Architectural Design

Today

- Software Architecture
- Why Software Architecture?
- Example
- Static Descriptions: Components, connectors and interfaces
- Architecture genre, descriptions
- Architecture styles
- Architecture Patterns

Software Architecture

The software architecture is the **earliest model** of the **whole software system** created along with the software lifecycle.

Example: Application

We want to develop a system that allows to vote electronically.

(1) The citizen **goes to the electoral place** and **votes using Hw/Sw**

(2) The vote is **store locally** and **automatically sent** to **other computer**

(3) The citizen identity must be **validated by the system**.

(4) So on……

Example: Basic Requirement

The voting system must satisfy the following requirements:

(1) One voter – one vote (no more than one vote for voter)

(2) The voter can vote in only one previous designated **voting place**

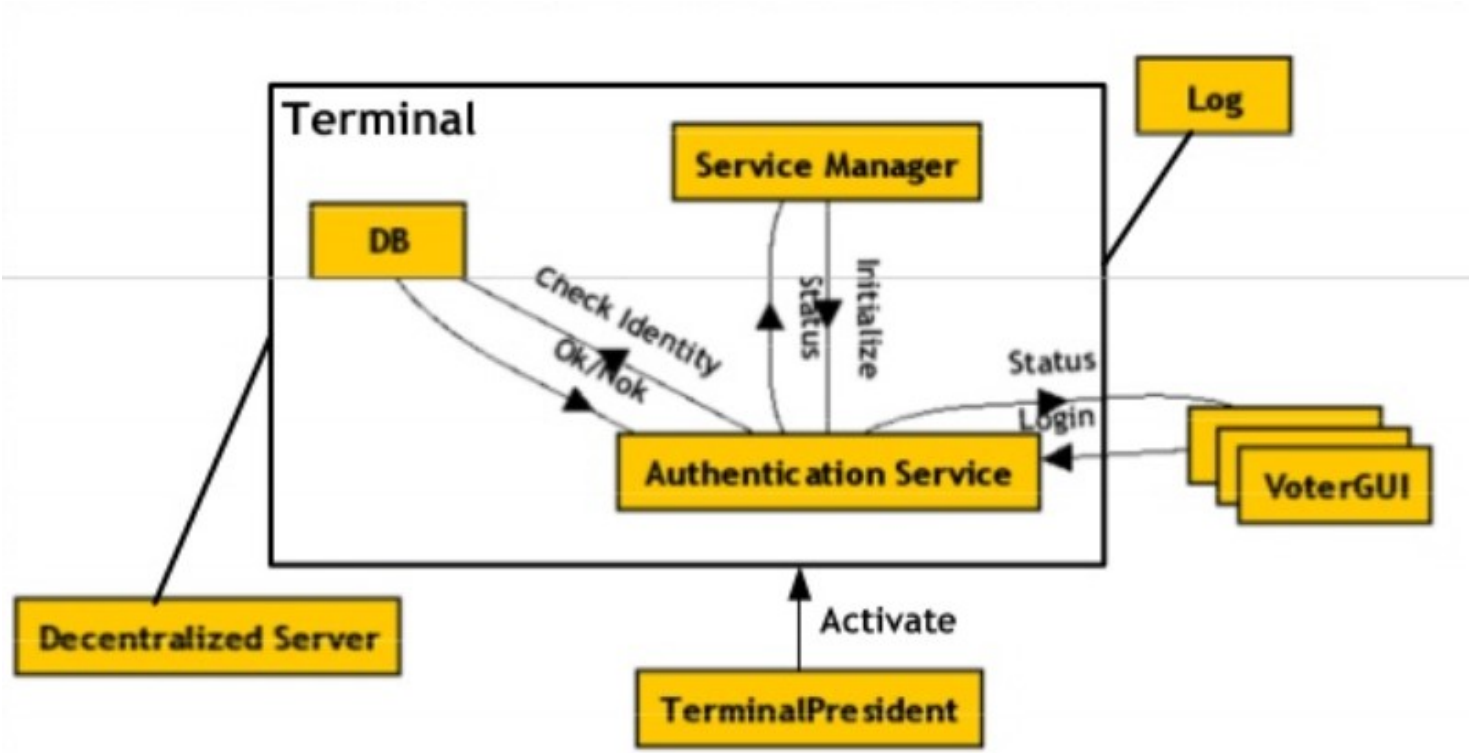
(3) The voter must be identified by the election officials at the voting place

    The citizen identity must be validated by the system

(4) It is not possible to trace the votes back to the voters

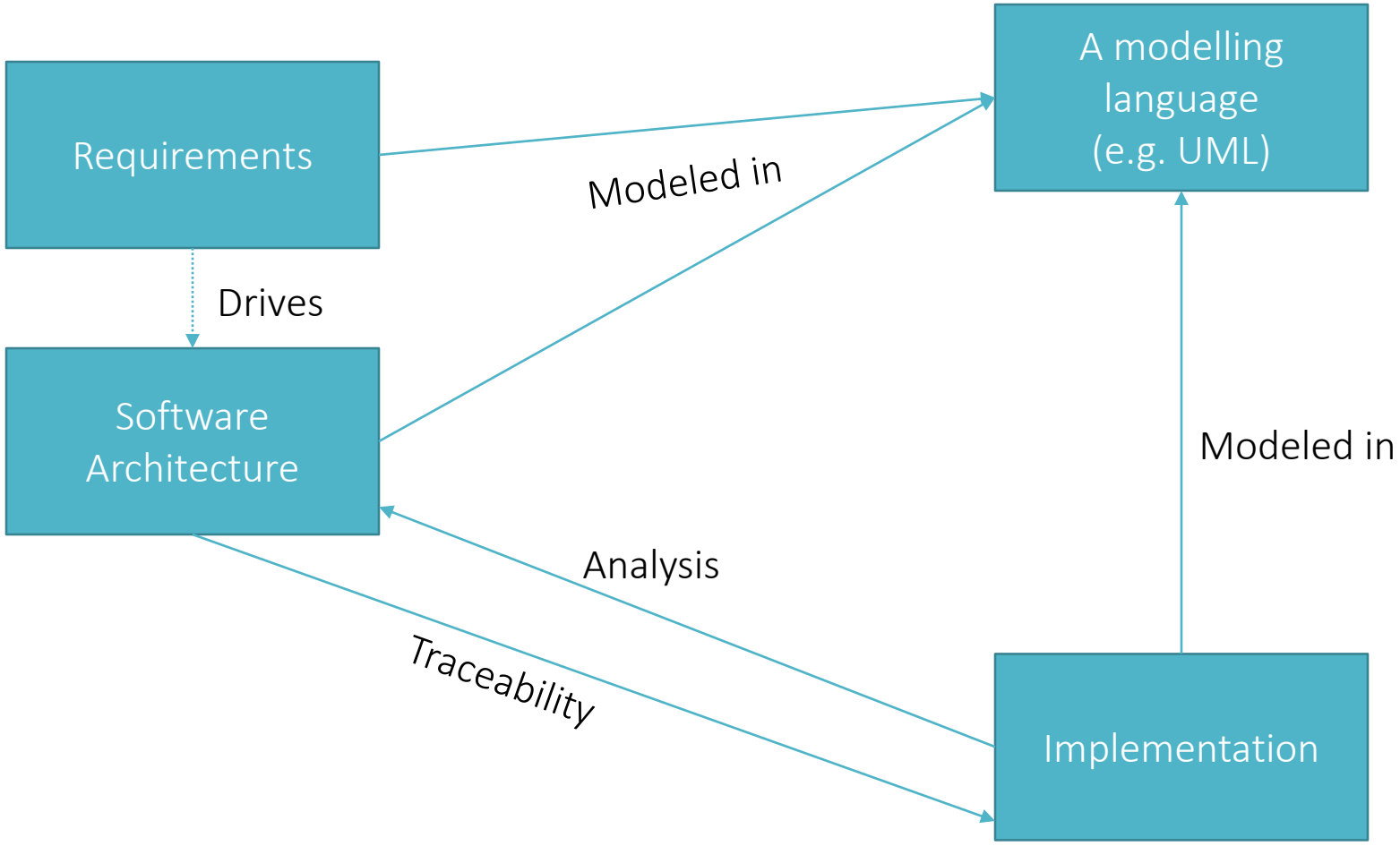
(5) The election officials can't read the results, guarantying that the results are unknown until the end of the voting process

Example



EXAMPLE TAKEN FROM INTRODUCTORY LECTURE TO SOFTWARE ARCHITECTURE, UNIVERSITY OF L'AQUILA, ITALY

Process



EXAMPLE TAKEN FROM INTRODUCTORY LECTURE TO SOFTWARE ARCHITECTURE, UNIVERSITY OF L'AQUILA, ITALY

History

1992 and 1993	1993-1995	1995-1997	1997-2003	2003-today
- SA is recognized as an independent area of research	- Described through box and line (i.e. informal diagrams)	Architectural Description Language(ADL's) are introduced to formally describe SA	- More interest on dynamic aspects of SA - SA recognized as a valid tool to deal with various aspects of complex, concurrent, real systems	SA vs component-based software engineering (CBSE), Aspect-oriented programming (AOP), Service oriented architecture (SOA), etc.…

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,

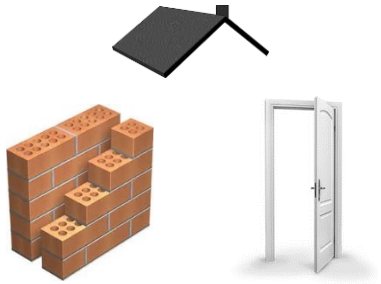
(2) **consider architectural alternatives** at a stage when making design changes is still relatively easy, and

(3) **reduce the risks** associated with the construction of the software.

(4) Stakeholder **communication**

Civil vs Software architecture

Civil:



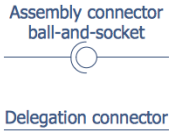
Software:

Civil:

Concrete
Construction
Standards

Software:

- Connectors
- Assembly constraints



IMAGES TAKEN FORM HTTP://WWW.PGHBRICKS.COM.AU/PRODUCTS/BRICKS, HTTP://WWW.CLIPARTPANDA.COM/CATEGORIES/HOUSE-ROOF-OUTLINE-CLIPART, HTTP://POLARWINDOWS.COM/DOORS/

In general terms

SA describes (in a more or less “formal” notation) how a system is structured in to **component**s and **connectors**

