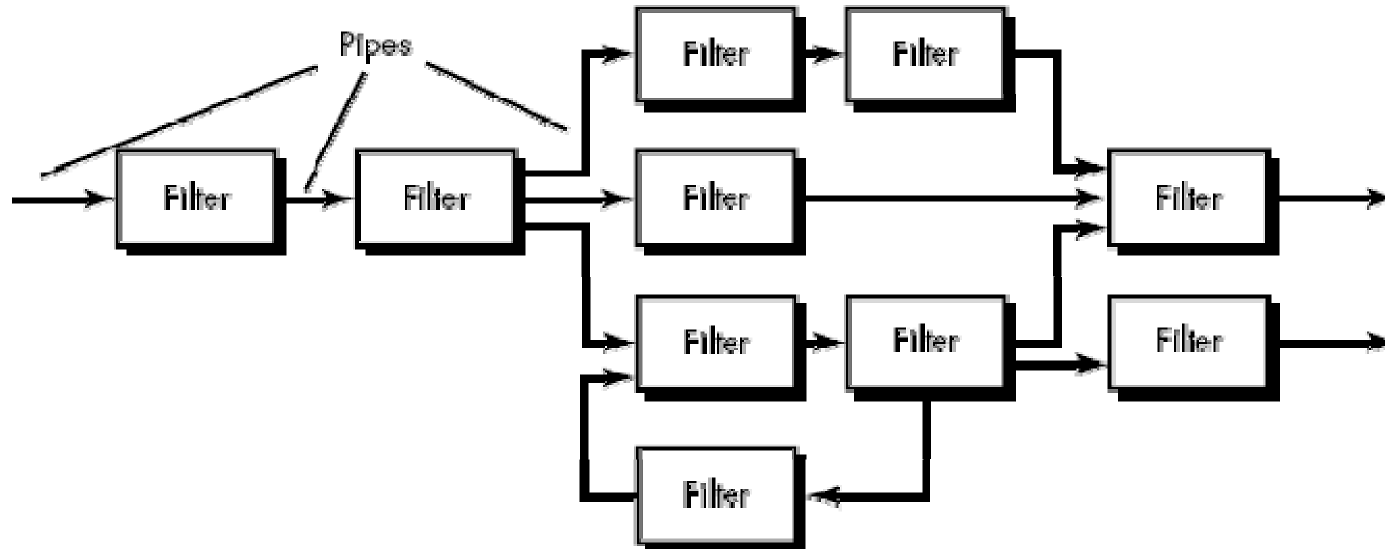
Data-Centered Architecture (cont.)

- Data repository is
  - Passive
    - Client SW accesses the data independent of any changes to the data or the actions of the other clients
  - Blackboard scheme
    - Sends notifications to client SW when data of interest to the client changes
- High integrability
  - Client components operate independently

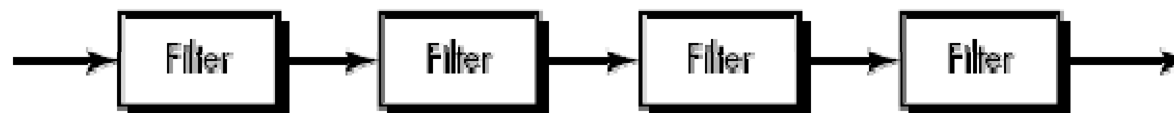


KAIST

Data Flow Architecture (pipe-filter pattern)



(a) Pipes and filters



(b) Batch sequential

KAIST

Pipes and Filters Pattern

- The Pipes and Filters pattern is a data-flow architectural pattern that views the system as a series of transformations on successive pieces of input data
- Pipes are stateless and serve as conduits for moving streams of data between multiple filters
- Filters are stream modifiers, which process incoming data in some specialized way and send that modified data stream out over a pipe to another filter

Excerpts from CSPP 51050 “OO Architecture…”
CS dept. Univ. of Chicago

Pipes and Filters Features

- Incremental delivery: data is output as work is conducted
- Concurrent (non-sequential) processing, data *flows* through the pipeline in a stream, so multiple filters can be working on different parts of the data stream simultaneously
  - Pipeline using different processes or threads
- Filters work *independently and ignorantly* of one another, and therefore are plug-and-play
  - Filters are ignorant of other filters in the pipeline
  - there are no filter-filter interdependencies
- Maintenance is again isolated to individual filters, which are loosely coupled
- Very good at supporting producer-consumer mechanisms
- Multiple readers and writers are possible

KAIST

Batch Sequential Data Processing

- Stand-alone programs would operate on data, producing a file as output
- This file would stand as input to another standalone program, which would read the file in, process it, and write another file out
- Each program was dependent on its version of input before it could begin processing
- Therefore processing took place sequentially, where each process in a fixed sequence would run to completion, producing an output file in some new format, and then the next step would begin

Benefits

- Fairly simple to understand and implement
- Simple, defined interface reduces complex integration issues
- Filters are substitutable black boxes, and can be plug and played, and thus *reused* in creative ways
- Filters are highly modifiable, since there's no coupling between filters and new filters can be created and added to an existing pipeline

