Architectural Design

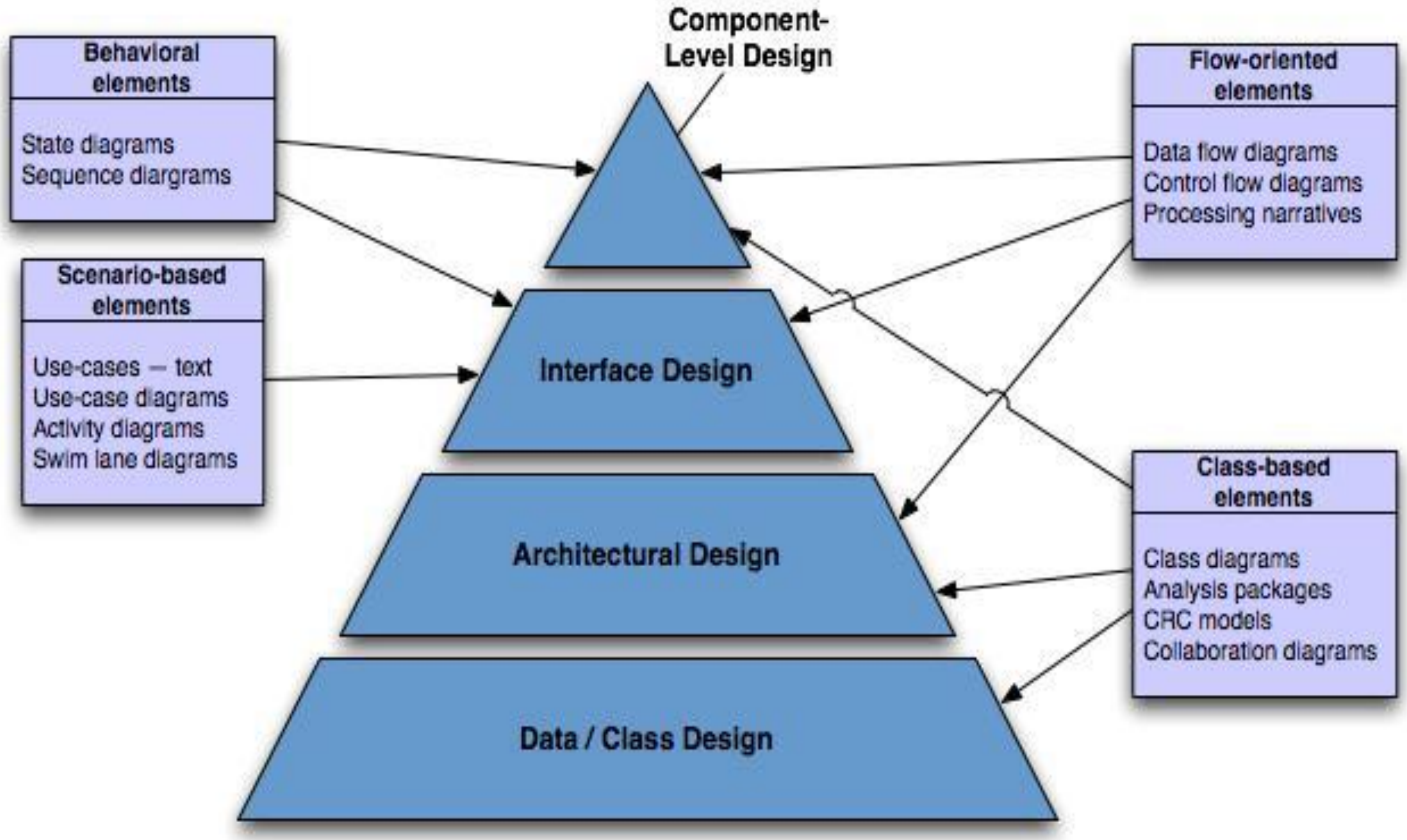
Some of the slides are taken from other sources. Those are explicitly indicated

Analysis Model to Design Model
(Review)

Each element of the analysis model provides information that is necessary to create the four design models

- The data/class design transforms analysis classes into design classes along with the data structures required to implement the software
- The architectural design defines the relationship between major structural elements of the software; architectural styles and design patterns help achieve the requirements defined for the system
- The interface design describes how the software communicates with systems that interoperate with it and with humans that use it
- The component-level design transforms structural elements of the software architecture into a procedural description of software components

Analysis → Design
(Review)



Definitions

- The <u>software architecture</u> of a program or computing system is the structure or structures of the system which <u>comprise</u>
  - The software <u>components</u>
  - The externally visible <u>properties</u> of those components
  - The <u>relationships</u> among the components
- <u>Software architectural design</u> represents the <u>structure</u> of the data and program <u>components</u> that are required to build a computer-based system
- An architectural design model is <u>transferable</u>
  - It can be <u>applied</u> to the design of other systems
  - It <u>represents</u> a set of <u>abstractions</u> that enable software engineers to describe architecture in <u>predictable</u> ways

Importance of Software Architecture

- Representations of software architecture are an enabler for communication between all stakeholders interested in the development of a computer-based system

- The software 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

- The software architecture constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together

Emphasis on Software Components

- A software architecture enables a software engineer to
  - Analyze the effectiveness of the design in meeting its stated requirements
  - Consider architectural alternatives at a stage when making design changes is still relatively easy
  - Reduce the risks associated with the construction of the software
- Focus is placed on the software component
  - A program module
  - An object-oriented class
  - A database
  - Middleware

Uses of software architecture descriptions

- **Understanding and communication:** An architecture description communicates the architecture to various stakeholders, including
  - users who will use the system,
  - clients who commissioned the system,
  - builders who will build the system, and,
  - other architects.
- **Reuse:** Architecture descriptions can help software reuse. The software engineering world has, for a long time, been working towards a discipline where software can be assembled from parts that are developed by different people and are available for others to use.
- **Construction and Evolution:** As architecture partitions the system into parts, some architecture provided partitioning can naturally be used for constructing the system, which also requires that the system be broken into parts such that different teams (or individuals) can separately work on different parts.
- **Analysis:** It is highly desirable if some important properties about the behaviour of the system can be determined before the system is actually built. This will allow the designers to consider alternatives and select the one that will best suit the needs.

An Integrated Approach to Software Engineering, Third Edition by Pankaj Jalote Springer © 2005

Architectural Design Process

- Basic Steps
  - <u>Creation</u> of the data design
  - <u>Derivation</u> of one or more representations of the <u>architectural structure</u> of the system
  - <u>Analysis</u> of alternative <u>architectural styles</u> to choose the one best suited to customer requirements and quality attributes
  - <u>Elaboration</u> of the architecture based on the selected architectural style

- A <u>database designer</u> creates the data architecture for a system to represent the data components

- A <u>system architect</u> selects an appropriate architectural style derived during system engineering and software requirements analysis

Architecture addresses Non-functional Requirements

Non-functional requirements are the ones that don't appear in use cases. Rather than define *what* the application does, they are concerned with *how* the application provides the required functionality. There are three distinct areas of nonfunctional requirements that can impact architecture

- *Technical constraints*: Constrain design options by specifying certain technologies the application must use. "We only have Java developers, so we must develop in Java". "The existing database runs on Windows XP only".
- *Business constraints*: These constraint design options for business reasons. For example, "In order to widen our potential customer base, we must interface with XYZ tools". Another example is "The supplier of our middleware has raised prices prohibitively, so we're moving to an open source version".
- *Quality attributes*: These define an application's requirements in terms of scalability, availability, ease of change, portability, usability, performance, and so on. Quality attributes address issues of concern to application users, as well as other stakeholders like the project team itself or the project sponsor.