- Components
- Connectors
- Channels and ports

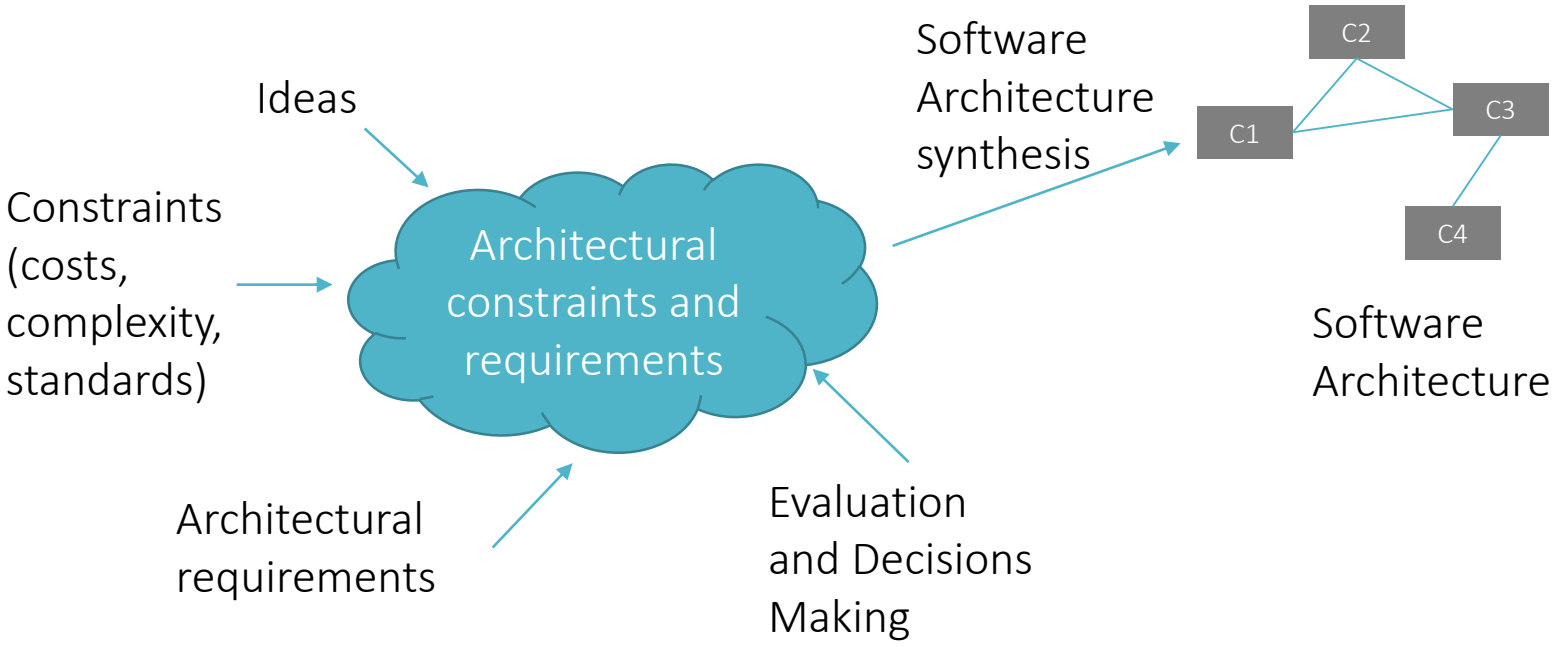
SA structure (topology)

And how these components interact

- Scenarios
- State diagrams
- So on…..

SA Dynamics (behavior)

General work flow



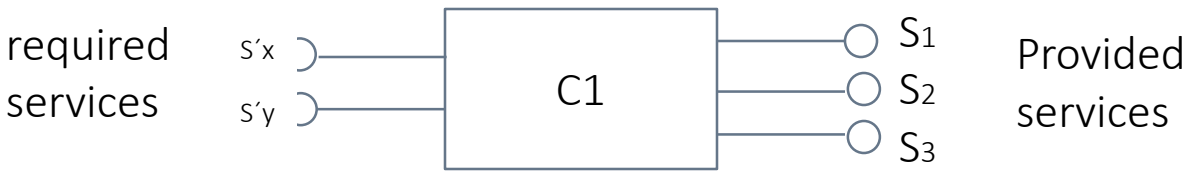
Static Descriptions

- Components
- Connectors
- Interfaces

Components

A component is a building block that is

- Unit of computation or a data store, with an interface specifying the services it provides and requires
- A unit of deployment
- A unit of reuse (e.g. client, server, database, filers,….)



Connectors

A connectors is a building block that enables interaction among components

- Events
- Client/server middleware
- Messages and message buses
- Shared Variables
- Procedure calls (local or remote)
- Pipes

Connector may be implicit or explicit

- Connectors are sometimes just channels
- Connectors sometimes have their own logic and complexity

Components and Connectors

- A component is (or should be) independent of the context in which it us used to provide services

- A Connector is (or should be) dependent on the context in which it is used to connect components

- Connectors sometimes are modeled as special kinds of components.

Interfaces

- A **interface** is the **external connection** of the component (or connector) that describes how to interact with it

- **Provided** and **required** interfaces are important

- Spectrum of interface specification
  - Loosely specified (events go in, events go out)
  - API style(list of functions)
  - Very highly specified (event protocols across the interface)

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

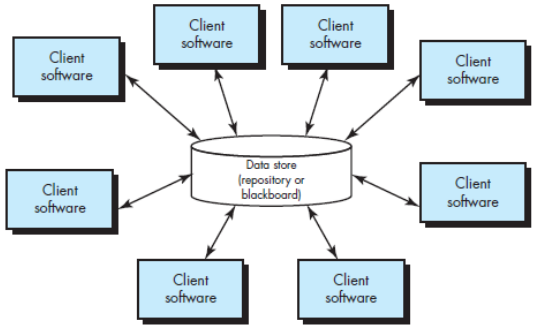
Object-oriented architectures

Layered architectures

Data-Centered Architecture

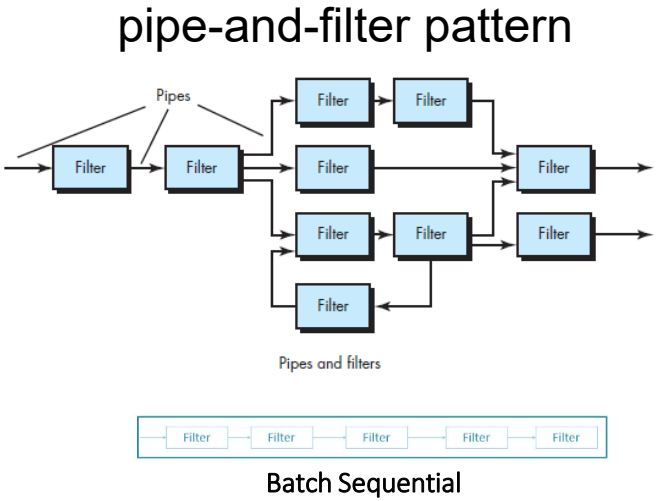
A data store (e.g., a file or database) resides at the center of this architecture

is accessed frequently by other components that update, add, delete, or otherwise modify data within the store.



Integrability : components can be changed and new client components added to the architecture without concern about other clients

Data Flow Architecture



- Applied when input data are to be transformed through a series of computational or manipulative components into output data.

- has a set of components

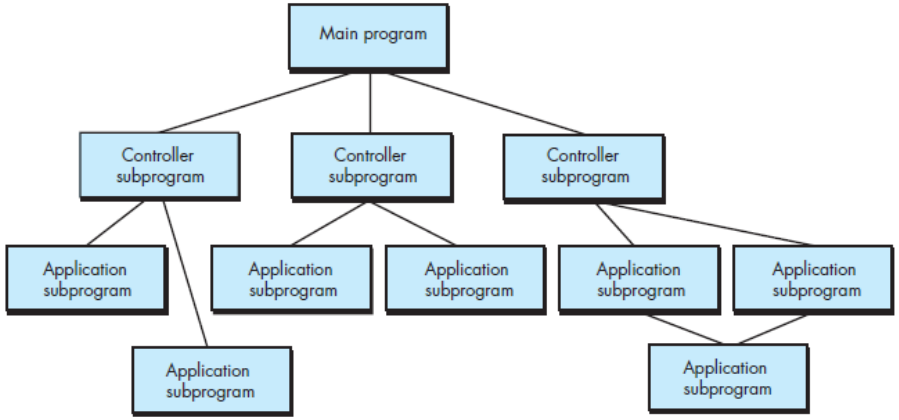
  - Pipe: stateless and they carry binary or character stream which exist between two filters.

  - Filters:
    - an independent data stream transformer or stream transducers.
    - transforms the data of the input data stream, processes it, and writes the transformed data stream over a pipe for the next filter to process.
    - does not require knowledge of the workings of its neighboring filters.

Call and Return Architecture

enables you to achieve a program structure that is relatively easy to modify and scale.

have been the dominant architectural style in large software systems for the past 30 years.



Object-Oriented Architectures

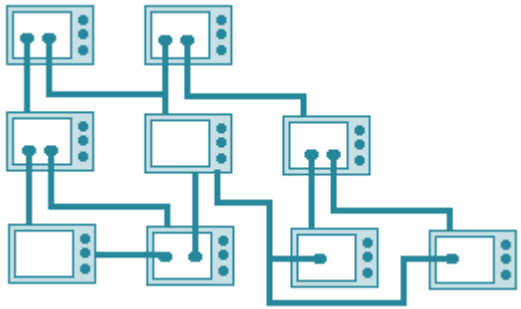
- systems are the modern version of call-and-return architectures.
- the abstract data type paradigm from which it evolved, emphasizes the bundling of data and methods to manipulate and access that data (Public Interface).
- The object abstractions form components that provide black-box services and other components that request those services.
- goal is to achieve the quality of modifiability.

Object-Oriented Architectures

- This bundle is an encapsulation that hides its internal secrets from its environment.
-
- Access to the object is allowed only through provided operations, typically known as methods, which are constrained forms of procedure calls.

- This encapsulation promotes reuse and modifiability, principally because it promotes separation of concerns:

  - The user of a service need not know, and should not know, anything about how that service is implemented.



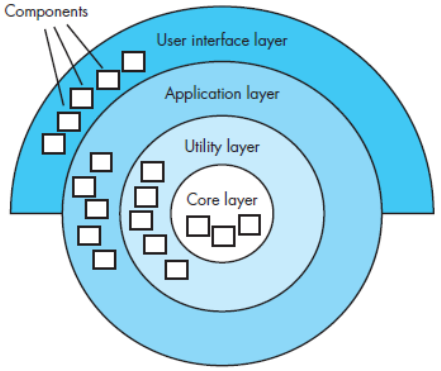
Call and Return Architecture

A number of substyles [Bas03] exist within this category:

- Main program/subprogram architectures:
  - classic program structure
  - goal is to decompose a “main” program in to small pieces
  - is decomposed hierarchically
  - typically a single thread of control and each component in the hierarchy gets this control (optionally along with some data) from its parent and passes it along to its children.

- Remote procedure call architectures:
  - components of a main program/ subprogram architecture are distributed across multiple computers on a network.
  - Goal is to increase performance by distributing the computations and multiple processors.

Layered Architecture



- components are assigned to layers to control intercomponent interaction.
- In the pure version of this architecture, each level communicates only with its immediate neighbors
- goal is to achieve the qualities of modifiability and, usually, portability.
- The lowest layer provides some core functionality, such as hardware, or an operating system kernel.
- Each successive layer is built on its predecessor, hiding the lower layer and providing some services that the upper layers make use of.

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.

Today

- Continue Architecture styles
- Architecture Patterns

Problem

- Arch. documents over-emphasize an aspect of development (i.e. team organization) or do not address the concerns of all stakeholders
- Various stakeholders of software system: end-user, developers, system engineers, project managers
- Software engineers struggled to represent more on one blueprint, and so arch. documents contain complex diagrams

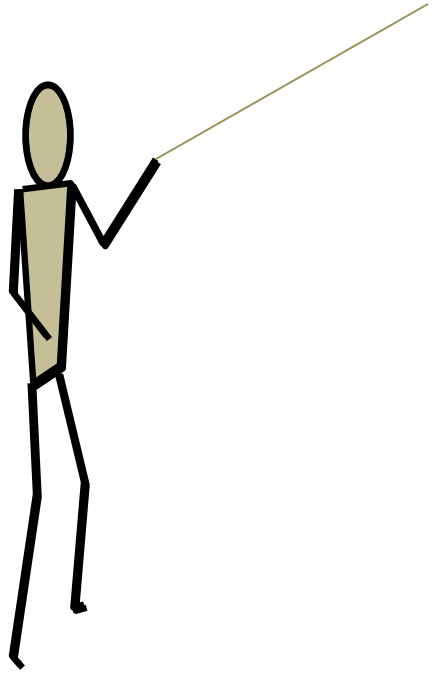
http://cic.javerianacali.edu.co/wiki/lib/exe/fetch.php?media=materias:mazeiar-kruchten-4_1.pdf

Solution

- Using several concurrent views or perspectives, with different notations each one addressing one specific set for concerns
- “4+1” view model presented to address large and challenging architectures

http://cic.javerianacali.edu.co/wiki/lib/exe/fetch.php?media=materias:mazeiar-kruchten-4_1.pdf

Kruchten's 4+1



Kruchten's 4+1 architectural view model

1995: Kruchten, Philippe. Architectural Blueprints — The "4+1" View Model of Software Architecture.