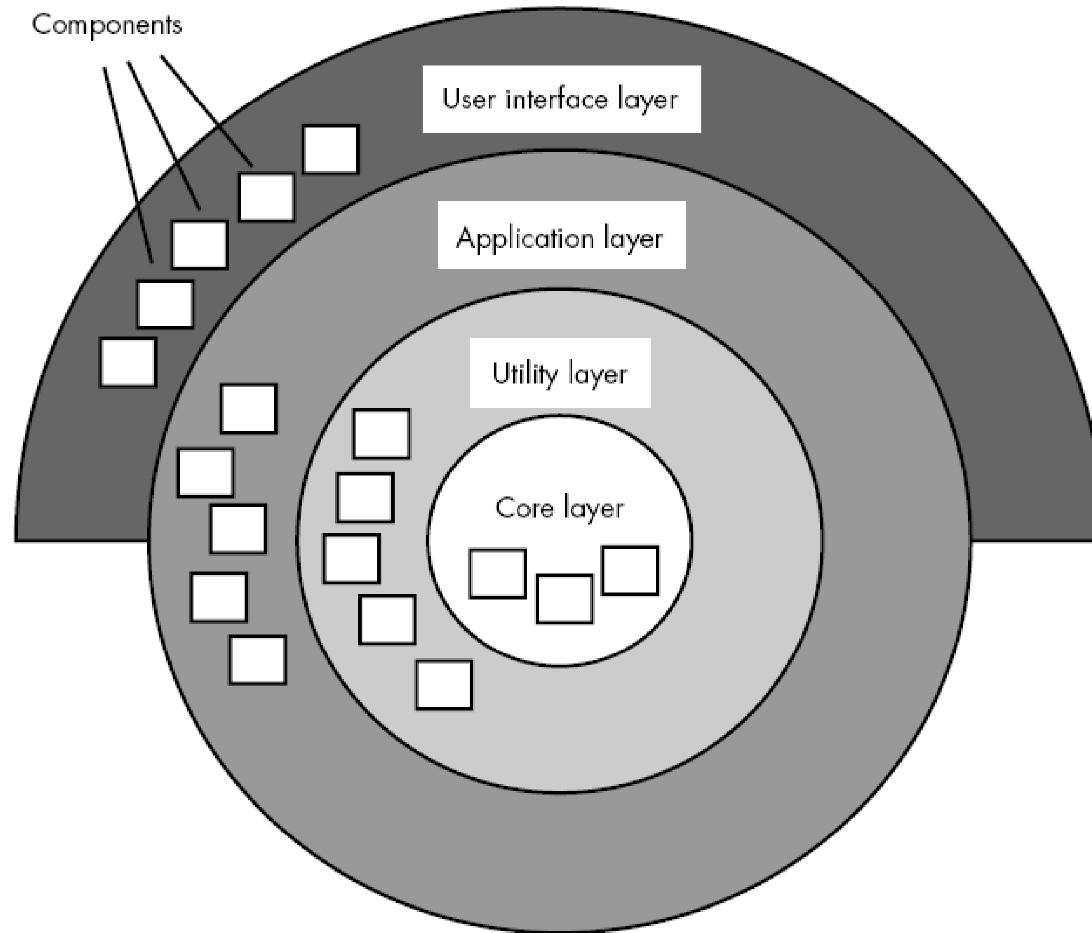
More Benefits

- Filters and Pipes can be hierarchical and can be composed into a mechanism to further simplify client access
- Because filters stand alone, they can be distributed easily and support concurrent execution (the stream is *in process*)
- Multiple filters can be used to design larger complex highly-modifiable algorithms, which may be modified by adding new filters or deleting others

Limitations

- A batch processing metaphor is not inherently limiting, but this pattern does not facilitate highly dynamic responses to system interaction
  - Because filters are black boxes, and are ignorant of one another, they cannot intelligently reorder themselves dynamically
  - Once a pipeline is in progress, it cannot be altered without corrupting the stream
  - Difficult to configure dynamic pipelines, where depending on content, data is routed to one filter or another

Layered Architecture



Layers

- Architectural layers are collaborating sections of an overall complex system that provide several benefits such as:
  - supporting incremental coding and testing, allowing localization of changes
  - well-defined interfaces allow substitution of different layers
  - protection between collaborating layers
  - Layers support a responsibility-driven architecture that divides subtasks into groups of related responsibilities

Layers Pattern

- In the pure sense, each layer provides services to the layer *directly* above it, and acts as a client to the layer *directly* below it
- In an "impure" implementation, distanced layers can be "bridged" which allows communication between them but reduces portability and flexibility and plug and play capability
- Each layer provides a defined interface to the layers above and below it
- Higher layers provide increasing levels of abstraction