Gorton, Ian. Essential Software Architecture, Second Edition. Springer. © 2011

Software Architectural Style

- The software that is built for computer-based systems exhibit one of many architectural styles

- Each style describes a system category that encompasses
  - A set of component types that perform a function required by the system
  - A set of connectors (subroutine call, remote procedure call, data stream, socket) that enable communication, coordination, and cooperation among components
  - Semantic constraints that define how components can be integrated to form the system
  - A topological layout of the components indicating their runtime interrelationships

(Source: Bass, Clements, and Kazman. *Software Architecture in Practice*. Addison-Wesley, 2003)

(Architectural) Style versus Pattern

- Architectural Style is a transformation that is imposed on the design of an entire system.
- Architectural Pattern also imposes a transformation on the design, but differs from style in following ways:
  - The scope is less broad; focus on one aspect
  - Imposes a rule on architecture w.r.t. some functionality e.g. concurrency
  - Address specific behavioral issues e.g. synchronization or interrupt handling

Architectural Patterns (examples)

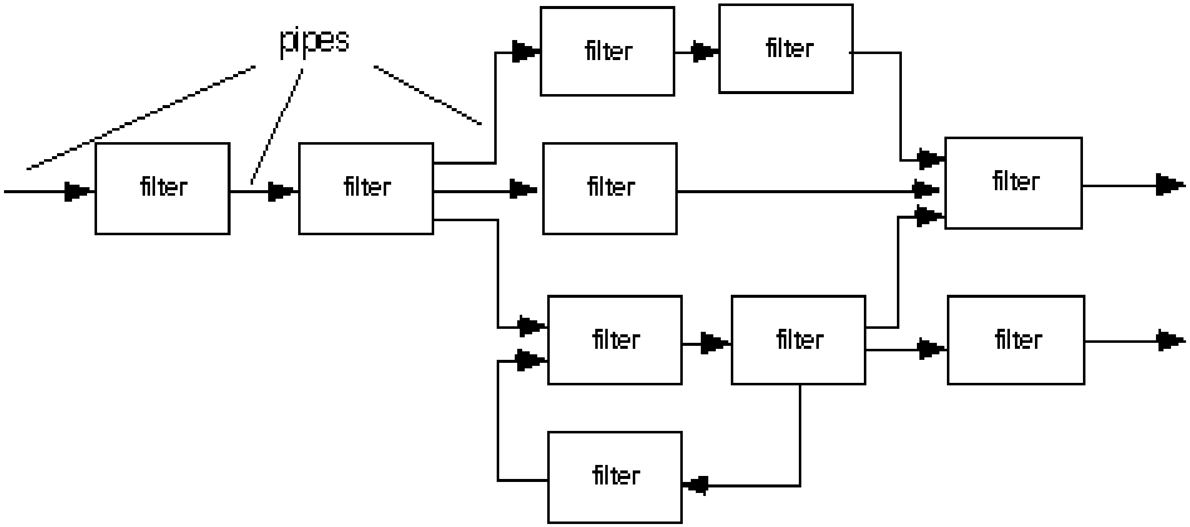
- Concurrency—applications must handle multiple tasks in a manner that simulates parallelism
  - *operating system process management* pattern
  - *task scheduler* pattern
- Persistence—Data persists if it survives past the execution of the process that created it. Two patterns are common:
  - a *database 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 Styles

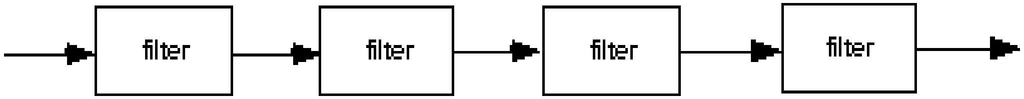
Can be considered **semantic models** that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

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

Data Flow Architecture

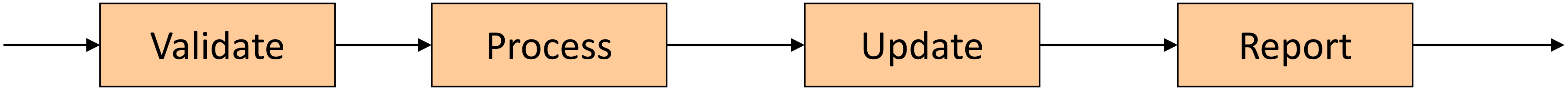


(a) pipes and filters



(b) batch sequential

Data Flow Example



Data Flow Style

- Characterized by viewing the system as a series of transformations on successive pieces of input data

- Data enters the system and then flows through the components one at a time until they are assigned to output or a data store

- Batch sequential style
  - The processing steps are independent components
  - Each step runs to completion before the next step begins
  - Example can be a batch job

- Pipe-and-filter style
  - Emphasizes the incremental transformation of data by successive components
  - The filters incrementally transform the data (entering and exiting via streams)
  - The filters use little contextual information and retain no state between instantiations
  - The pipes are stateless and simply exist to move data between filters
  - Example can be a compiler doing
    Lexical analysis -> Parsing -> Semantic analysis -> Code generation

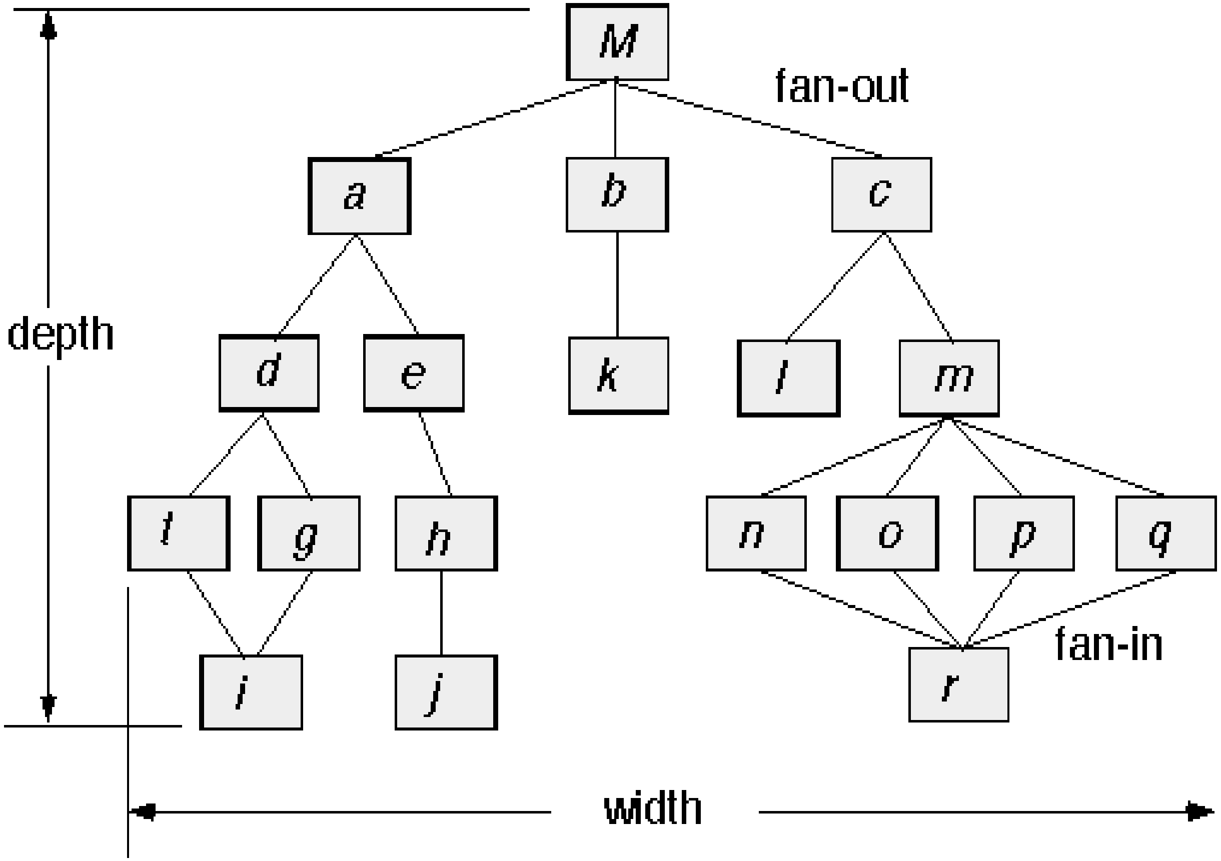
Data Flow Style (continued)

- Advantages
  - Has a simplistic design in the limited ways in which the components interact with the environment
  - Consists of no more and no less than the construction of its parts
  - Simplifies reuse and maintenance
  - Is easily made into a parallel or distributed execution in order to enhance system performance
- Disadvantages
  - Implicitly encourages a batch mentality so interactive applications are difficult to create in this style
  - Ordering of filters can be difficult to maintain so the filters cannot cooperatively interact to solve a problem
  - Exhibits poor performance
    - Filters typically force the least common denominator of data representation (usually ASCII stream)
    - Filter may need unlimited buffers if they cannot start producing output until they receive all of the input
    - Each filter operates as a separate process or procedure call, thus incurring overhead in set-up and take-down time