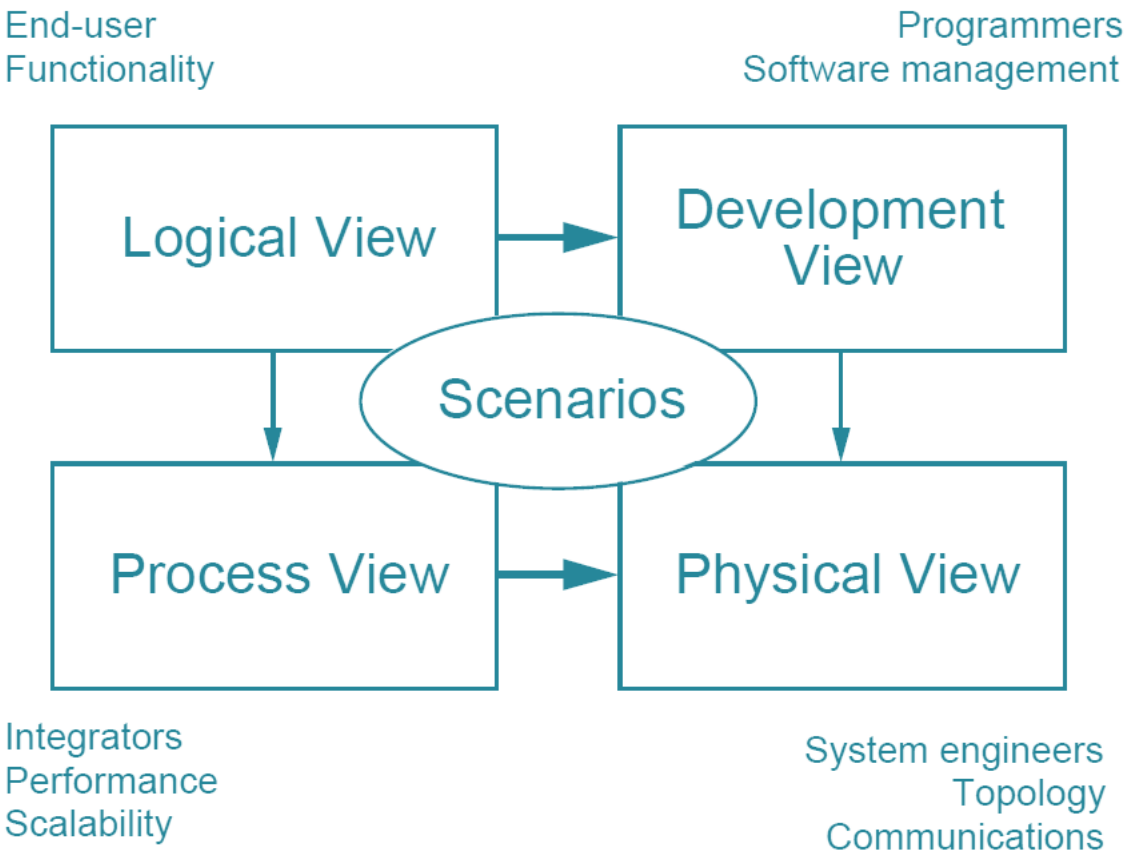
- popularized the "multiple views" idea of different stakeholders

- 2000: IEEE Standard 1471
  - formal conceptual model for architectural descriptions
  - standard terminology
  - distinguish between views and viewpoints

- Be careful: Kruchten's "views" are viewpoints
  - The word "view" is used for historical reasons

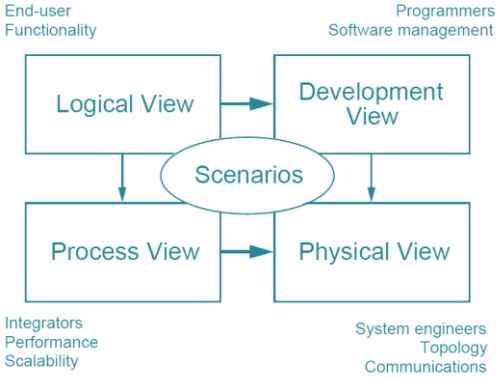
Source: http://www.win.tue.nl/~aserebre/2IW80/2014-2015/A1%20-%20Introduction%20to%20Architecture.pdf

Kruchten's 4+1 architectural view model



Source: http://www.win.tue.nl/~aserebre/2IW80/2014-2015/A1%20-%20Introduction%20to%20Architecture.pdf

Logical view



Viewer: End-User

The user's view on the system, i.e. what a user encounters while using the system

Notation: Object and Dynamic Models

UML diagrams that show the logical view include

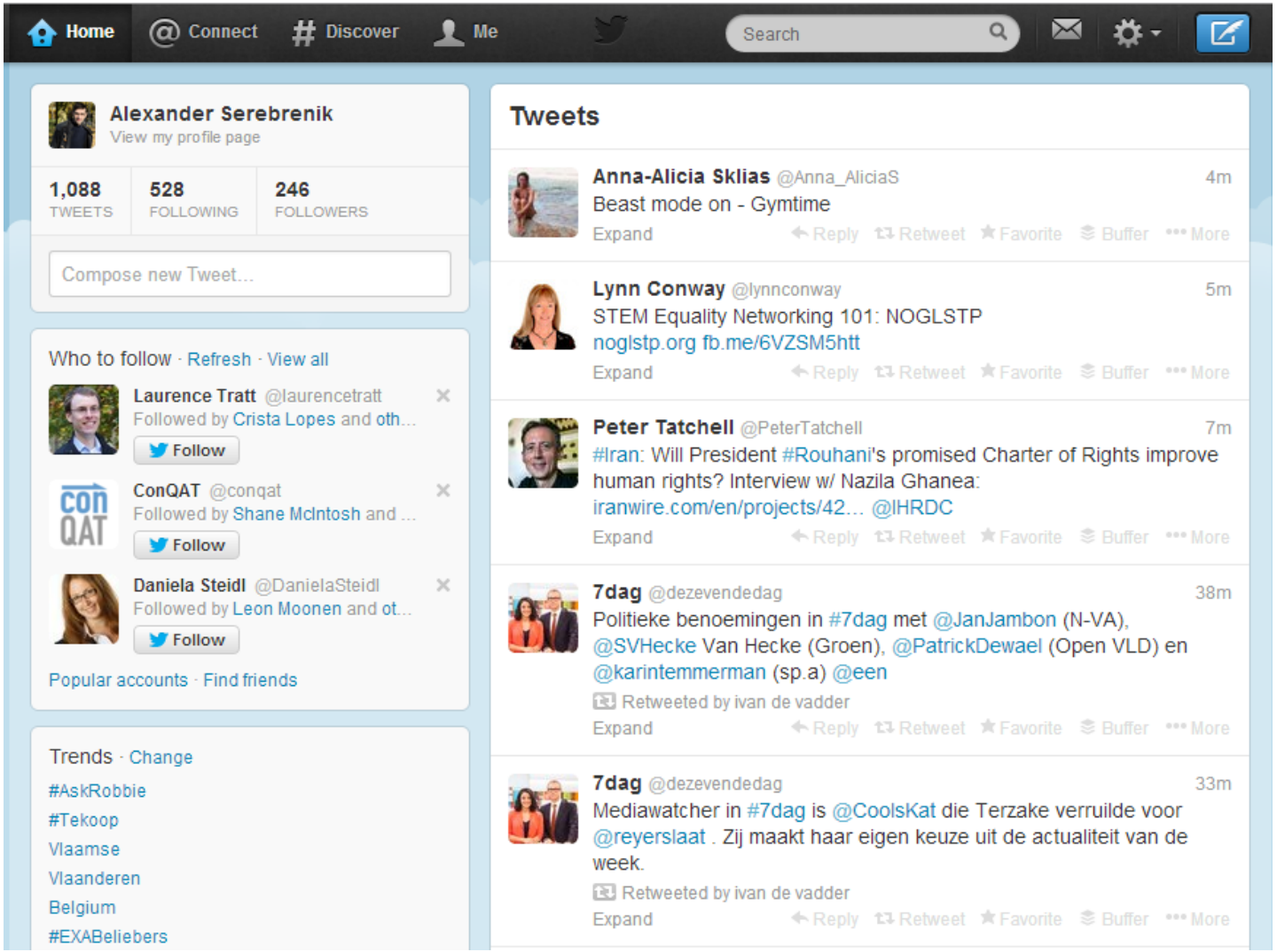
- classes and objects (class instances) documenting user-visible entities
  – including interfaces, that imply responsibilities
- sequence diagrams
- state diagrams: describing state changes as result of interactions

Tools: UML modeling, Rational Rose

Should be consistent with the scenarios

Source: http://www.win.tue.nl/~aserebre/2IW80/2014-2015/A1%20-%20Introduction%20to%20Architecture.pdf

Example: Twitter

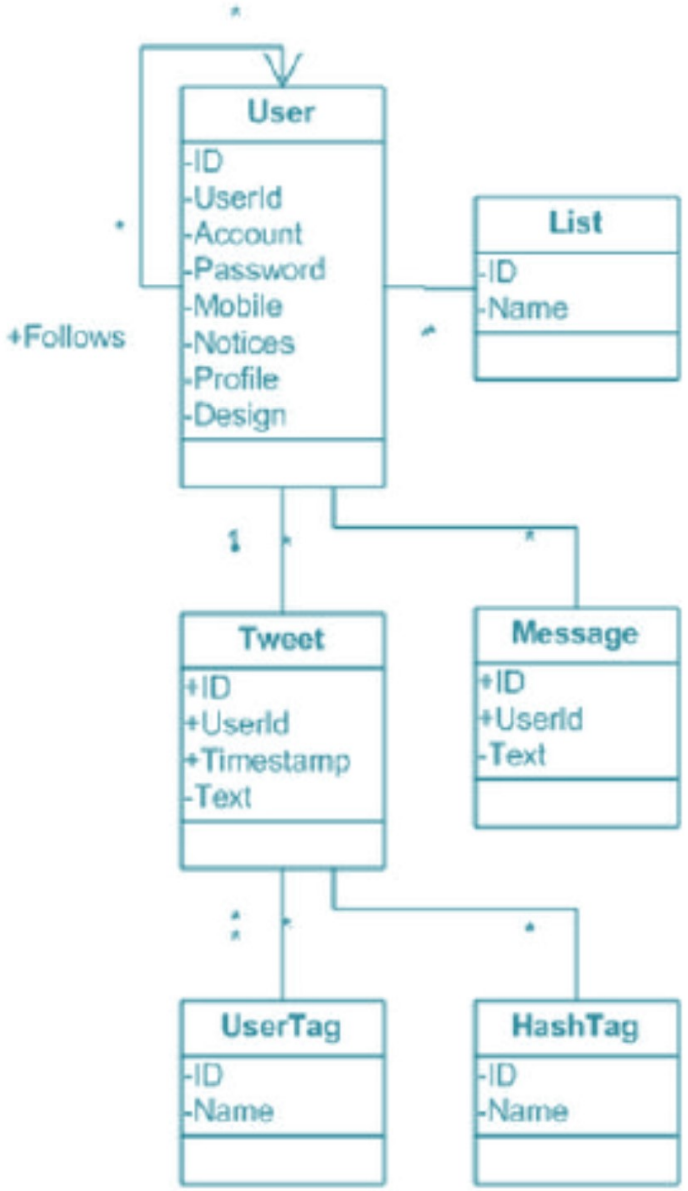


Source: http://www.win.tue.nl/~aserebre/2IW80/2014-2015/A1%20-%20Introduction%20to%20Architecture.pdf

Example: Twitter

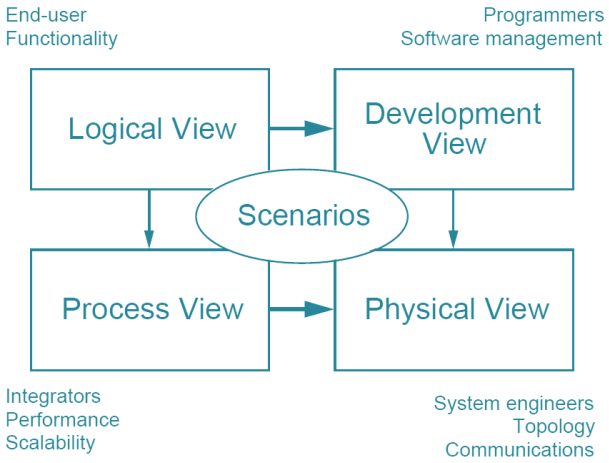
A domain model
- class diagram
- captures concepts from the application domain and their relationships

Additional models will typically be used to represent the logical view.