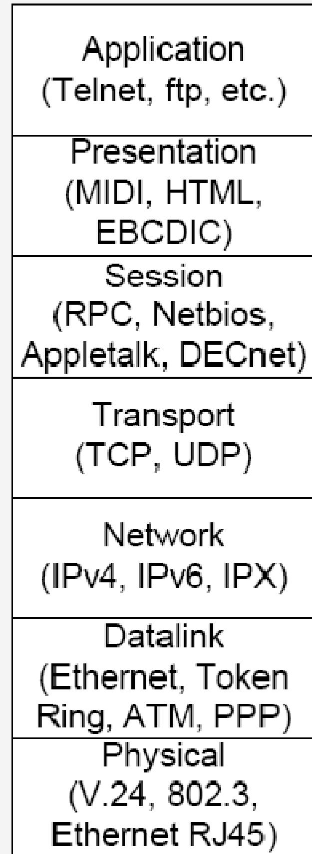
KAIST

Features

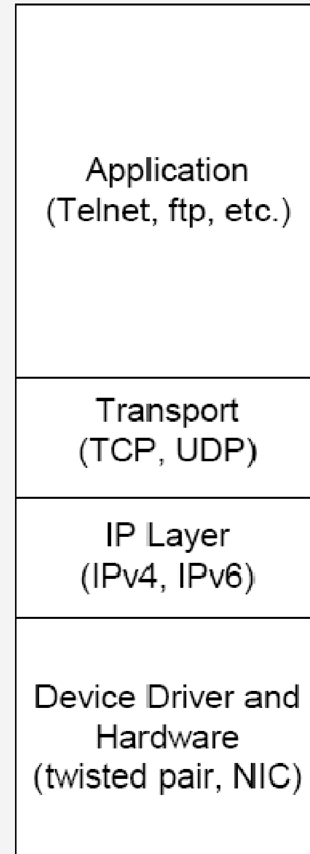
- Often, layered architectures are applied to virtual machine architectures (such as in interpreters)
  - hardware layers are virtualized in software, and either act as mediators to actual hardware layers, or are stubbed out entirely
- A layered system can be seen as a static pipes and filters system but without the pipes, where the filters talk directly to their neighbors

KAIST

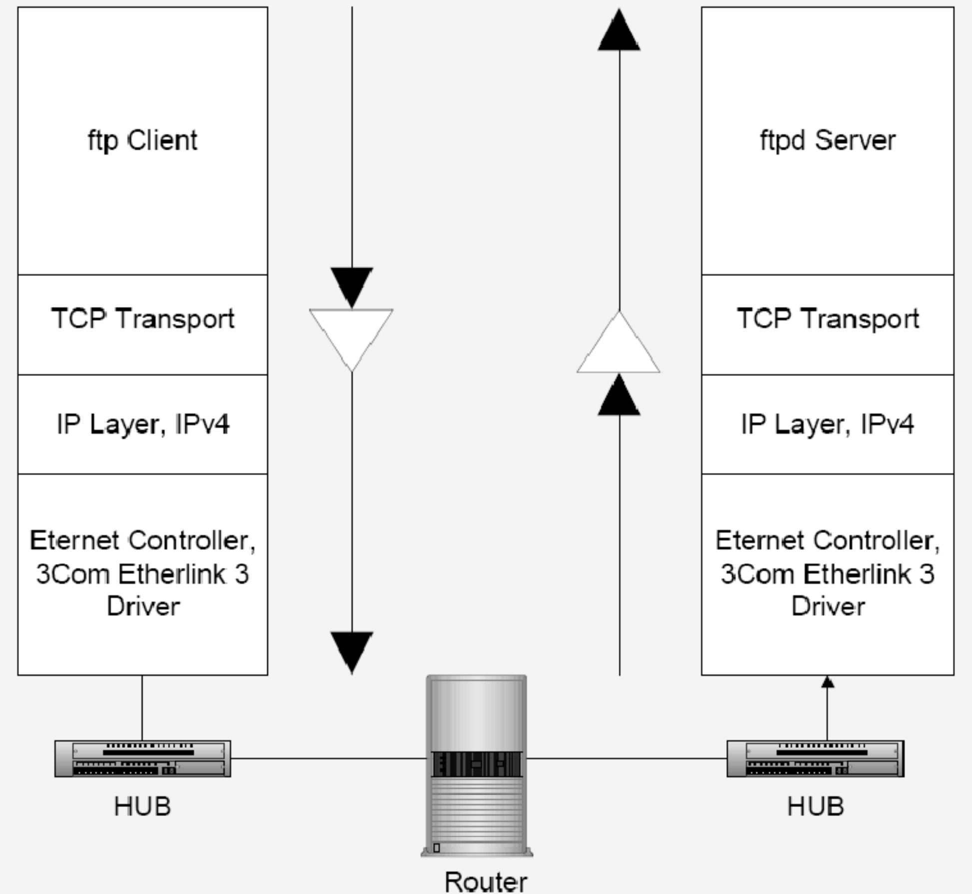
Example: Protocol Stacks



OSI Model
(Tannenbaum, 1988)