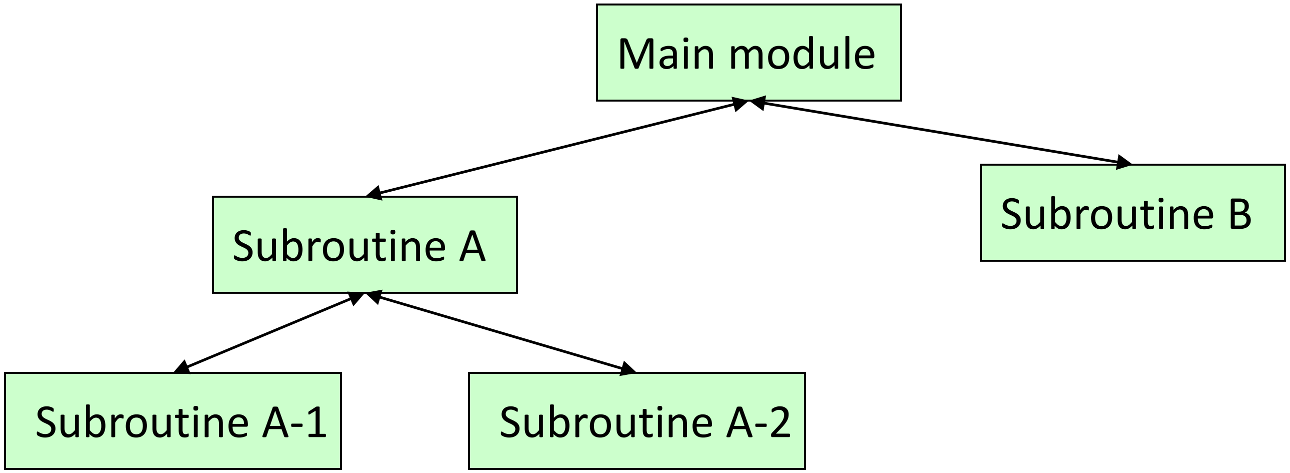
Data Flow Style (continued)

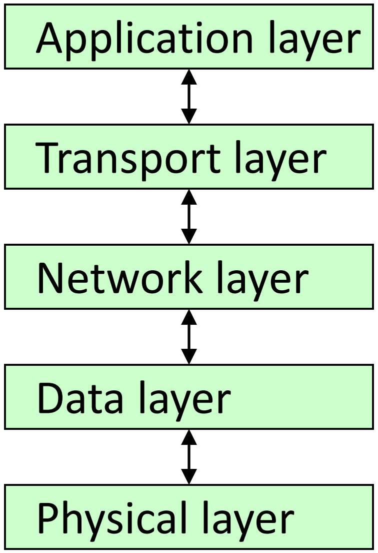
- Use this style when it makes sense to view your system as one that produces a well-defined easily identified output
  - The output should be a direct result of sequentially transforming a well-defined easily identified input in a time-independent fashion

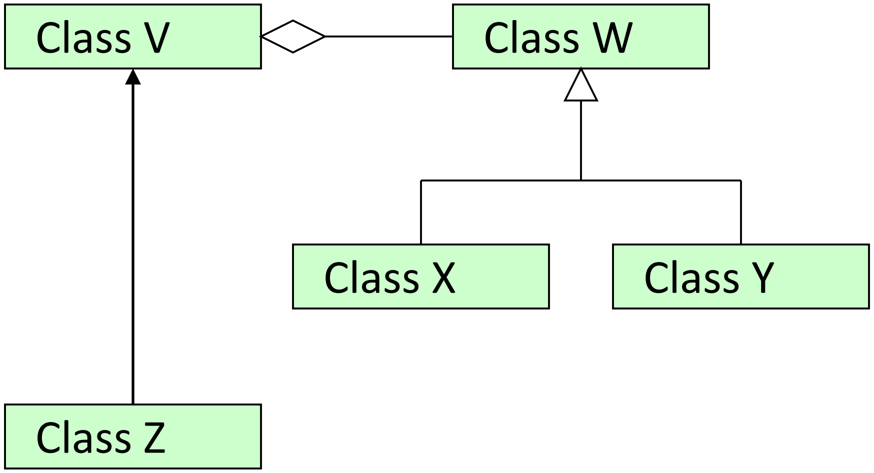
Call and Return Architecture



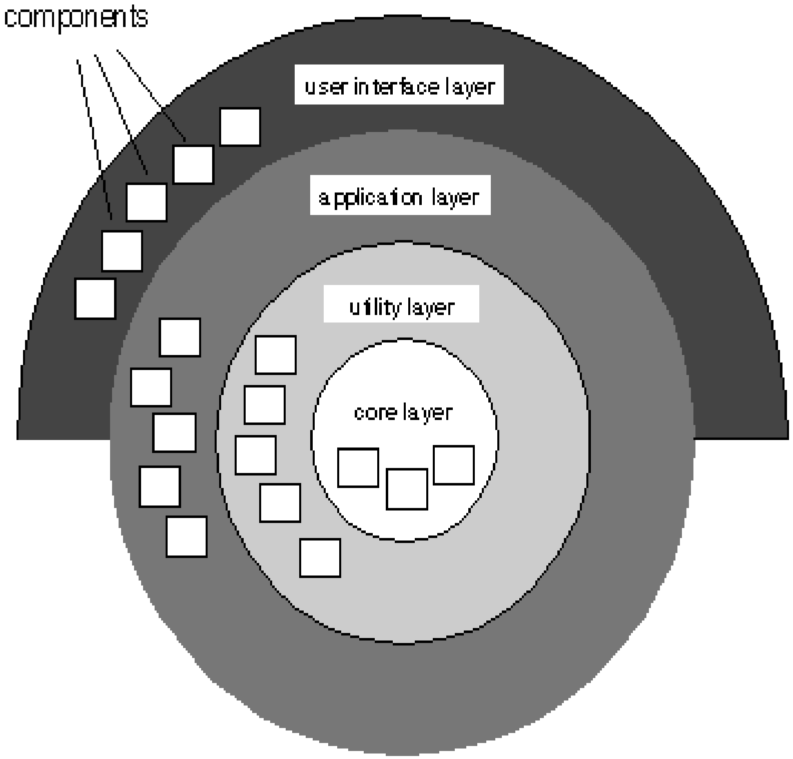
Call-and-Return Style







Layered Architecture



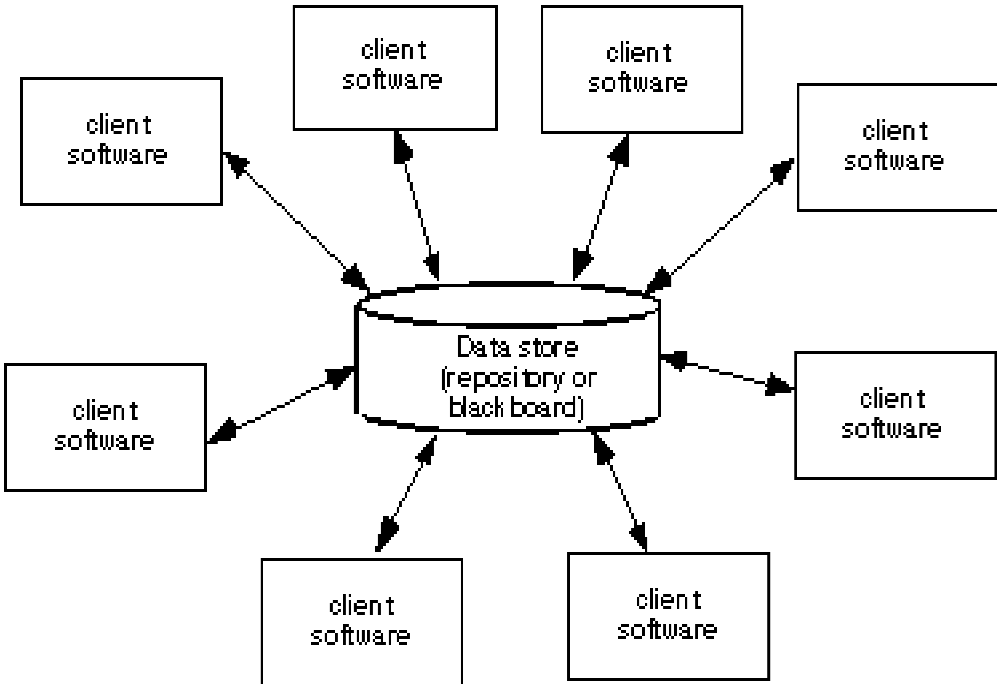
Call-and-Return Style

- Has the goal of modifiability and scalability
- Has been the dominant architecture since the start of software development
- Main program and subroutine style
  - Decomposes a program hierarchically into small pieces (i.e., modules)
  - Typically has a single thread of control that travels through various components in the hierarchy
- Remote procedure call style
  - Consists of main program and subroutine style of system that is decomposed into parts that are resident on computers connected via a network
  - Strives to increase performance by distributing the computations and taking advantage of multiple processors
  - Incurs a finite communication time between subroutine call and response