Source: http://www.win.tue.nl/~aserebre/2IW80/2014-2015/A1%20-%20Introduction%20to%20Architecture.pdf

Development View



Viewers: Programmers and software managers

Considers:

Components, functions, subsystems organized in modules and packages

- Component/module interface descriptions, access protocols

- Logical organization – layering of functionality, dependencies
  - Don't misunderstand the name 'logical'
- Organization into files and folders
- Typical relations: uses, contains, shares, part-of, depends-on

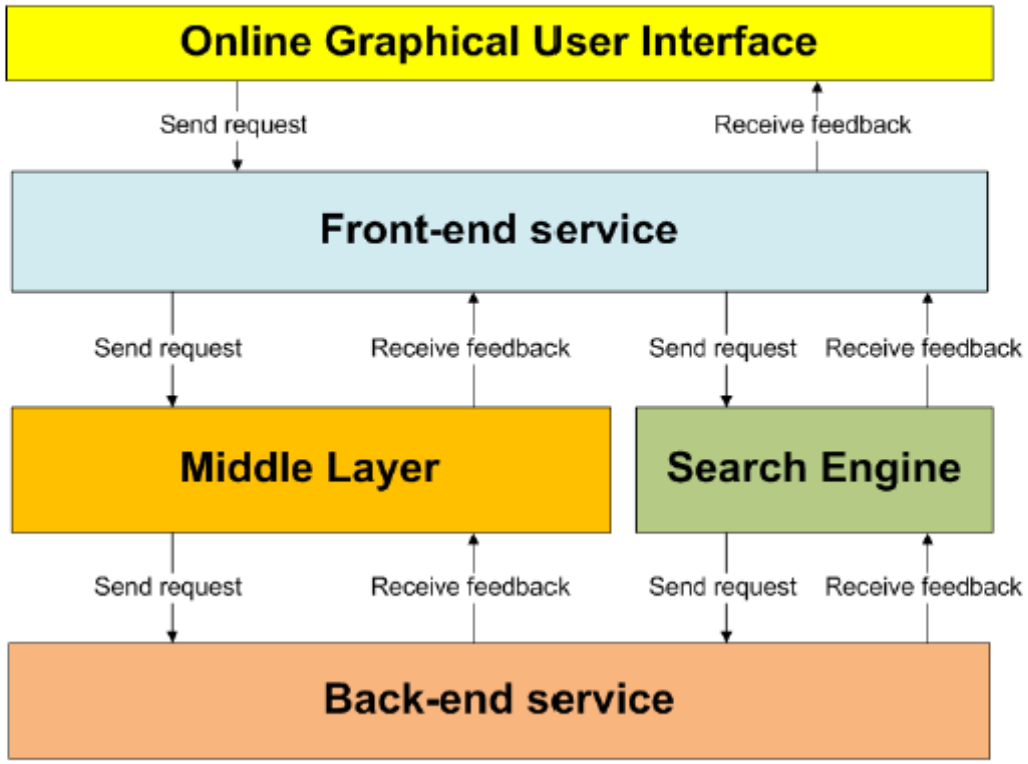
Architecture Style: layered style

Development view should be consistent with the logical view

Source: http://www.win.tue.nl/~aserebre/2IW80/2014-2015/A1%20-%20Introduction%20to%20Architecture.pdf

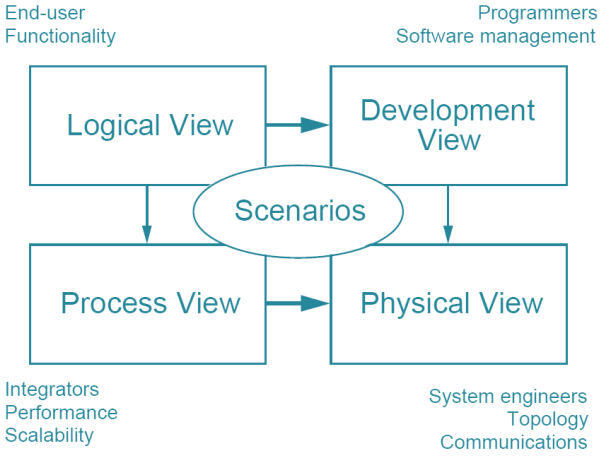
Twitter example: Development View

twitter



Source: http://www.win.tue.nl/~aserebre/2IW80/2014-2015/A1%20-%20Introduction%20to%20Architecture.pdf

Process View



- **Viewers:** Integrators

- describes a system processes
- shows any communication between those processes
- Explores what needs to happen inside the system
- Mapping of applications to distinct memory spaces and units of execution
    - unit of execution: process, thread
    - memory space: associated with a process
- Choice of communication protocols

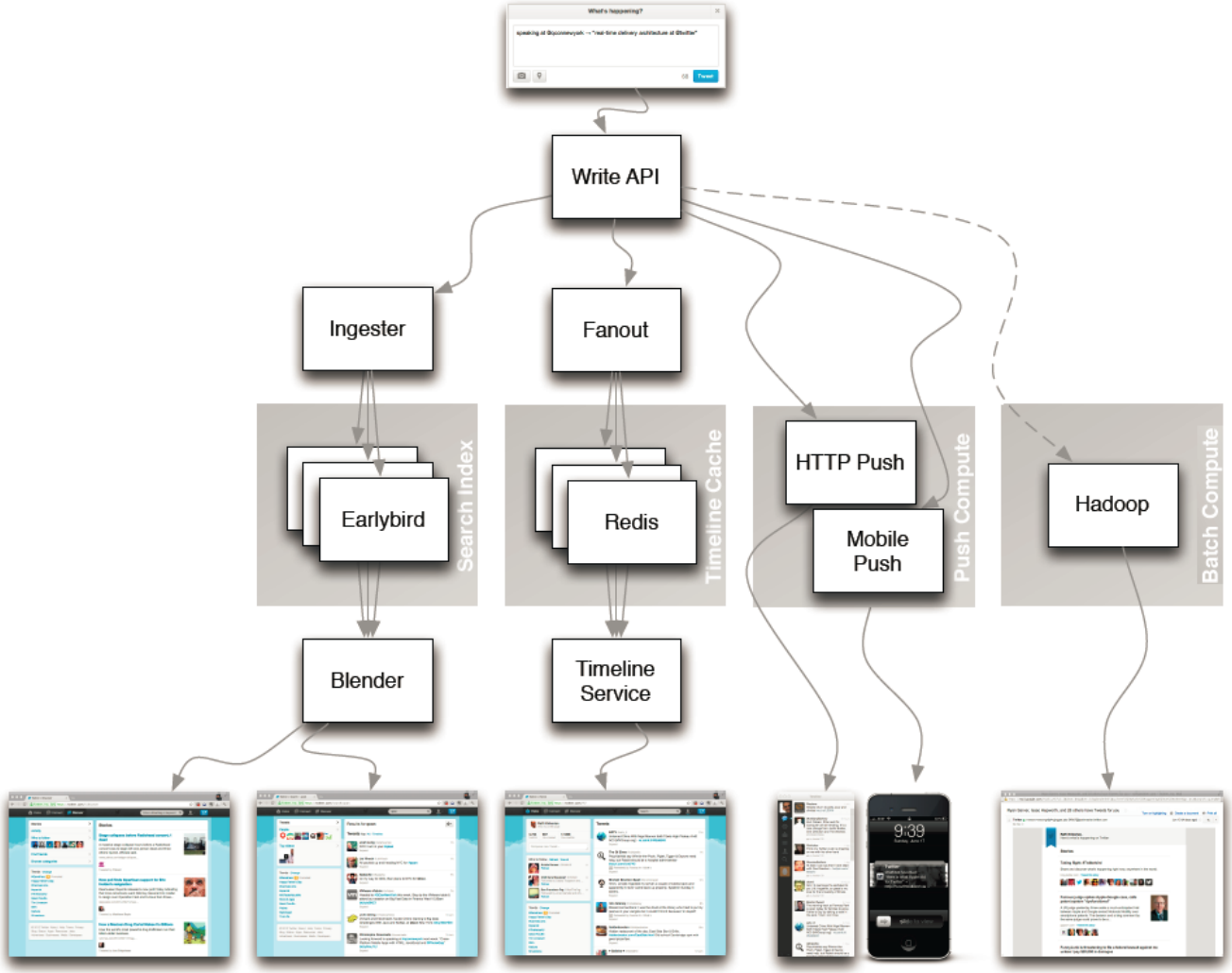
Considers:

- Scheduling of activities such as to satisfy **time** and resource constraints
    - Performance

UML **activity diagram** represent the process view

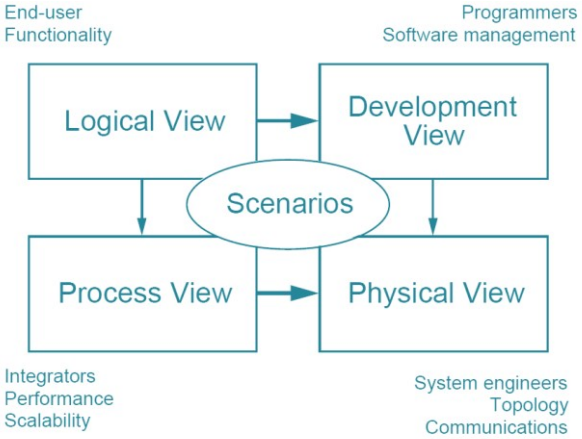
Source: http://www.win.tue.nl/~aserebre/2IW80/2014-2015/A1%20-%20Introduction%20to%20Architecture.pdf

Twitter example: Process View



https://www.slideshare.net/raffikrikorian/qcon-nyc-2012-twitters-real-time-architecture/30-Write_APIIngester_Fanout_Batch_Compute

Physical View



Viewer: System Engineers

Machines (processors, memories), networks, connections
- including specifications, e.g. speeds, sizes

