Chapter 10

Architectural Design

- Introduction
- Data design
- Software architectural styles
- Architectural design process
- Assessing alternative architectural designs

(Source: Pressman, R. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 2005)

Introduction

Definitions

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

Architectural Design Process

- Basic Steps
  - Creation of the data design
  - Derivation of one or more representations of the architectural structure of the system
  - Analysis of alternative architectural styles to choose the one best suited to customer requirements and quality attributes
  - Elaboration of the architecture based on the selected architectural style
- A database designer creates the data architecture for a system to represent the data components
- A system architect selects an appropriate architectural style derived during system engineering and software requirements analysis

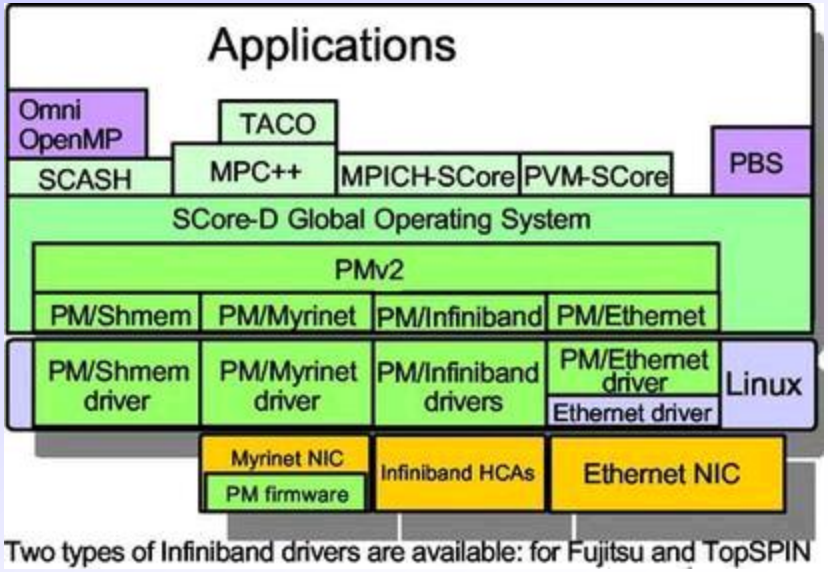
Emphasis on Software Components

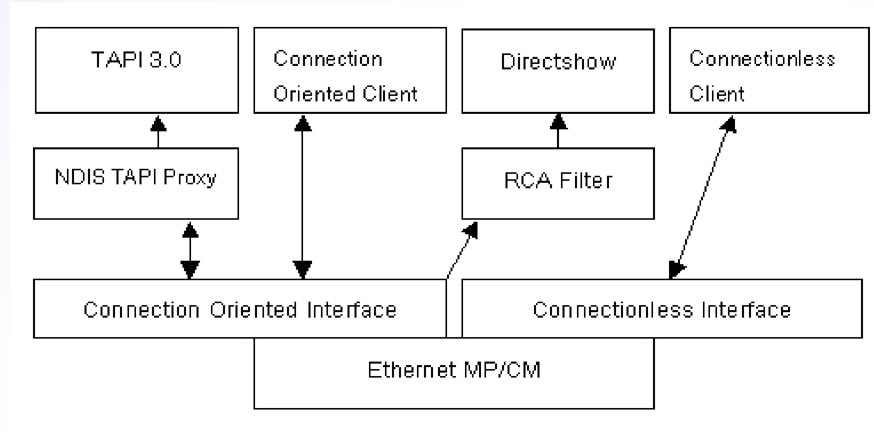
- A software architecture enables a software engineer to
  - Analyze the effectiveness of the design in meeting its stated requirements
  - Consider architectural alternatives at a stage when making design changes is still relatively easy
  - Reduce the risks associated with the construction of the software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