Call-and-Return Style (continued)

- Object-oriented or abstract data type system
  - Emphasizes the bundling of data and how to manipulate and access data
  - Keeps the internal data representation hidden and allows access to the object only through provided operations
  - Permits inheritance and polymorphism
- Layered system
  - Assigns components to layers in order to control inter-component interaction
  - Only allows a layer to communicate with its immediate neighbor
  - Assigns core functionality such as hardware interfacing or system kernel operations to the lowest layer
  - Builds each successive layer on its predecessor, hiding the lower layer and providing services for the upper layer
  - Is compromised by layer bridging that skips one or more layers to improve runtime performance
- Use this style when the order of computation is fixed, when interfaces are specific, and when components can make no useful progress while awaiting the results of request to other components

Data-Centered Architecture



Data-Centered Style (continued)

- Has the goal of integrating the data
- Refers to systems in which the access and update of a widely accessed data store occur
- A client runs on an independent thread of control
- The shared data may be a passive repository or an active blackboard
  - A blackboard notifies subscriber clients when changes occur in data of interest
- At its heart is a centralized data store that communicates with a number of clients
- Clients are relatively independent of each other so they can be added, removed, or changed in functionality
- The data store is independent of the clients

Data-Centered Style (continued)

- Use this style when a central issue is the storage, representation, management, and retrieval of a large amount of related persistent data
- Note that this style becomes client/server if the clients are modeled as independent processes

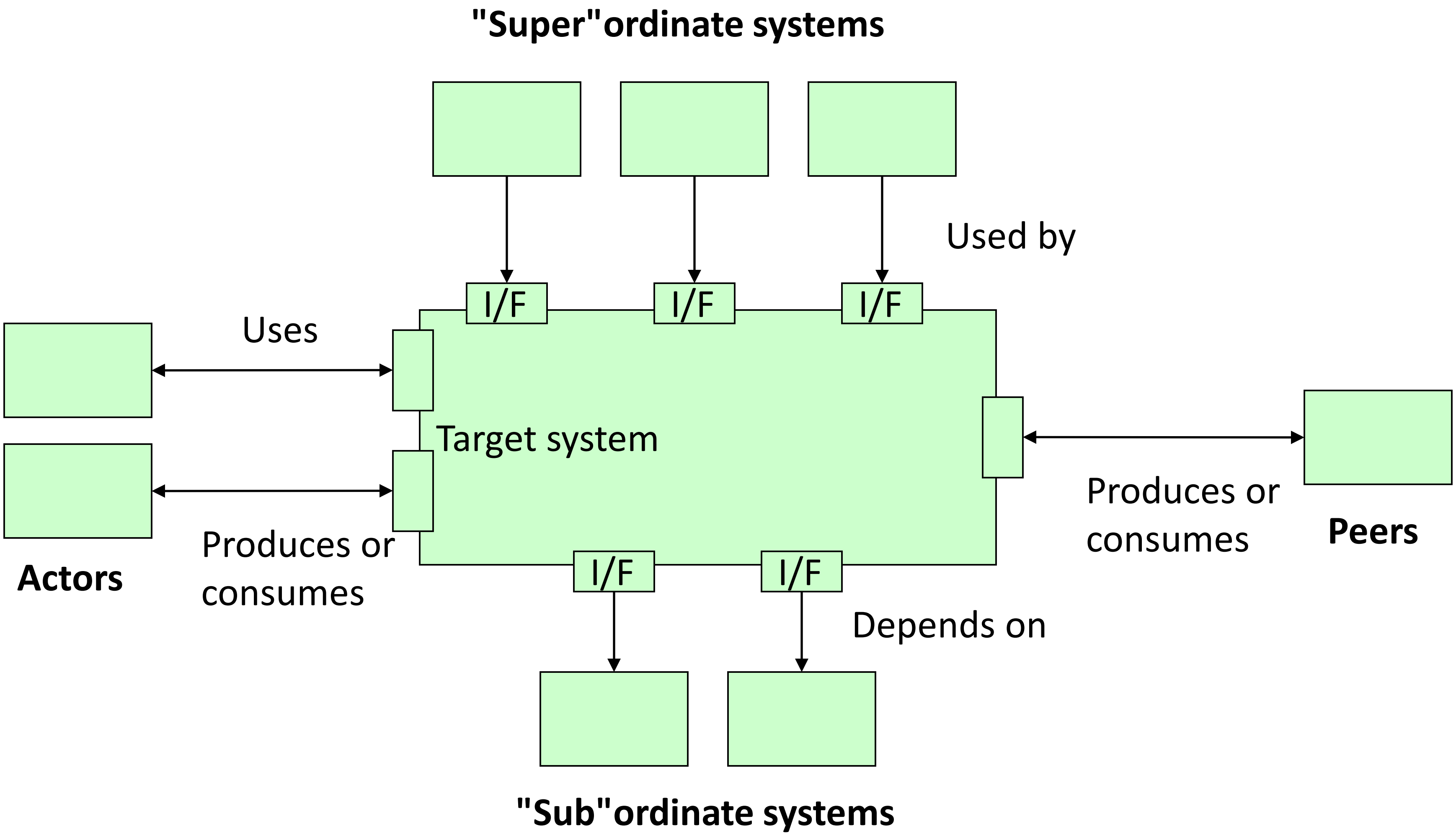
Architectural Design Process

Architectural Design Steps

1) Represent the system in context
2) Define archetypes (well-understood term or symbol…so that you don’t need to elaborate it hard)
3) Refine the architecture into components
4) Describe instantiations of the system

"A doctor can bury his mistakes, but an architect can only advise his client to plant vines." Frank Lloyd Wright

1. Represent the System in Context



1. Represent the System in Context (continued)

- Use an architectural context diagram (ACD) that shows
  - The identification and flow of all information into and out of a system
  - The specification of all interfaces
  - Any relevant support processing from/by other systems
- An ACD models the manner in which software interacts with entities external to its boundaries
- An ACD identifies systems that interoperate with the target system
  - Super-ordinate systems
    - Use target system as part of some higher level processing scheme
  - Sub-ordinate systems
    - Used by target system and provide necessary data or processing
  - Peer-level systems
    - Interact on a peer-to-peer basis with target system to produce or consume data
  - Actors
    - People or devices that interact with target system to produce or consume data

2. Define Archetypes

- Archetypes indicate the important abstractions within the problem domain (i.e., they model information)
- An archetype is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system
- It is also an abstraction from a class of programs with a common structure and includes class-specific design strategies and a collection of example program designs and implementations
- Only a relatively small set of archetypes is required in order to design even relatively complex systems
- The target system architecture is composed of these archetypes
  - They represent stable elements of the architecture
  - They may be instantiated in different ways based on the behavior of the system
  - They can be derived from the analysis class model
- The archetypes and their relationships can be illustrated in a UML class diagram

Example Archetypes in Software Architecture

- Node
- Detector/Sensor
- Indicator
- Controller
- Manager

3. Refine the Architecture into Components

- Based on the archetypes, the architectural designer refines the software architecture into components to illustrate the overall structure and architectural style of the system

- These components are derived from various sources
  - The application domain provides application components, which are the domain classes in the analysis model that represent entities in the real world
  - The infrastructure domain provides design components (i.e., design classes) that enable application components but have no business connection
    - Examples: memory management, communication, database, and task management
  - The interfaces in the ACD imply one or more specialized components that process the data that flow across the interface

- A UML class diagram can represent the classes of the refined architecture and their relationships

4. Describe Instantiations of the System

- An actual instantiation of the architecture is developed by applying it to a specific problem
- This demonstrates that the architectural structure, style and components are appropriate
- A UML component diagram can be used to represent this instantiation

Enterprise Architecture
(Coming from Big Picture)