Internet Protocol Suite



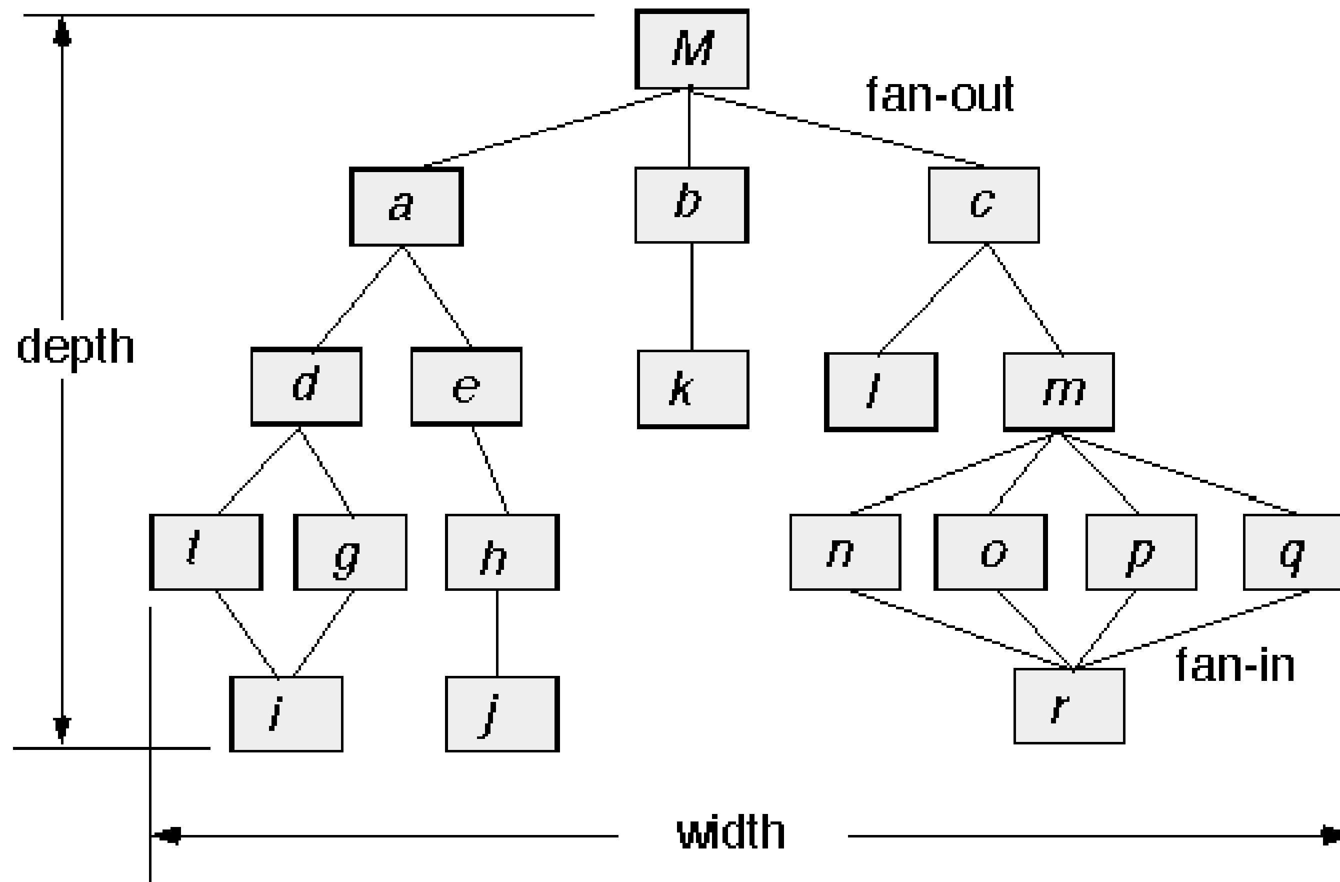
Benefits

- A layered pattern supports increasing levels of abstraction, thus simplifying design
  - allows a complex problem to be *partitioned* into a sequence of manageable incremental strategies (as layers)
- Like Pipes and Filters, layers are loosely coupled, so maintenance is enhanced because new layers can be added affecting only two existing components (as layers)
- Layers support plug-and-play (−reusable)designs. As long as the interfaces do not change, one layer can be substituted for another changing the behavior of the layer system

Disadvantages

- Close coupling of juxtaposed layers lowers maintainability
- Each layer must manage all data marshaling and buffering
- Lower runtime efficiency
- Sometimes difficult to establish the granularity of the various layers (10 layers or 4?)