- **Deployment**: mapping of elements of other views to machines

- Typical relations: connects-to, contains, maps-to

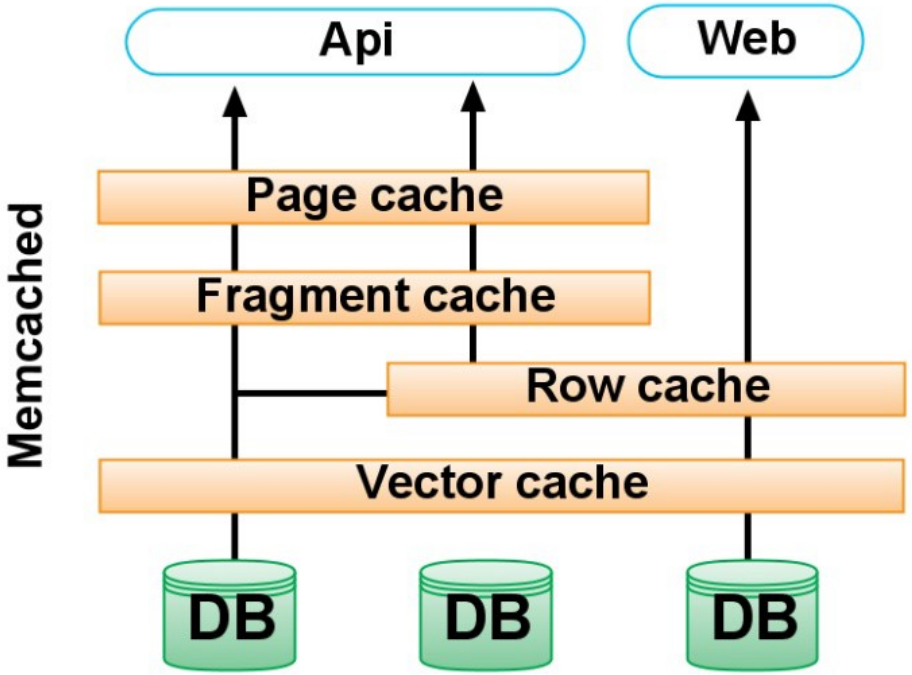
- Concerns: performance (throughput, latency), availability, reliability, etc., together with the process view

Source: http://www.win.tue.nl/~aserebre/2IW80/2014-2015/A1%20-%20Introduction%20to%20Architecture.pdf

Twitter example: Physical View

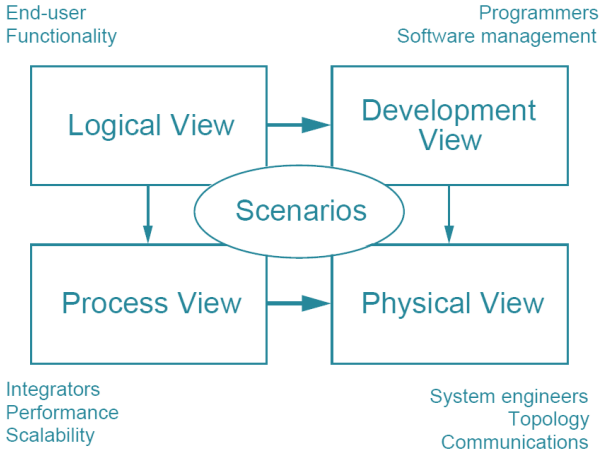
Not very convincing
- No physical components

- It is hard to find a twitter deployment model



http://www.win.tue.nl/~aserebre/2IW80/2014-2015/A1%20-%20Introduction%20to%20Architecture.pdf

Scenarios



Putting it all together

Viewer: All users of other views and Evaluators.

Considers: System consistency, validity

Notation: almost similar to logical view

Tool: Rational Rose

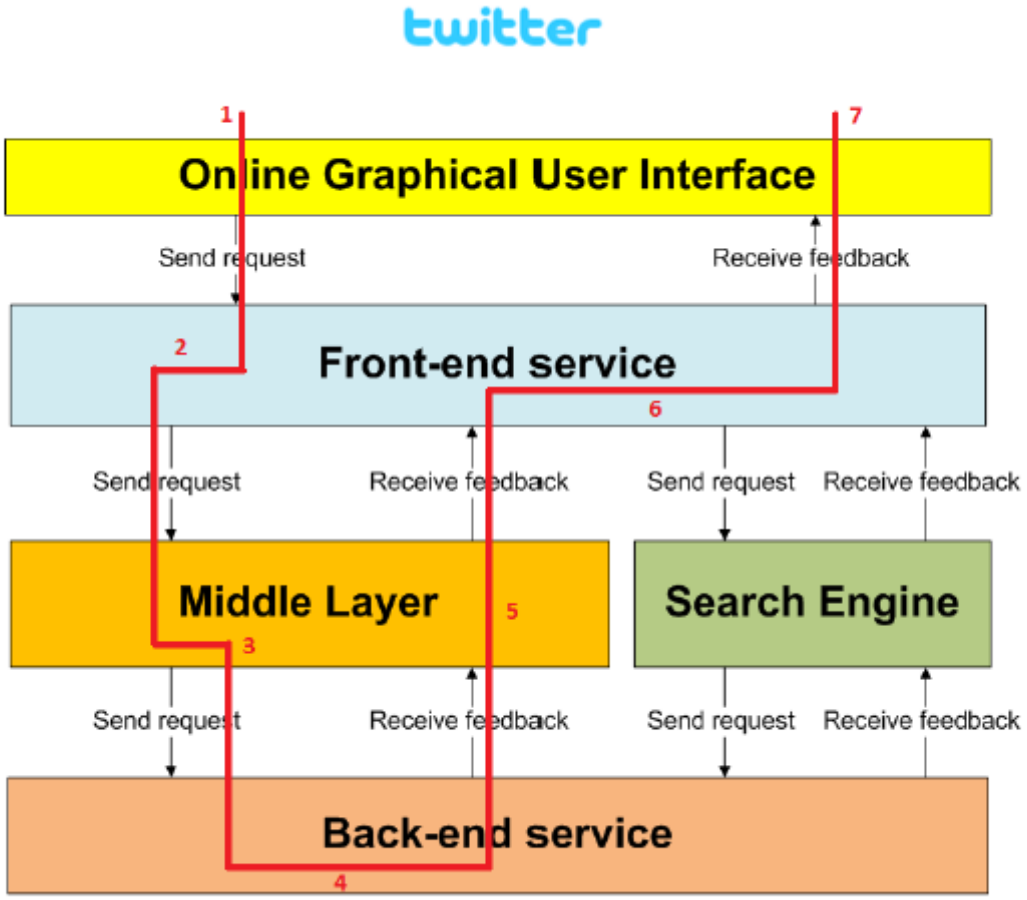
- Help illustrate and validate the document
- Help Architect during the architecture design

Source: http://www.win.tue.nl/~aserebre/2IW80/2014-2015/A1%20-%20Introduction%20to%20Architecture.pdf

Scenarios

1. User:
    a) Logon to GUI
    b) Send tweet

2. Process the insertion request
3. Put the request in the queue system
4. Back-end
    a) Get request from front of queue
    b) Insert tweet into memory.
    c) Update the search engine with an index entry
5. Send feedback to queuing system
6. Notify GUI



Source: http://www.win.tue.nl/~aserebre/2IW80/2014-2015/A1%20-%20Introduction%20to%20Architecture.pdf

The Iterative process

Not all software arch. Need all views.

- A scenario-driven approach to develop the
- system
- Documentation:
- Software architecture document
- Software design guidelines

Source: http://www.win.tue.nl/~aserebre/2IW80/2014-2015/A1%20-%20Introduction%20to%20Architecture.pdf

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

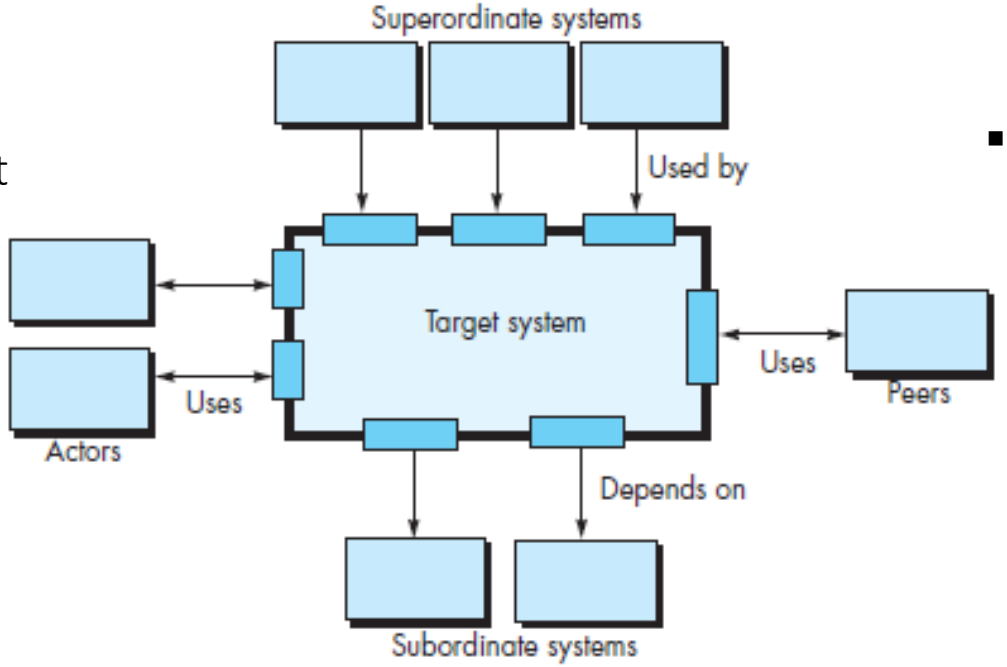
Representing the system in Context

Software architect uses an architectural context diagram (ACD) to model how software interacts with entities external to its boundaries.

Architectural Context

- those systems that use the target system as part of some higher-level processing scheme.

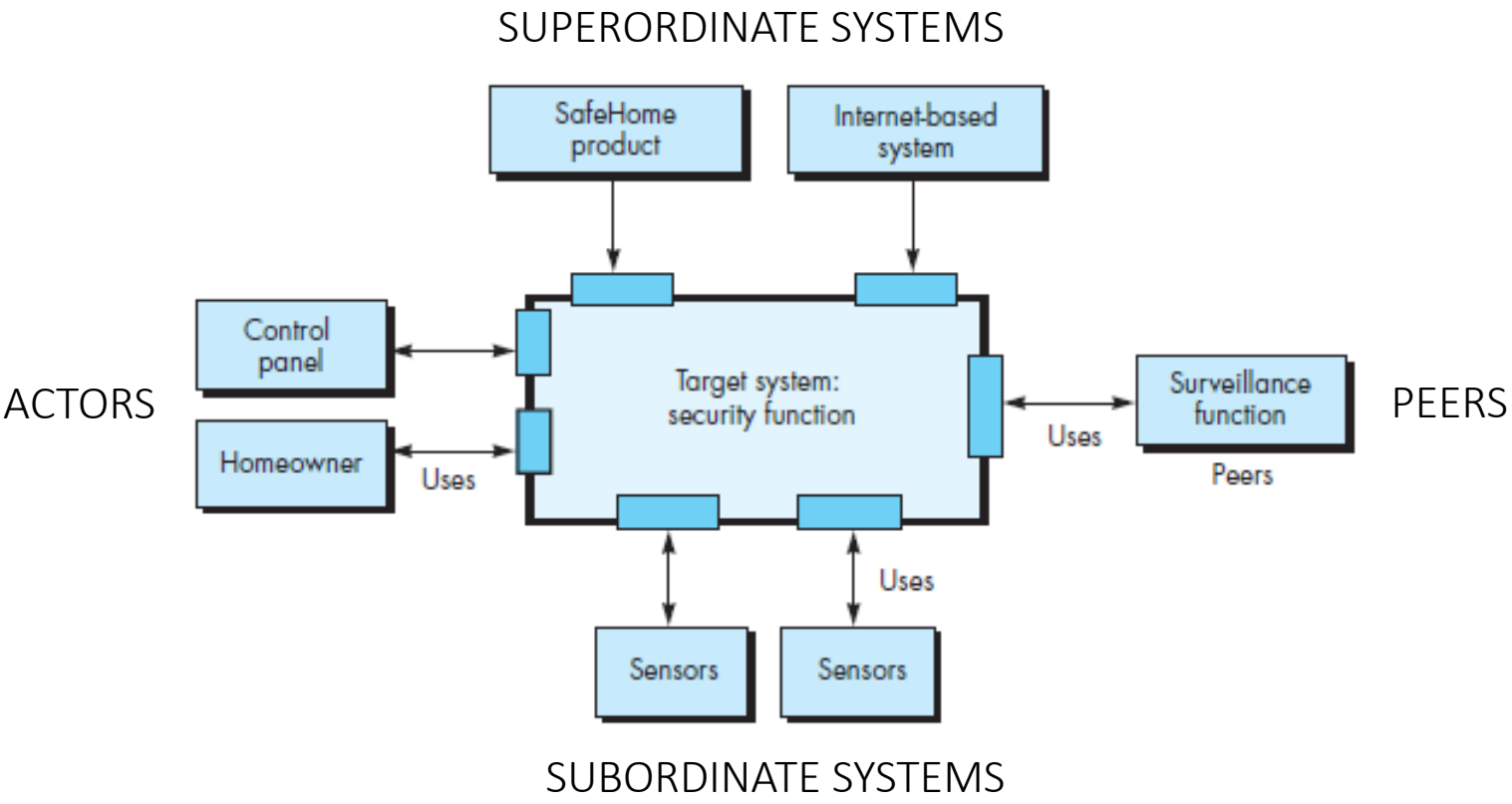
- entities (people, devices) that interact with the target system by producing or consuming information that is necessary for requisite processing.