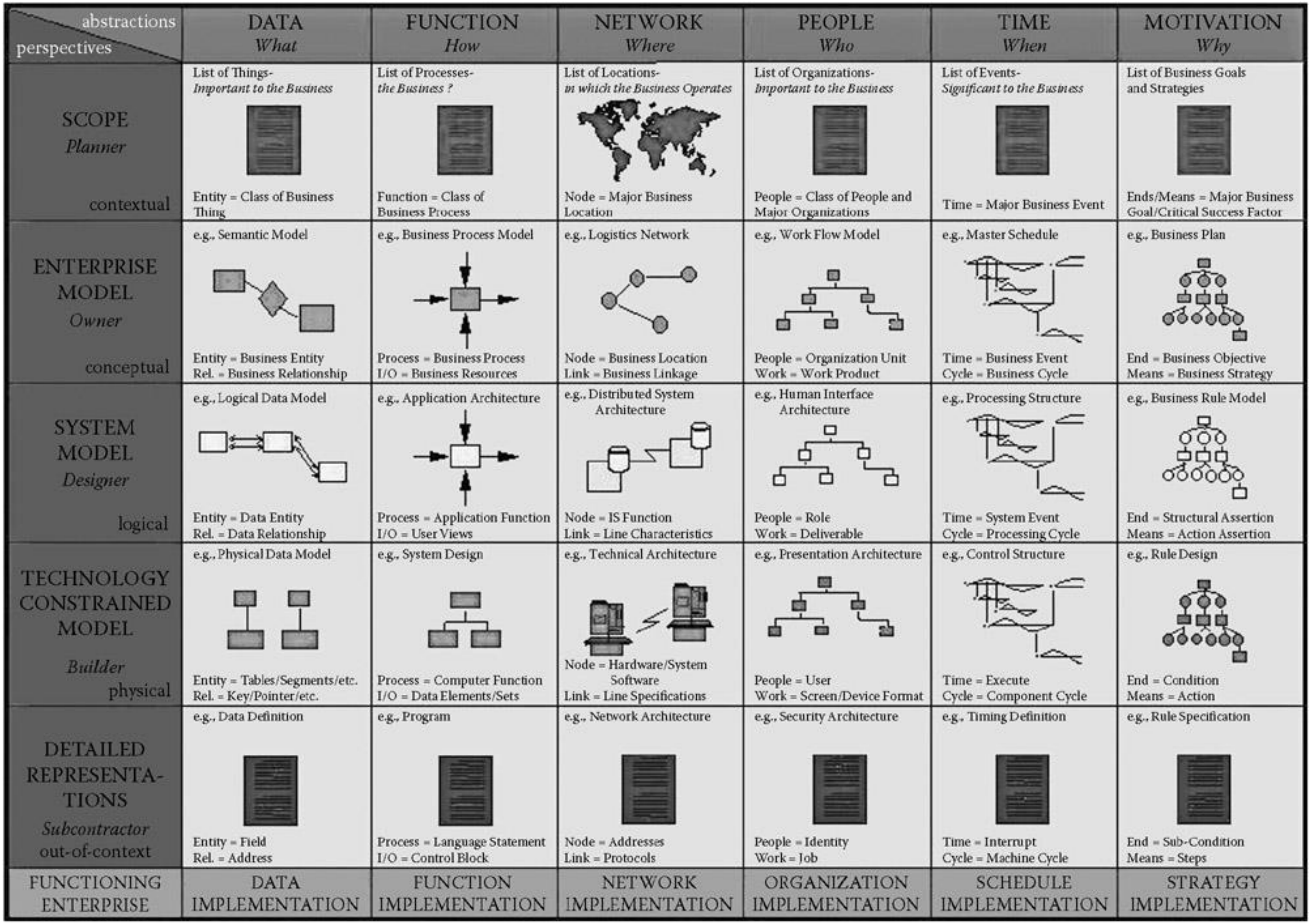
Enterprise architecture is the organizing logic for business processes and IT infrastructure reflecting the integration and standardization requirements of the company's operating model. The operating model is the desired state of business process integration and business process standardization for delivering goods and services to customers

-MIT Center for Information Systems Research

The Zachman Framework (evolving since 1987) establishes a common vocabulary and set of perspectives for defining and describing Enterprise Architecture.

The Zachman Framework describes a holistic model of an enterprise's information infrastructure from six perspectives: planner, owner, designer, builder, subcontractor, and the working system

Zachman Framework for EA



abstractions perspectives	DATA *What*	FUNCTION *How*	NETWORK *Where*	PEOPLE *Who*	TIME *When*	MOTIVATION *Why*
SCOPE *Planner* contextual	List of Things- *Important to the Business* Entity = Class of Business Thing	List of Processes- *the Business ?* Function = Class of Business Process	List of Locations- *in which the Business Operates* Node = Major Business Location	List of Organizations- *Important to the Business* People = Class of People and Major Organizations	List of Events- *Significant to the Business* Time = Major Business Event	List of Business Goals and Strategies Ends/Means = Major Business Goal/Critical Success Factor
ENTERPRISE MODEL *Owner* conceptual	e.g., Semantic Model Entity = Business Entity Rel. = Business Relationship	e.g., Business Process Model Process = Business Process I/O = Business Resources	e.g., Logistics Network Node = Business Location Link = Business Linkage	e.g., Work Flow Model People = Organization Unit Work = Work Product	e.g., Master Schedule Time = Business Event Cycle = Business Cycle	e.g., Business Plan End = Business Objective Means = Business Strategy
SYSTEM MODEL *Designer* logical	e.g., Logical Data Model Entity = Data Entity Rel. = Data Relationship	e.g., Application Architecture Process = Application Function I/O = User Views	e.g., Distributed System Architecture Node = IS Function Link = Line Characteristics	e.g., Human Interface Architecture People = Role Work = Deliverable	e.g., Processing Structure Time = System Event Cycle = Processing Cycle	e.g., Business Rule Model End = Structural Assertion Means = Action Assertion
TECHNOLOGY CONSTRAINED MODEL *Builder* physical	e.g., Physical Data Model Entity = Tables/Segments/etc. Rel. = Key/Pointer/etc.	e.g., System Design Process = Computer Function I/O = Data Elements/Sets	e.g., Technical Architecture Node = Hardware/System Software Link = Line Specifications	e.g., Presentation Architecture People = User Work = Screen/Device Format	e.g., Control Structure Time = Execute Cycle = Component Cycle	e.g., Rule Design End = Condition Means = Action
DETAILED REPRESENTATIONS *Subcontractor* out-of-context	e.g., Data Definition Entity = Field Rel. = Address	e.g., Program Process = Language Statement I/O = Control Block	e.g., Network Architecture Node = Addresses Link = Protocols	e.g., Security Architecture People = Identity Work = Job	e.g., Timing Definition Time = Interrupt Cycle = Machine Cycle	e.g., Rule Specification End = Sub-Condition Means = Steps
FUNCTIONING ENTERPRISE	DATA IMPLEMENTATION	FUNCTION IMPLEMENTATION	NETWORK IMPLEMENTATION	ORGANIZATION IMPLEMENTATION	SCHEDULE IMPLEMENTATION	STRATEGY IMPLEMENTATION

Enterprise Architecture A to Z: Frameworks, Business Process Modeling, SOA, and Infrastructure Technology by Daniel Minoli - Auerbach © 2008