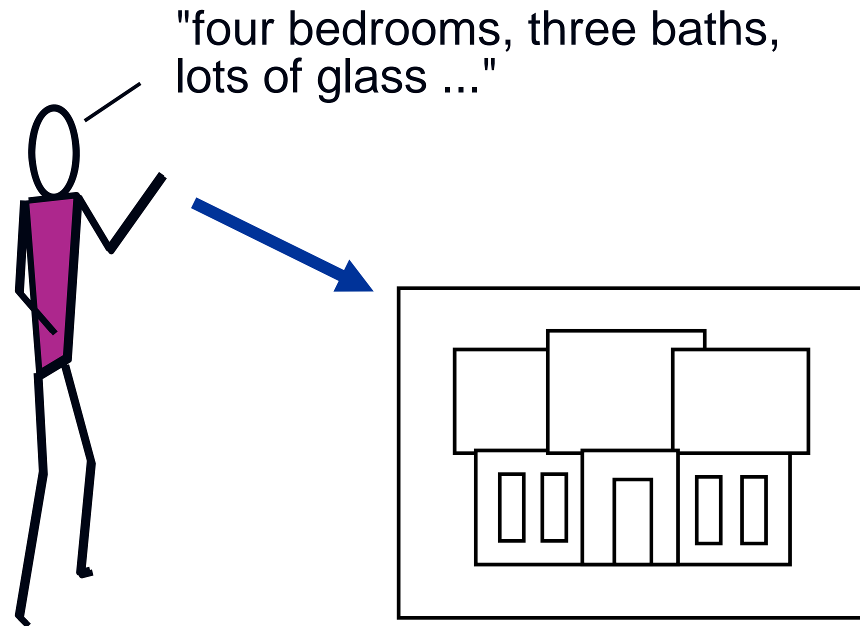
Call and Return Architecture



KAIST

An Architectural Design Method