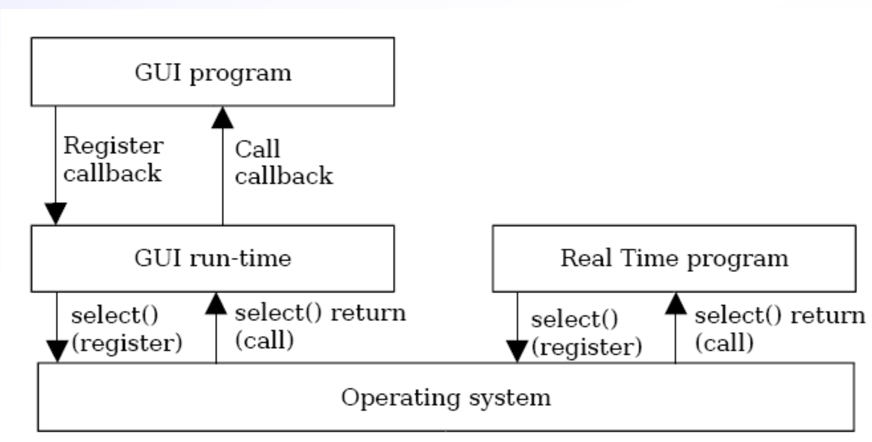
Importance of Software Architecture

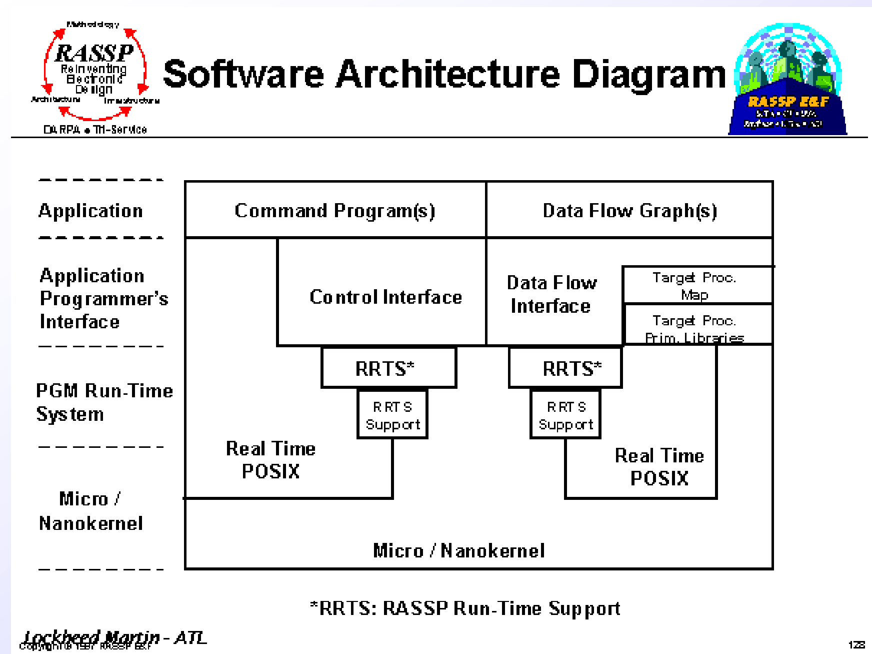
- Representations of software architecture are an enabler for communication between all stakeholders interested in the development of a computer-based system
- The software architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, 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

Example Software Architecture Diagrams









Data Design

Purpose of Data Design

- Data design translates data objects defined as part of the analysis model into
  - Data structures at the software component level
  - A possible database architecture at the application level
- It focuses on the representation of data structures that are directly accessed by one or more software components
- The challenge is to store and retrieve the data in such way that useful information can be extracted from the data environment
- "Data quality is the difference between a data warehouse and a data garbage dump"

Data Design Principles

- The systematic analysis principles that are applied to function and behavior should also be applied to data
- All data structures and the operations to be performed on each one should be identified
- 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
- Low-level data design decisions should be deferred until late in the design process
- The representation of a data structure should be known only to those modules that must make direct use of the data contained within the structure
- A library of useful data structures and the operations that may be applied to them should be developed
- A software programming language should support the specification and realization of abstract data types