Deriving Architecture

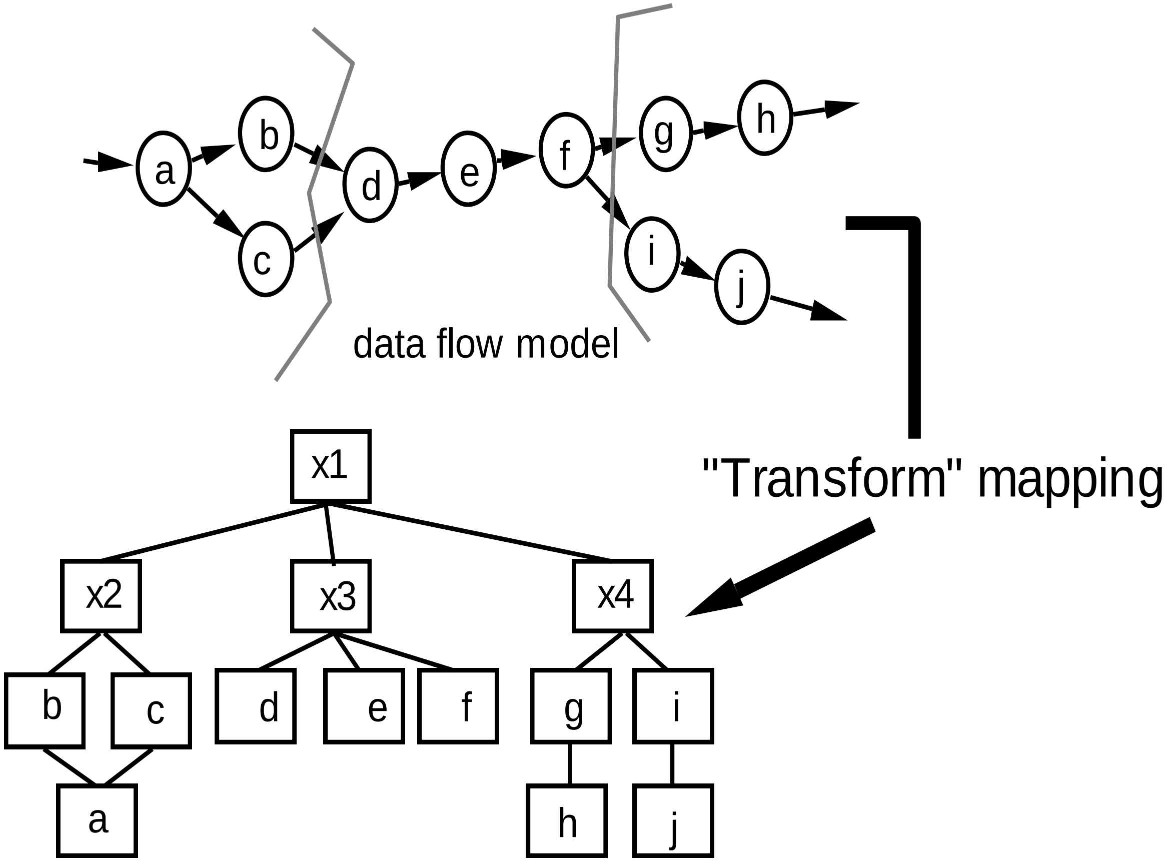
DFD to Architecture

- ❑ isolate incoming and outgoing flow boundaries
- ❑ 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

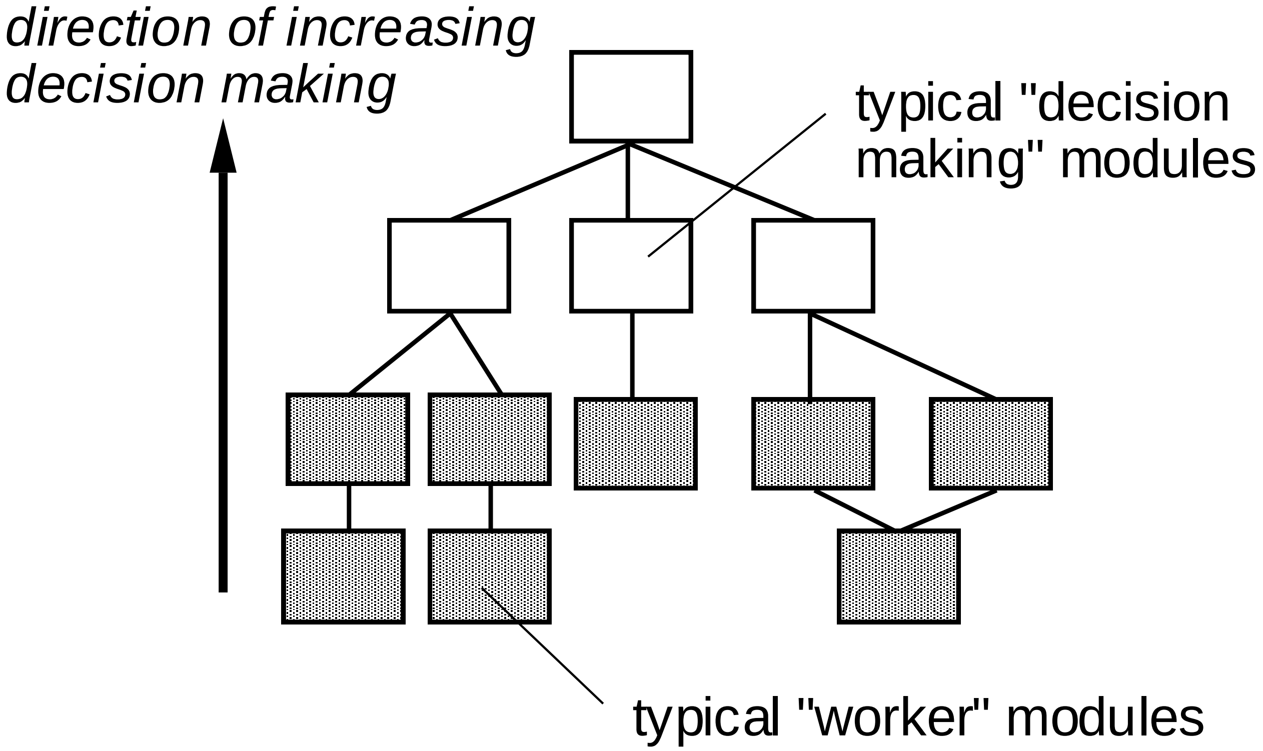
General Mapping Approach

- **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."**

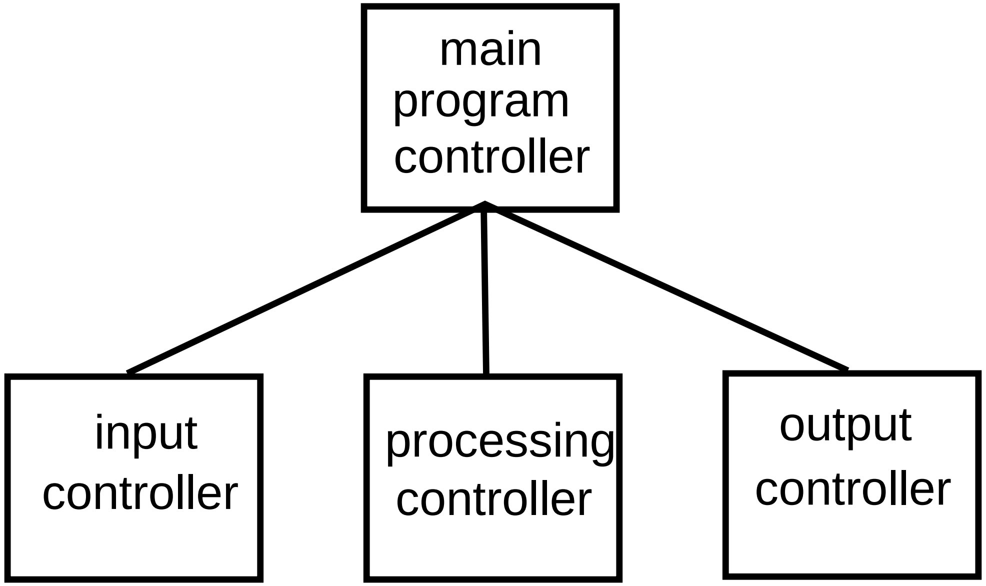
Transform Mapping



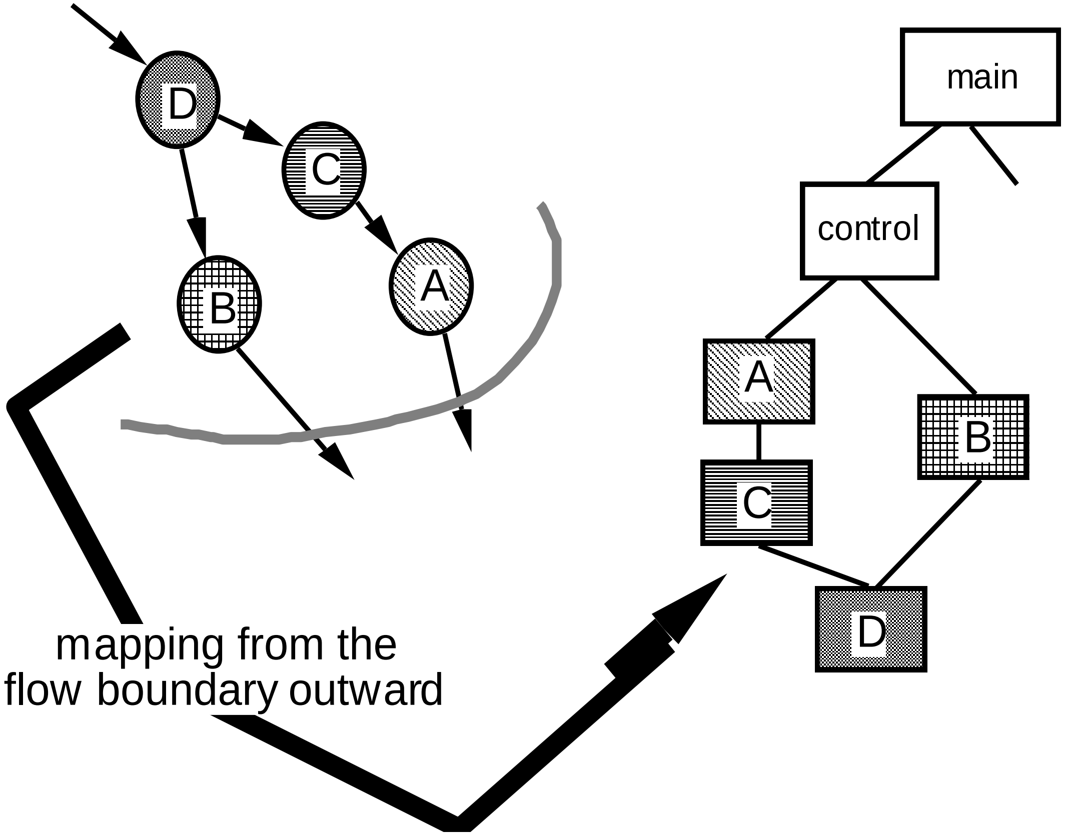
Factoring


direction of increasing decision making
typical "decision making" modules
typical "worker" modules