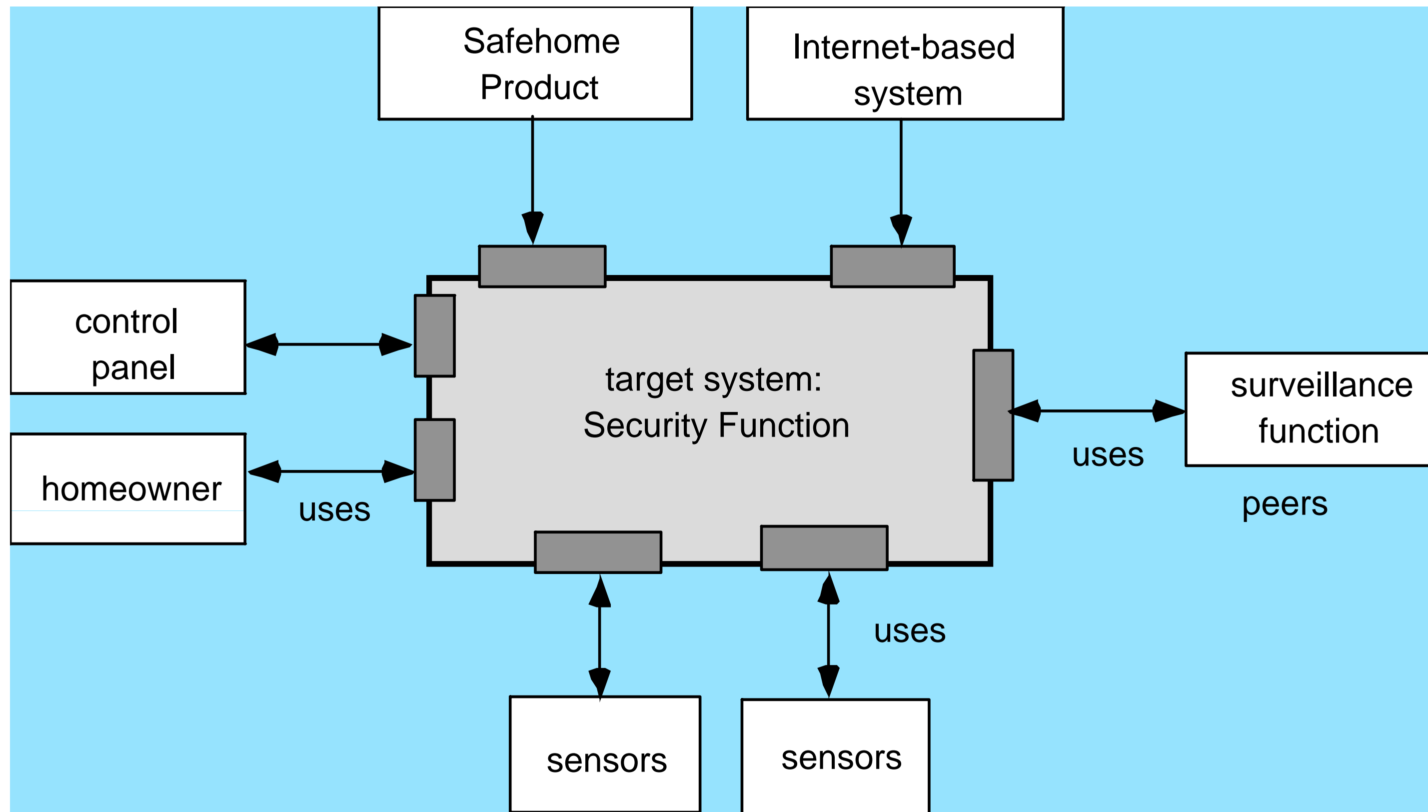
customer requirements



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