- 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.

- those systems that are used by the target system and provide data or processing that are necessary to complete target system functionality.

Example: Architectural Context



Archetypes

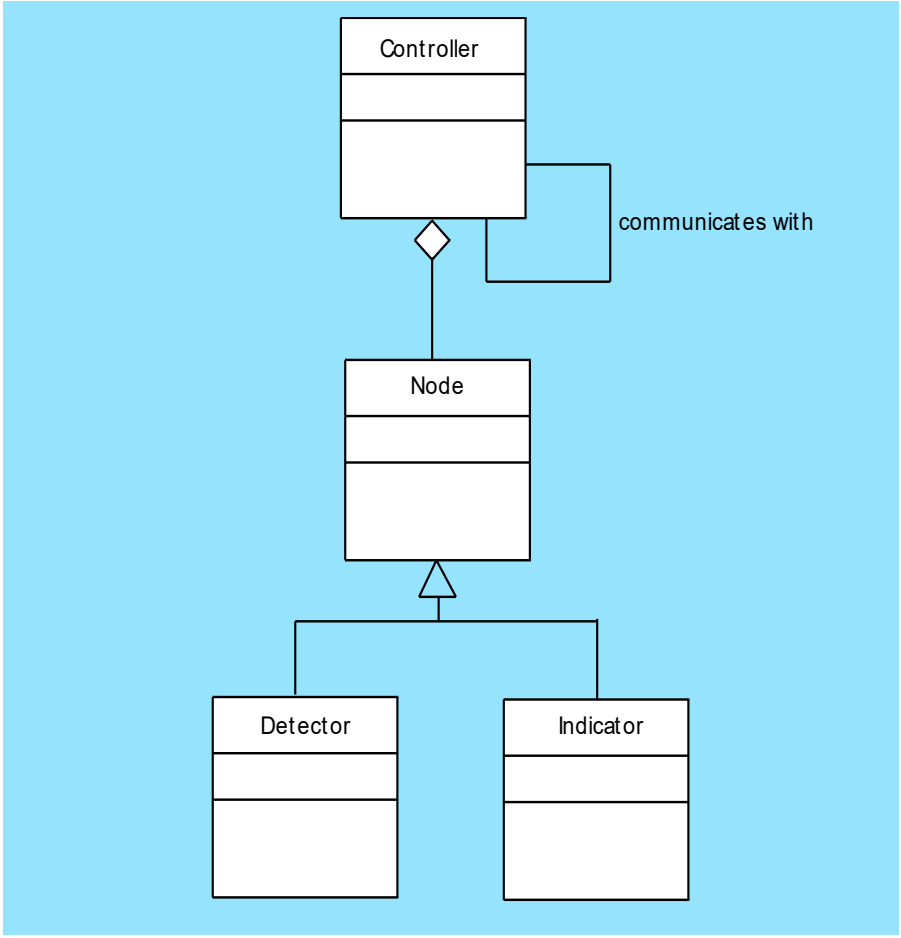
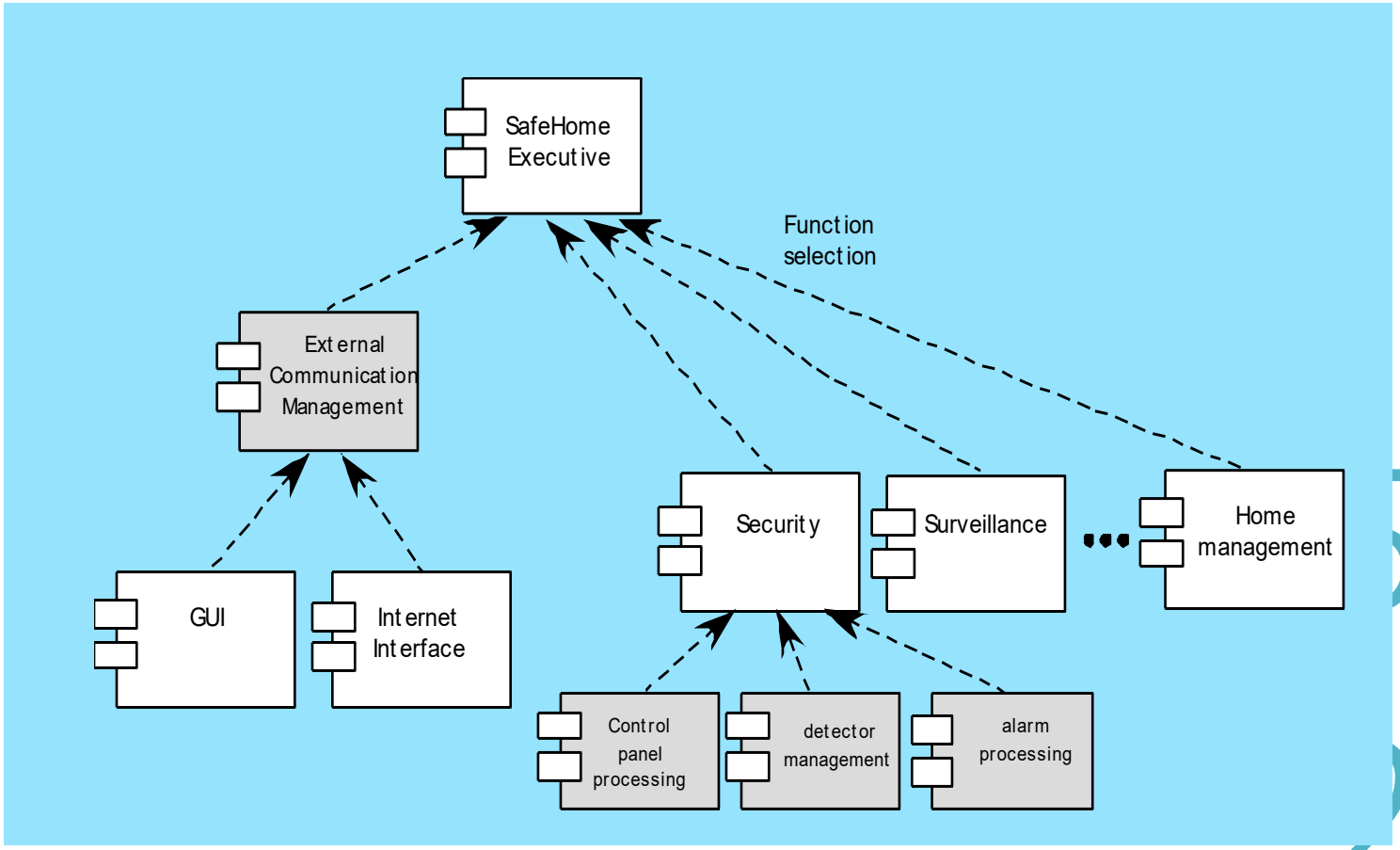
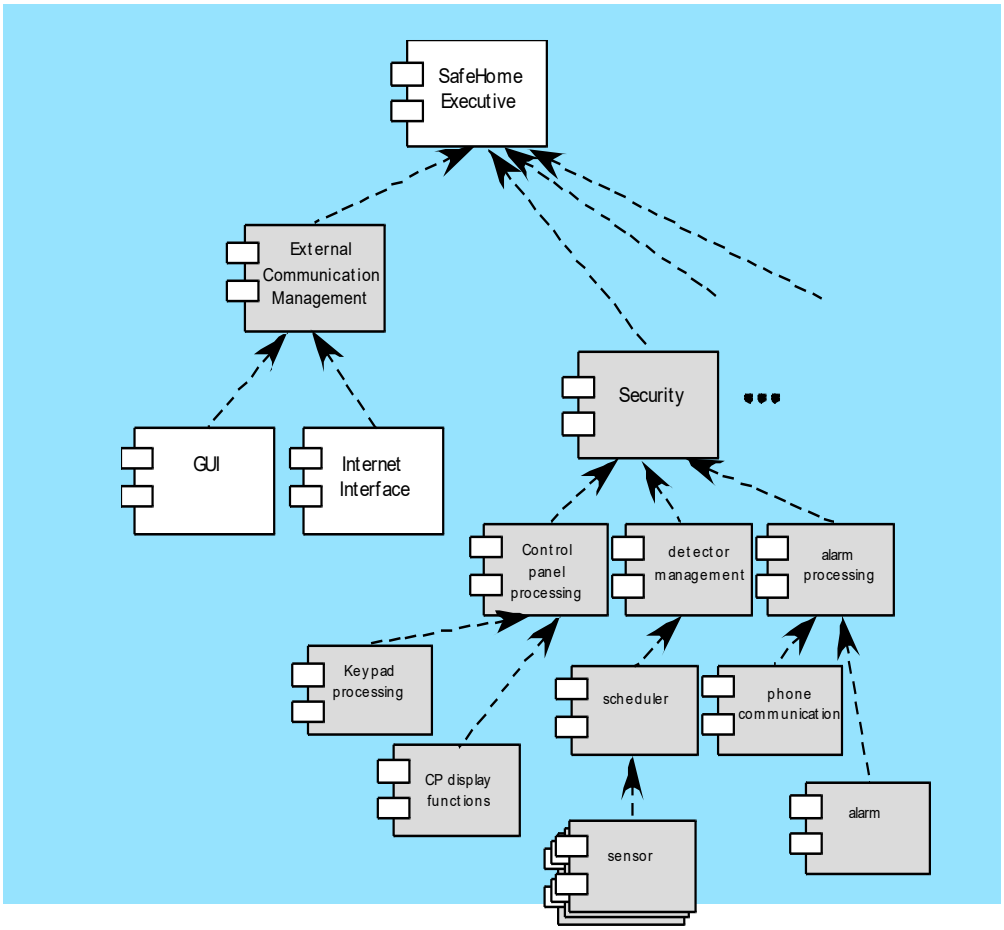


Figure 10.7 UML relationships for SafeHome security function archetypes (adapted from [BOS00])