First Level Factoring



Second Level Mapping



mapping from the
flow boundary outward

Representing Architecture

Kruchten’s Questions

Boxes and Arrows of architecture diagram struggle to represent more on one blueprint than it can actually express

Do the boxes represent

running programs? Or

chunks of source code? Or

physical computers? Or

Merely logical groupings of functionality?

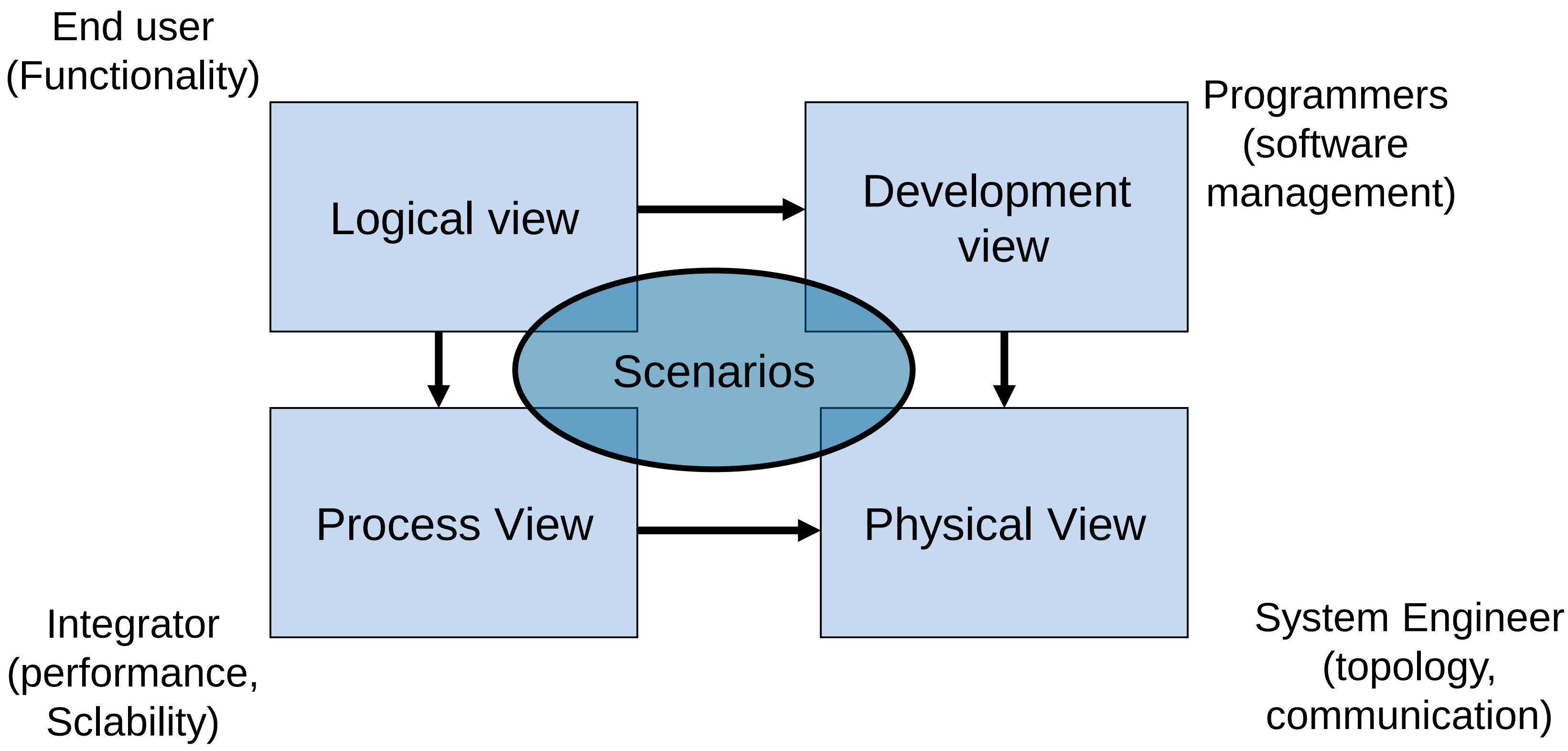
Do the arrows represent

compilation dependencies? Or
control flows? Or

data flows?

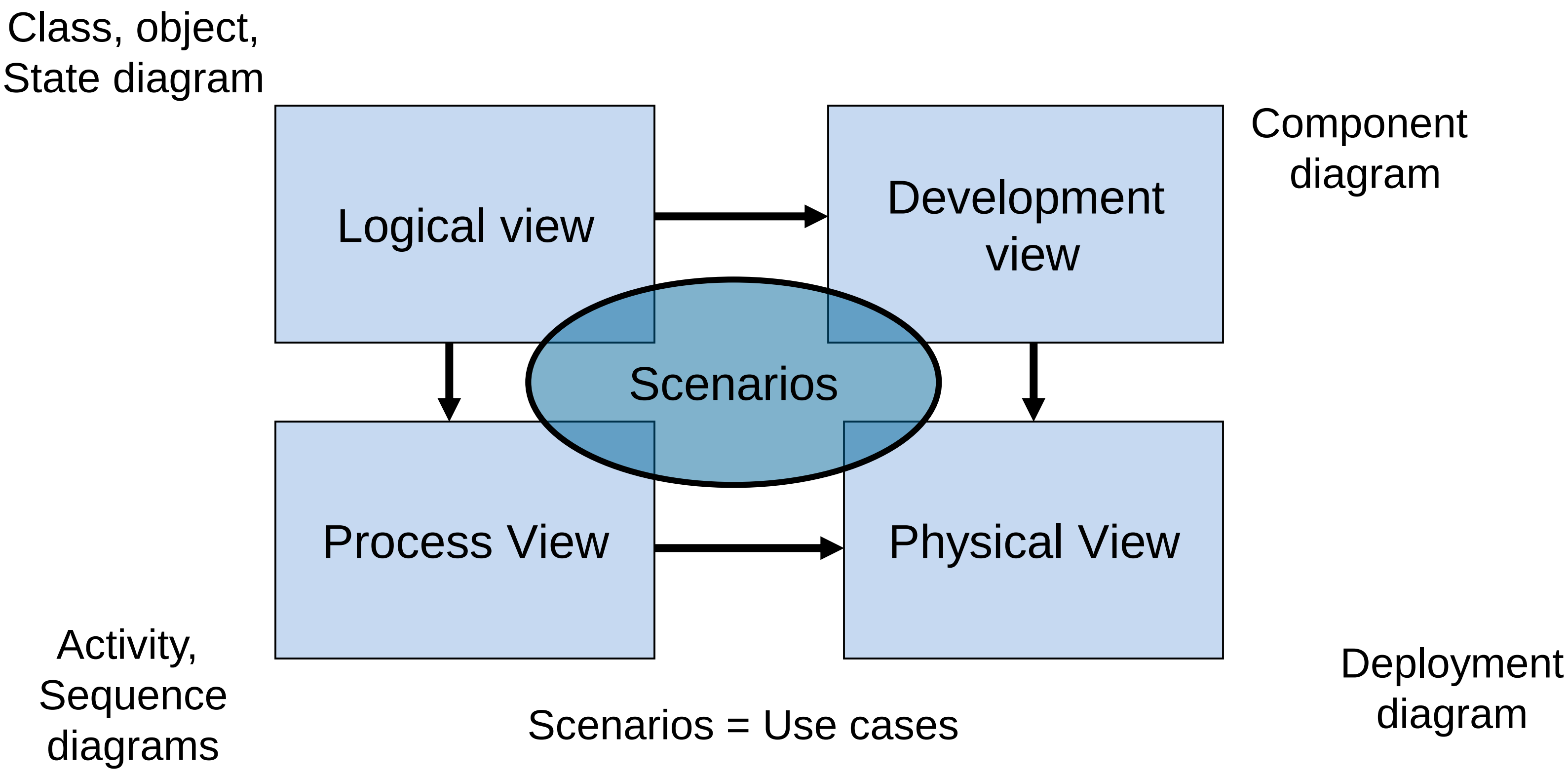
Kruchten, Philippe. Architectural Blueprints—The “4+1” View Model of Software Architecture, IEEE Software Nov 1995