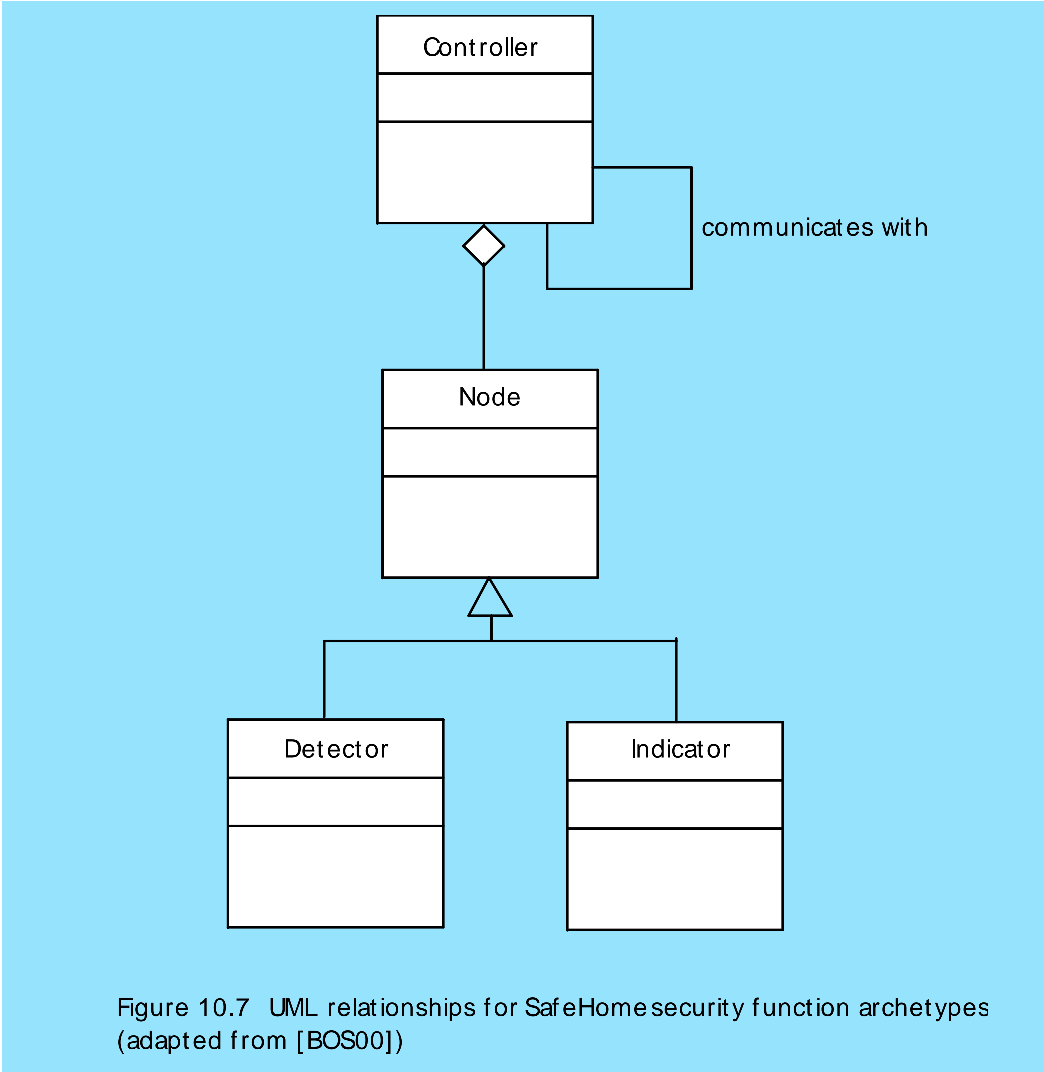
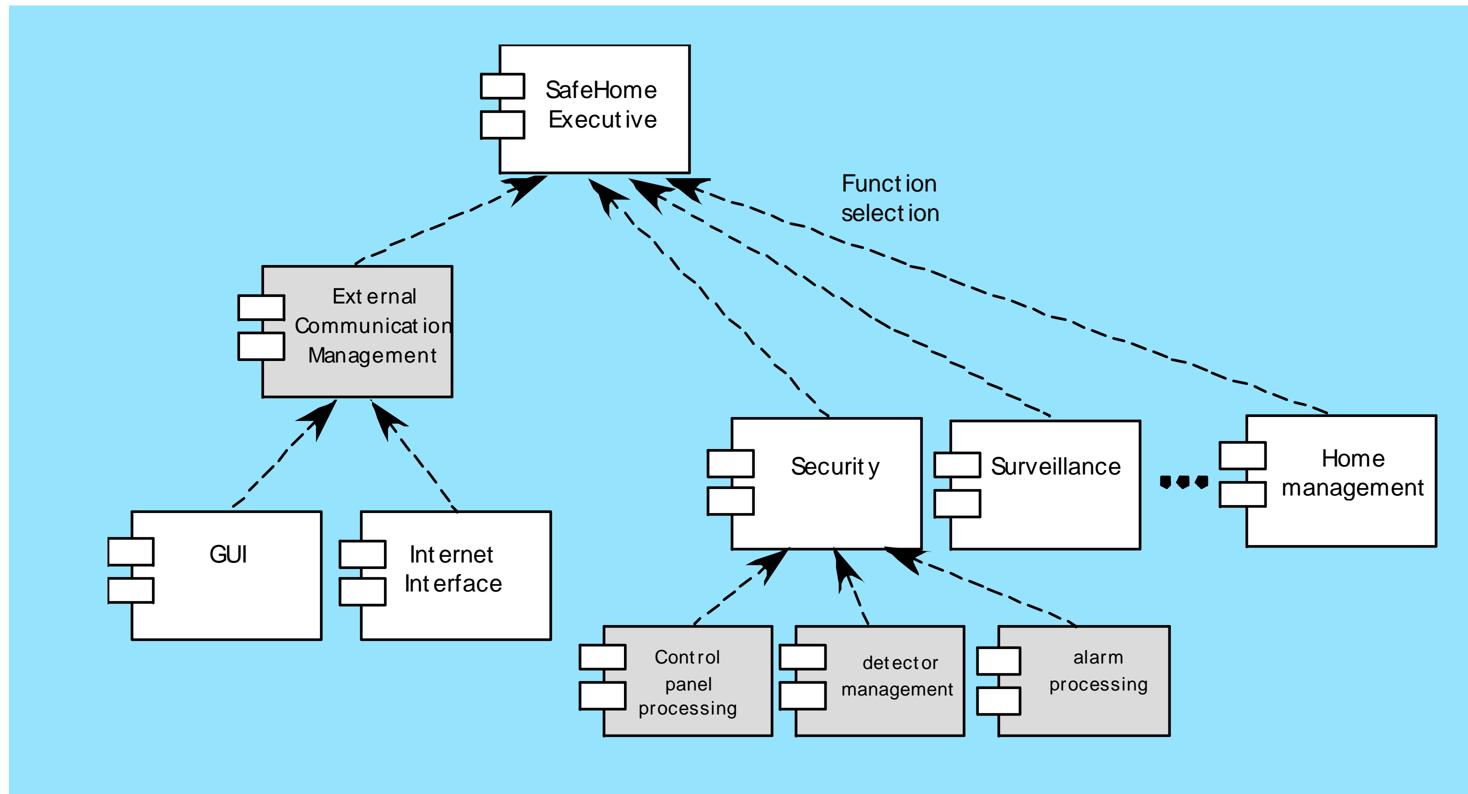
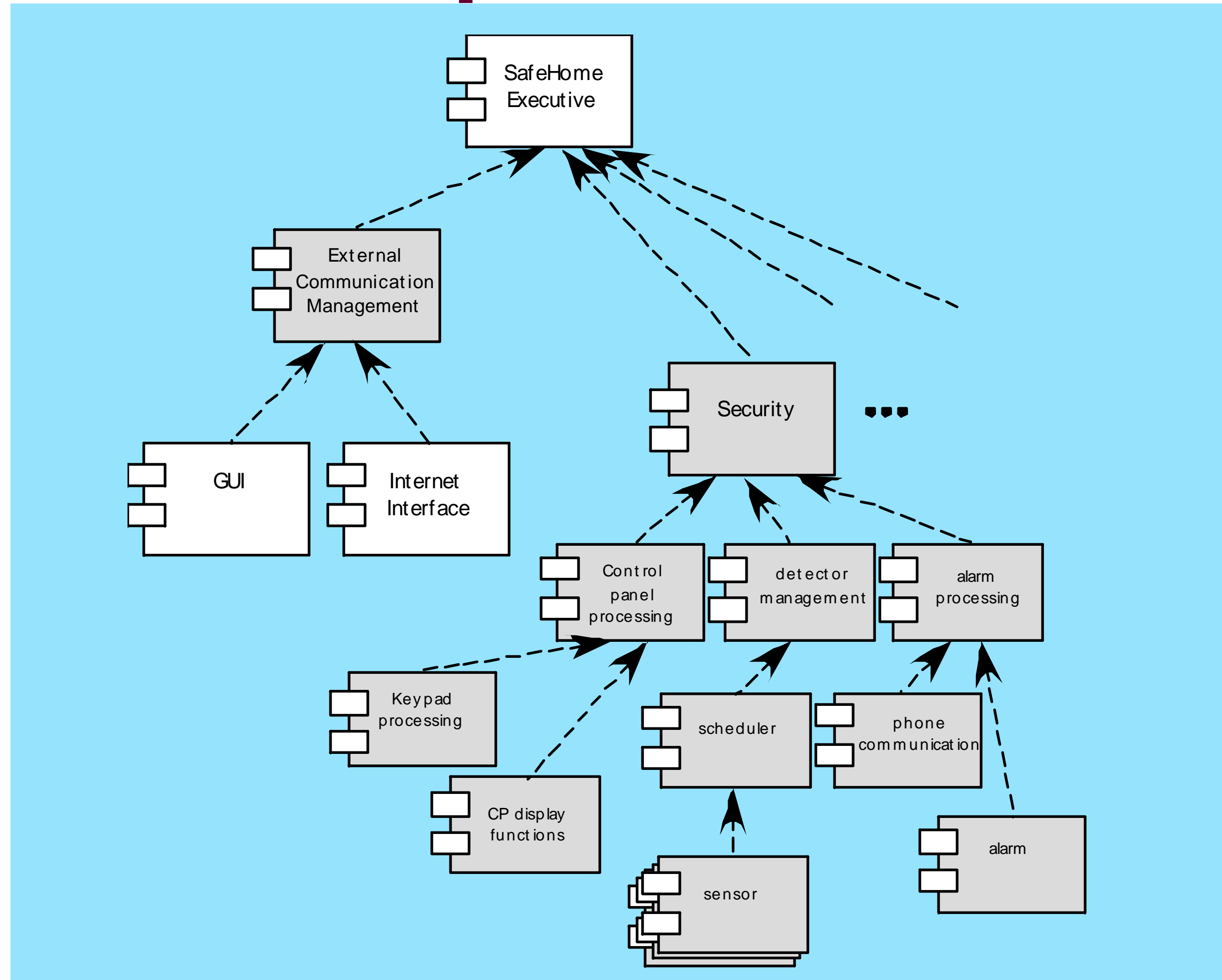