Software Architectural Styles

Common Architectural Styles of American Homes

Common Architectural Styles of American Homes

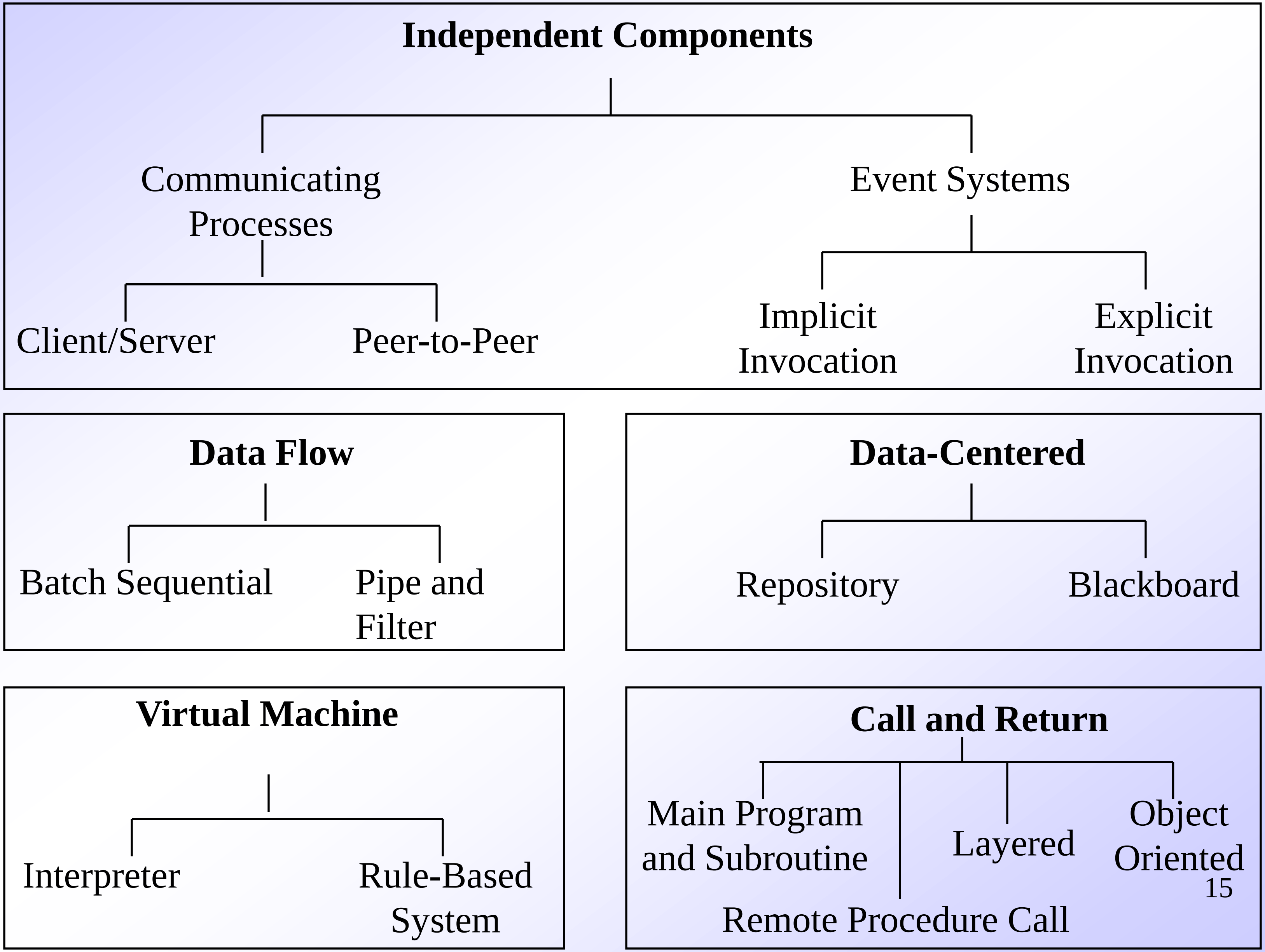
- A-Frame
- Bungalow
- Cape Cod
- Colonial
- Federal
- Four square
- Georgian
- Greek Revival
- Prairie Style
- Pueblo
- Ranch
- Split level
- Tidewater
- Tudor
- Victorian