“4+1” Views of Architecture



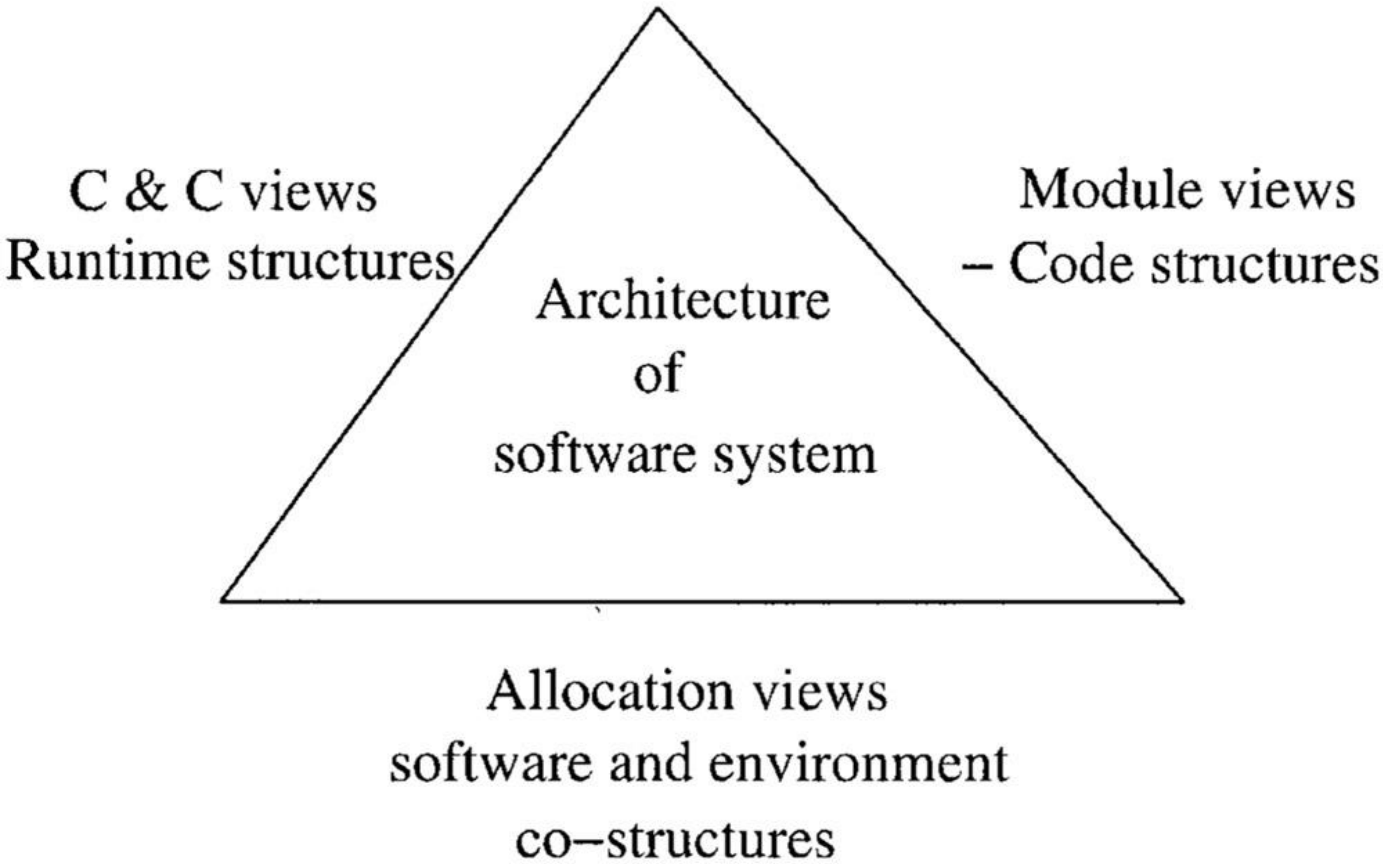
Kruchten, Philippe. Architectural Blueprints—The “4+1” View Model of Software Architecture, IEEE Software Nov 1995

“4+1” Views in UML



Scenarios = Use cases

Another Representation



An Integrated Approach to Software Engineering, Third Edition by Pankaj Jalote Springer © 2005

Assessing Alternative Architectural Designs

Various Assessment Approaches

A. Ask a set of questions that provide the designer with an early assessment of design quality and lay the foundation for more detailed analysis of the architecture

- Assess the control in an architectural design (see next slide)
- Assess the data in an architectural design (see upcoming slide)

B. Apply the architecture trade-off analysis method

C. Assess the architectural complexity

Approach A: Questions -- Assessing Control in an Architectural Design

- How is control managed within the architecture?
- Does a distinct control hierarchy exist, and if so, what is the role of components within this control hierarchy?
- How do components transfer control within the system?
- How is control shared among components?
- What is the control topology (i.e., the geometric form that the control takes)?
- Is control synchronized or do components operate asynchronously?

Approach A: Questions -- Assessing Data in an Architectural Design

- How are data communicated between components?
- Is the flow of data continuous, or are data objects passed to the system sporadically?
- What is the mode of data transfer (i.e., are data passed from one component to another or are data available globally to be shared among system components)
- Do data components exist (e.g., a repository or blackboard), and if so, what is their role?
- How do functional components interact with data components?
- Are data components passive or active (i.e., does the data component actively interact with other components in the system)?
- How do data and control interact within the system?

Approach B: Architecture Trade-off Analysis Method

1) Collect scenarios representing the system from the user's point of view
2) Elicit requirements, constraints, and environment description to be certain all stakeholder concerns have been addressed
3) Describe the candidate architectural styles that have been chosen to address the scenarios and requirements
4) Evaluate quality attributes by considering each attribute in isolation (reliability, performance, security, maintainability, flexibility, testability, portability, reusability, and interoperability)
5) Identify the sensitivity of quality attributes to various architectural attributes for a specific architectural style by making small changes in the architecture
6) Critique the application of the candidate architectural styles (from step #3) using the sensitivity analysis conducted in step #5

Based on the results of steps 5 and 6, some architecture alternatives may be eliminated. Others will be modified and represented in more detail until a target architecture is selected

Approach C: Assessing Architectural Complexity

- The overall complexity of a software architecture can be assessed by considering the dependencies between components within the architecture
- These dependencies are driven by the information and control flow within a system
- Three types of dependencies
  - Sharing dependency **U ← → □ ← → V**
    - Represents a dependency relationship among consumers who use the same source or producer
  - Flow dependency **→ U → V →**
    - Represents a dependency relationship between producers and consumers of resources
  - Constrained dependency **U "XOR" V**
    - Represents constraints on the relative flow of control among a set of activities such as mutual exclusion between two components

Summary

- A software architecture provides a uniform, high-level view of the system to be built
- It depicts
  - The structure and organization of the software components
  - The properties of the components
  - The relationships (i.e., connections) among the components
- Software components include program modules and the various data representations that are manipulated by the program
- The choice of a software architecture highlights early design decisions and provides a mechanism for considering the benefits of alternative architectures
- Data design translates the data objects defined in the analysis model into data structures that reside in the software

(More on next slide)

Summary (continued)

- A number of different architectural styles are available that encompass a set of component types, a set of connectors, semantic constraints, and a topological layout
- The architectural design process contains four distinct steps
  1) Represent the system in context
  2) Identify the component archetypes (the top-level abstractions)
  3) Identify and refine components within the context of various architectural styles
  4) Formulate a specific instantiation of the architecture
- Once a software architecture has been derived, it is elaborated and then analyzed against quality criteria