Figure 10.7 UML relationships for SafeHome security function archetypes (adapted from [BOS00])

Top-level Component Structure



Refined Component Structure



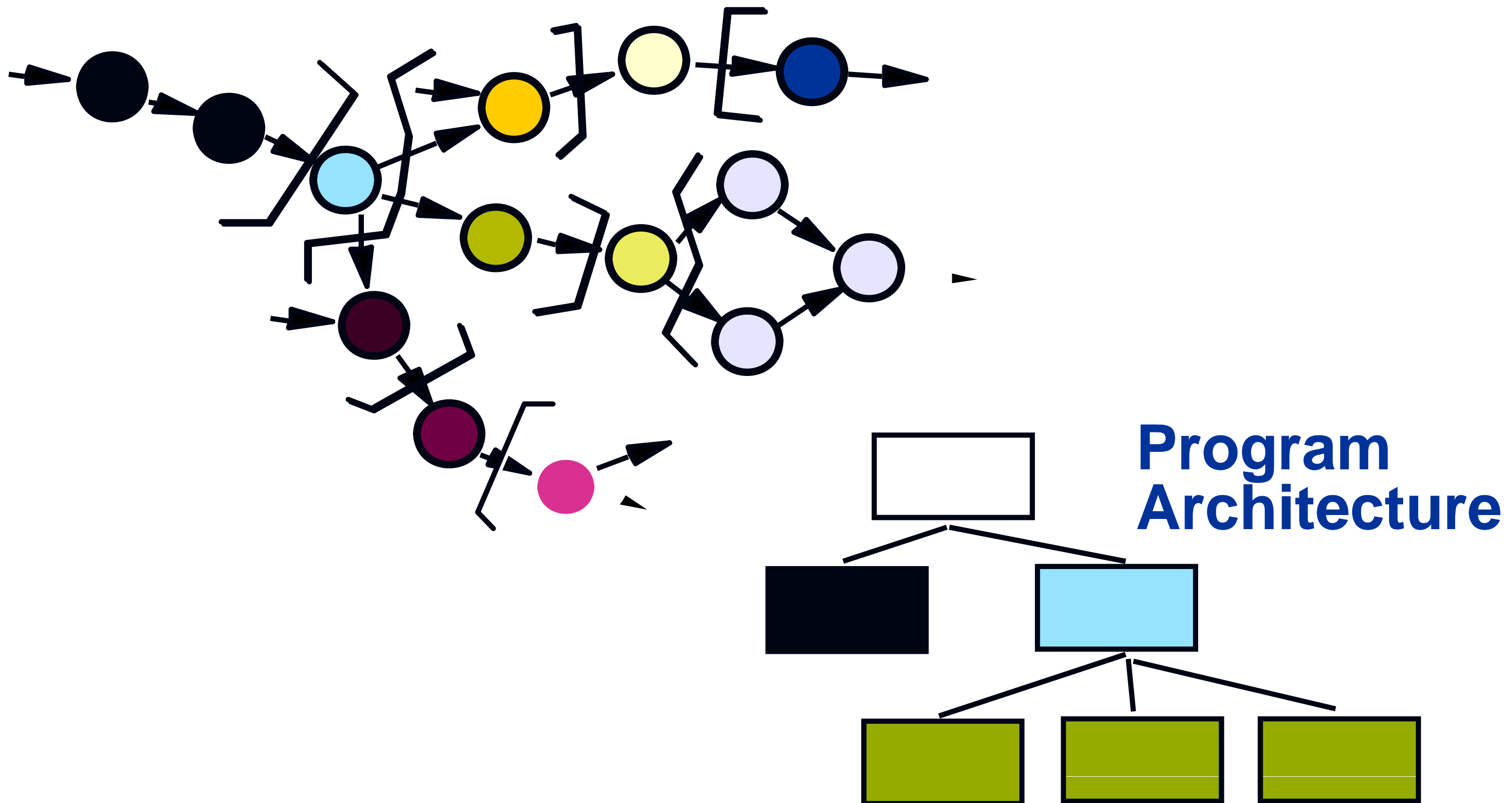
KAIST

Analyzing Architectural Design

Architecture trade-off analysis method (ATAM) by SEI

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 considering each attribute in isolation.
   - reliability, performance, maintainability, etc
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.

Deriving Program Architecture



KAIST

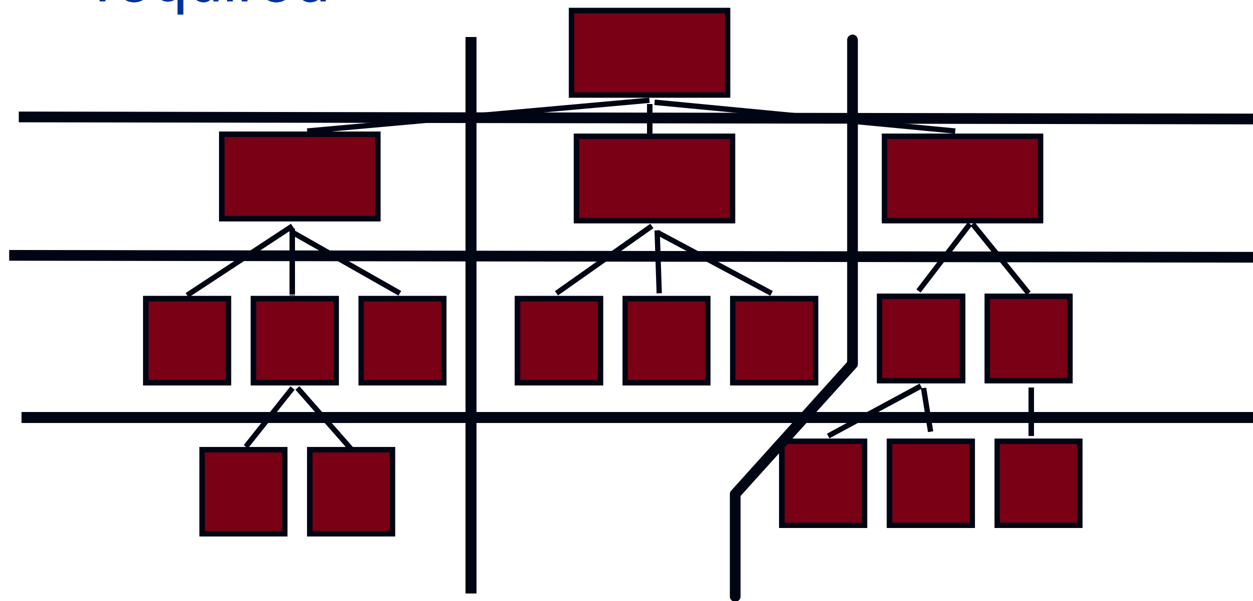
Why Partitioned Architecture?

- 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

Low coupling
More reuse
Better maintenance

Partitioning the Architecture

- “horizontal” and “vertical” partitioning are required



HW #3. Due Mar 29

8.2 Is it possible to develop an effective analysis model without developing all four elements shown in Fig.8.3? Explain

8.3 Is it possible to begin coding immediately after an analysis model has been created? Explain your answer and then argue the counterpoint (e.g. negative effects)

8.5 An analysis rule of thumb is that the model "should focus on requirements that are visible within the problem or business domain." What types of requirements are not visible in these domains? Provide a few examples.

8.19 Write a template-based use-case for the SafeHome home management system ("SafeHome home management function") described informally in the sidebar following Sect 8.7.4. Also draw a swimlane activity diagram (use a template at Sect.7.5)