Component Structure



Refined Component Structure



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

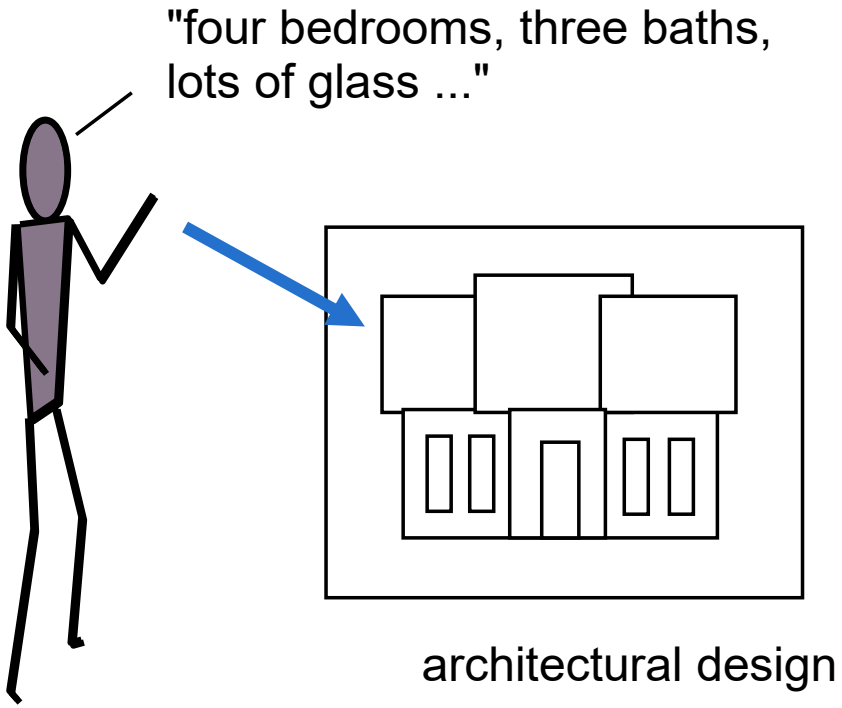
Architectural description language (ADL) provides a semantics and syntax for describing a software architecture

Provide the designer with the ability to:

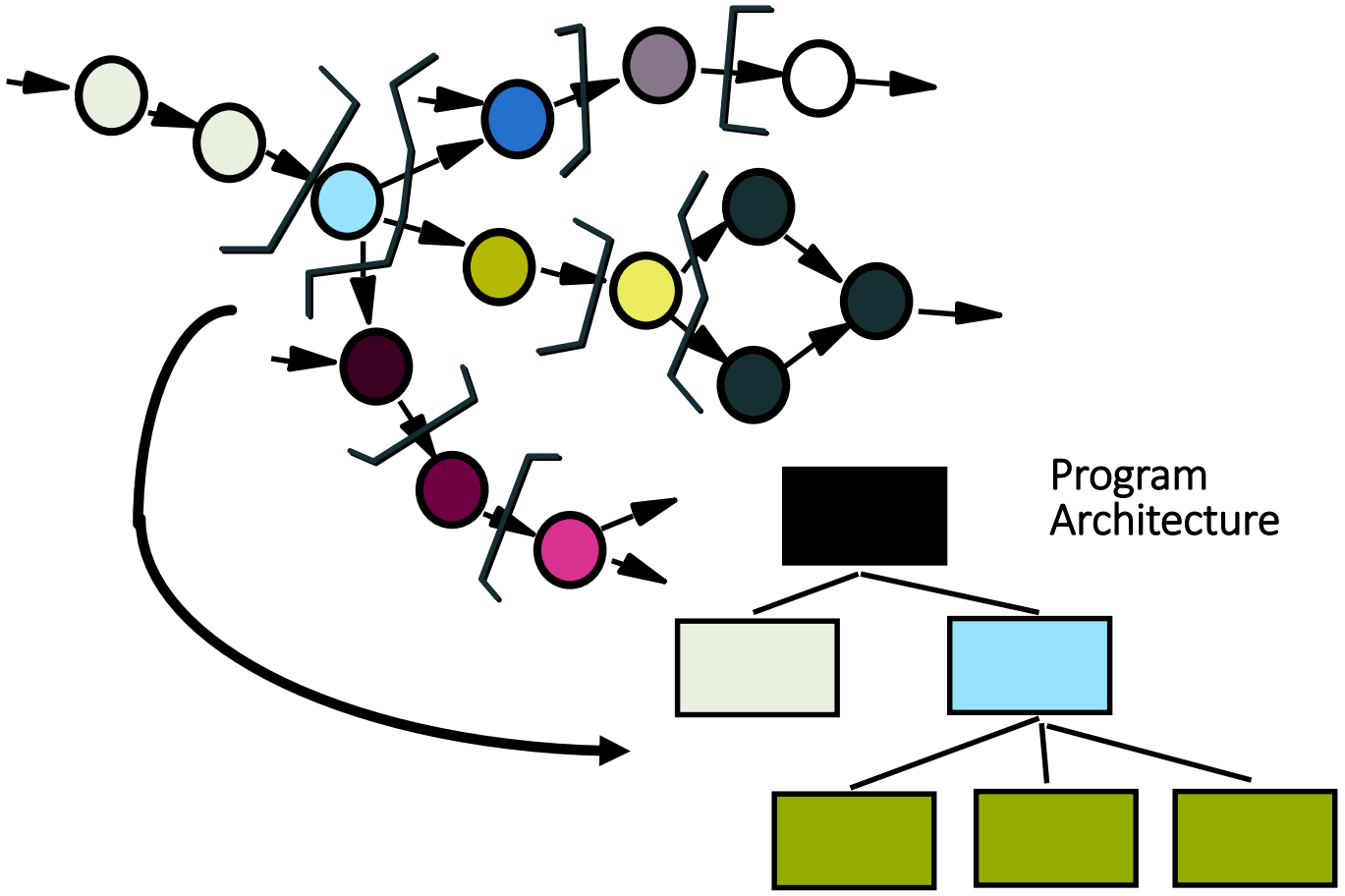
- decompose architectural components
- compose individual components into larger architectural blocks and
- represent interfaces (connection mechanisms) between components.

An Architectural Design Method

customer requirements

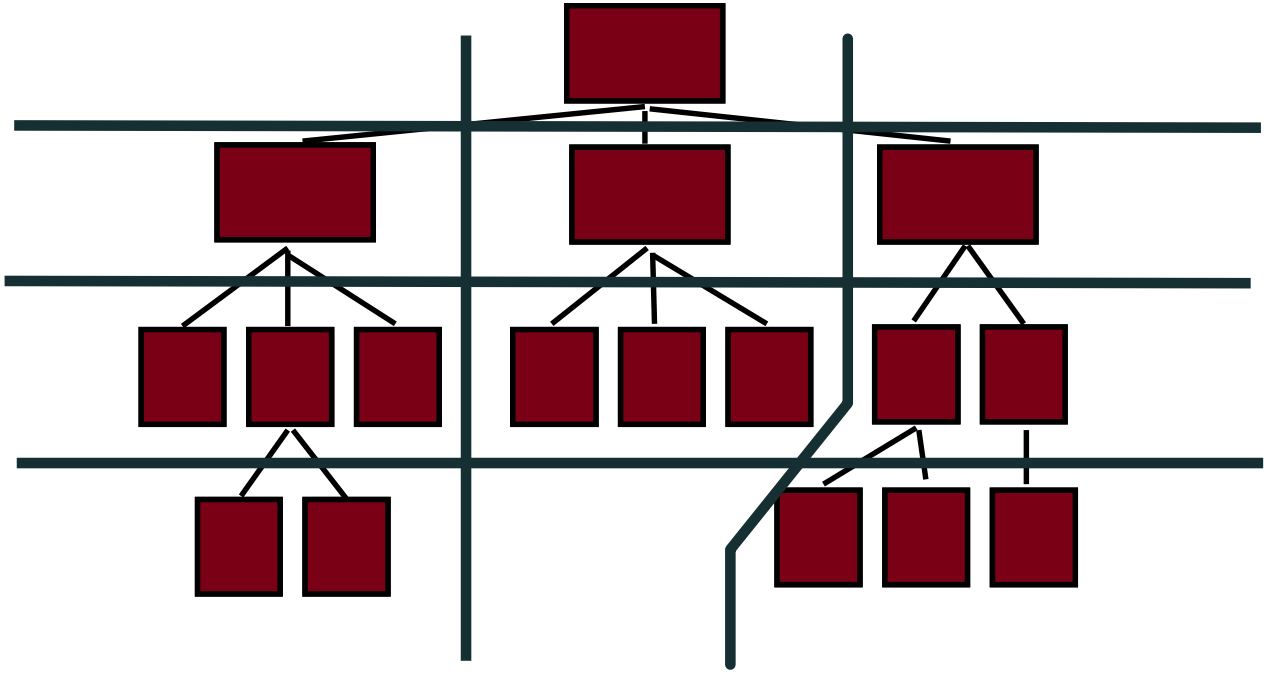


Deriving Program Architecture



Partitioning the Architecture

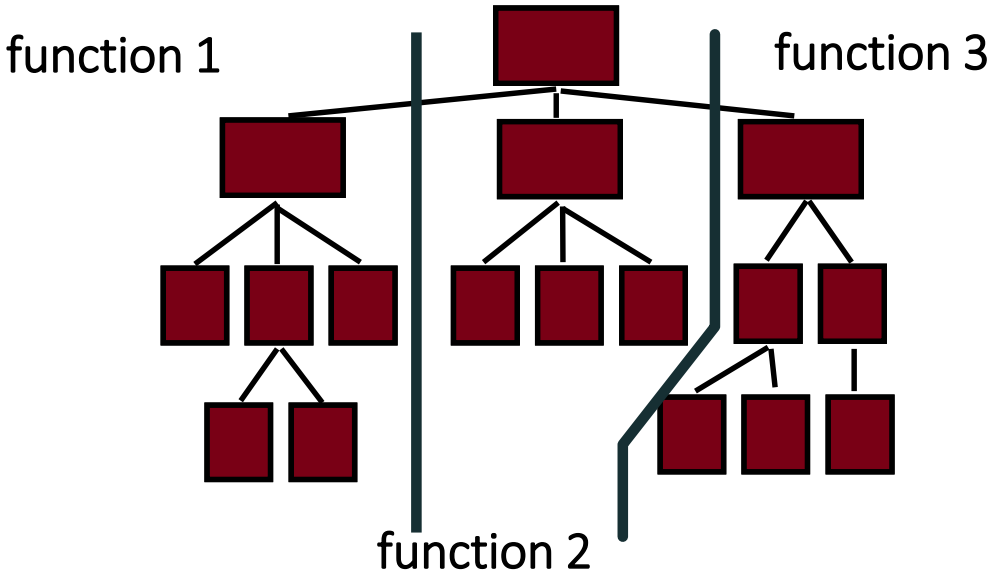
"horizontal" and "vertical" partitioning are required



Horizontal Partitioning

define separate branches of the module hierarchy for each major function

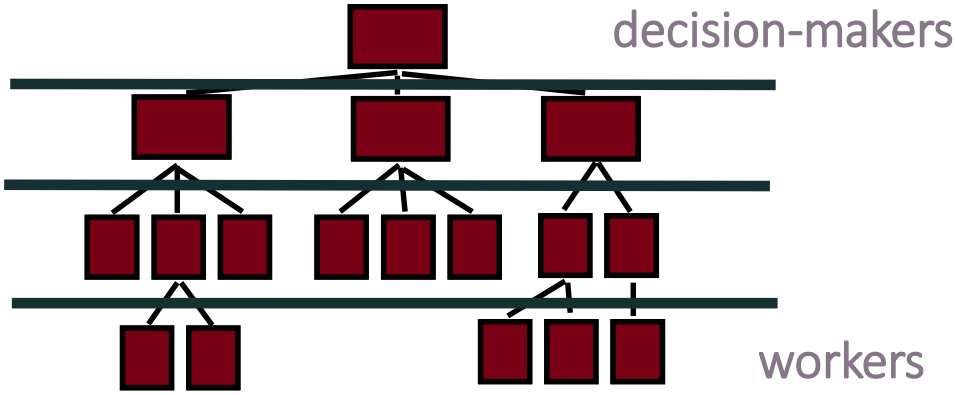
use control modules to coordinate communication between functions



Vertical 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

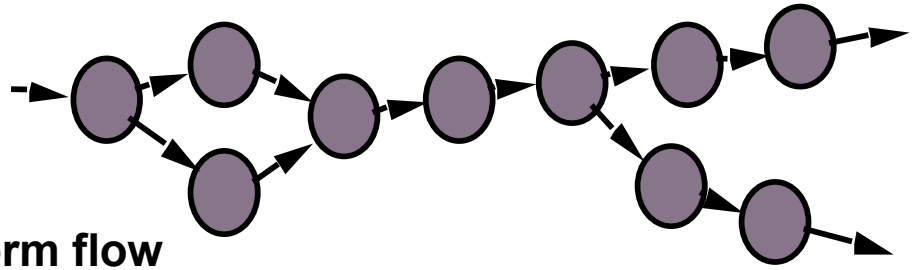
approach:

a DFD is mapped into a program architecture

the PSPEC and STD are used to indicate the content of each module

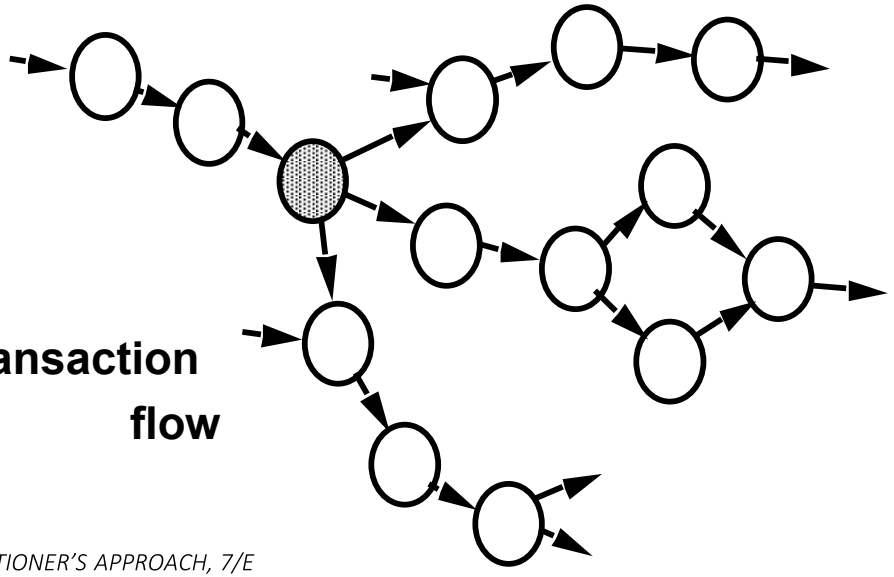
notation: structure chart

Flow Characteristics



Transform flow

This edition of SEPA does not cover transaction mapping. For a detailed discussion see the SEPA website



Transaction flow

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.

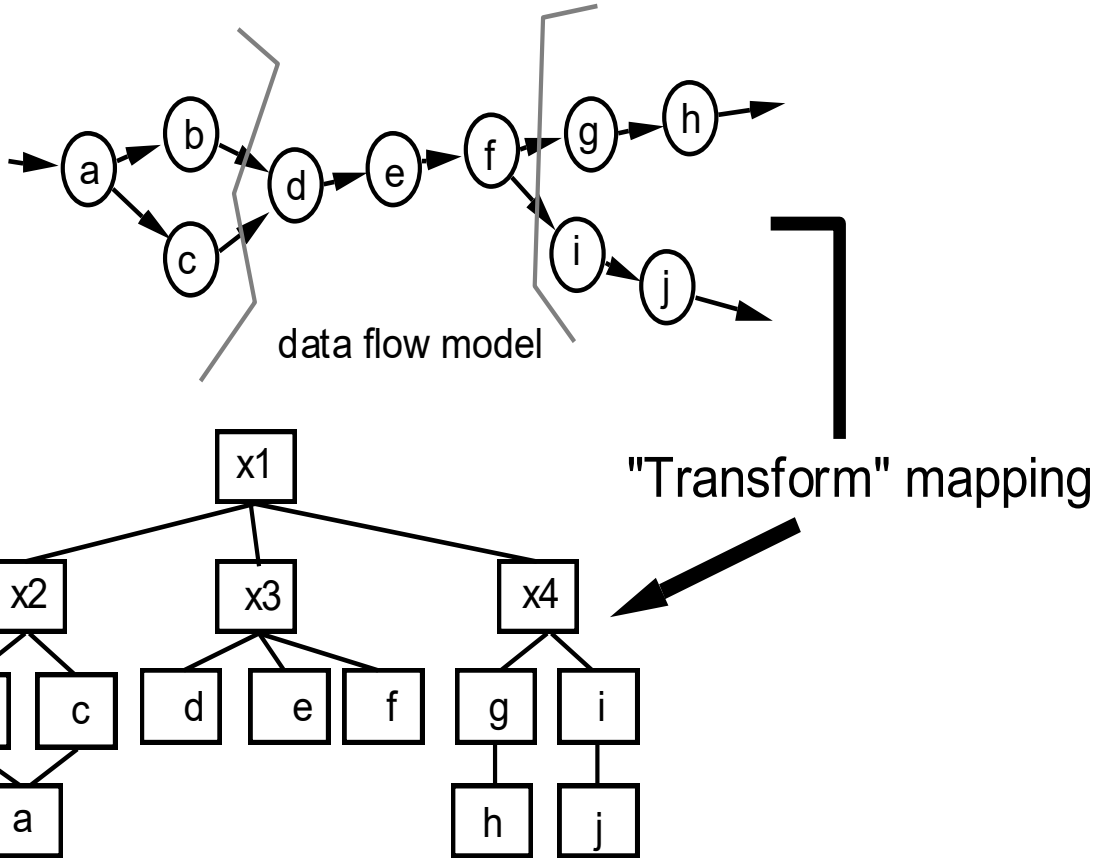
Factoring leads to a program structure in which top-level components perform decision-making and low-level components perform most input, computation, and output work.

Middle-level components perform some control and do moderate amounts of work.

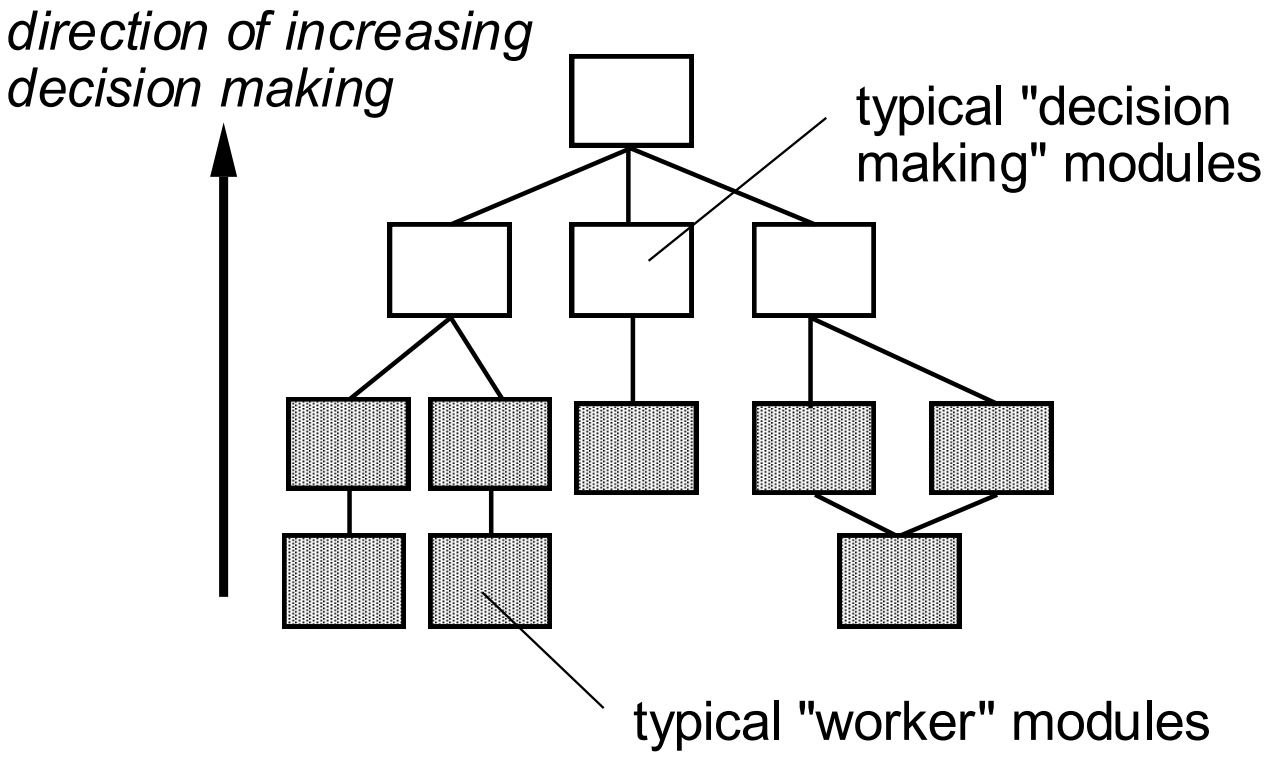
Perform "second-level factoring."

6

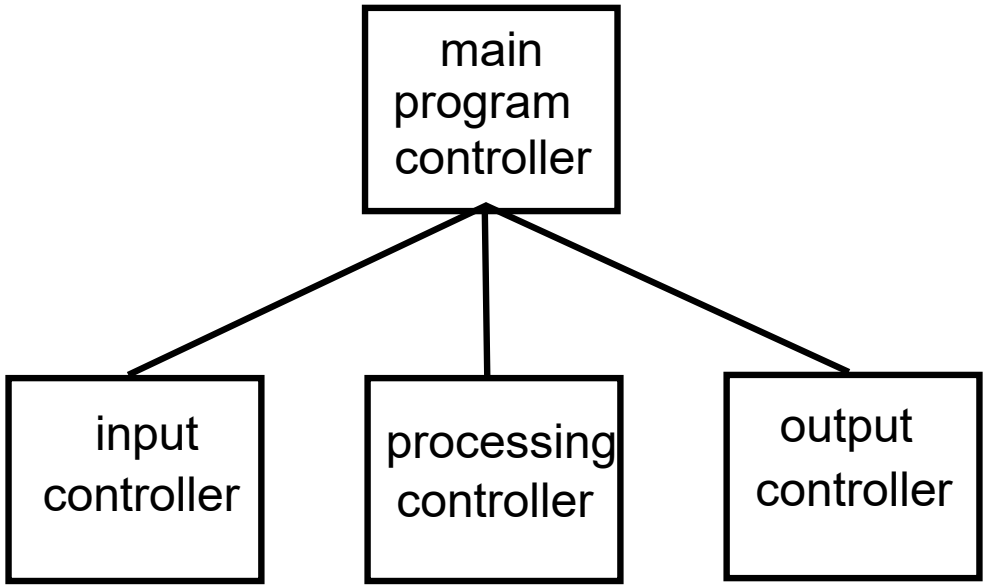
Transform Mapping



Factoring



First Level Factoring



Second Level Mapping