Software Architectural Style

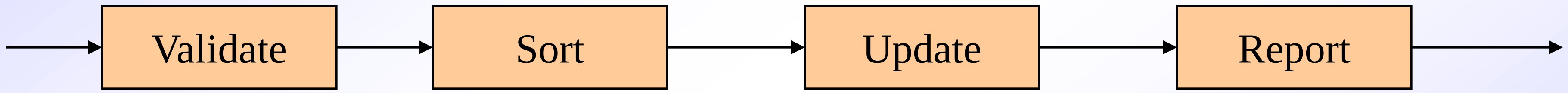
- The software that is built for computer-based systems exhibit one of many architectural styles
- Each style describes a system category that encompasses
  - A set of component types that perform a function required by the system
  - A set of connectors (subroutine call, remote procedure call, data 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)

A Taxonomy of Architectural Styles



Data Flow Style



Data Flow Style

- Has the goal of modifiability
- 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
- 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

(More on next slide)

Data Flow Style (continued)

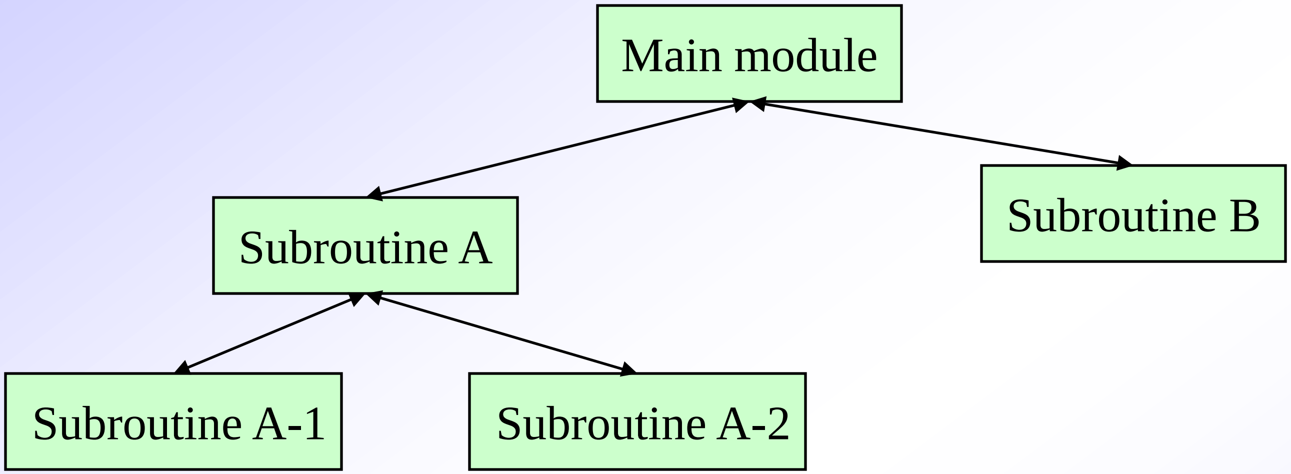
- Advantages
  - Has a simplistic design in the limited ways in which the components interact with the environment
  - Consists of no more and no less than the construction of its parts
  - Simplifies reuse and maintenance
  - Is easily made into a parallel or distributed execution in order to enhance 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

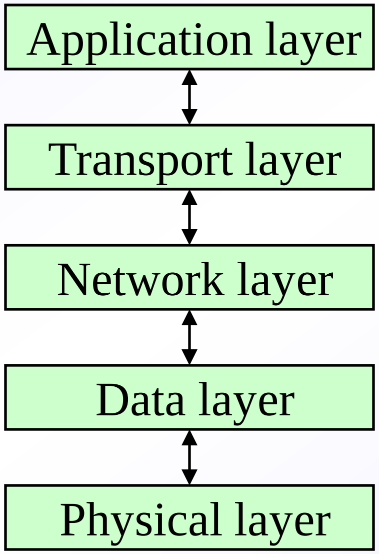
(More on next slide)

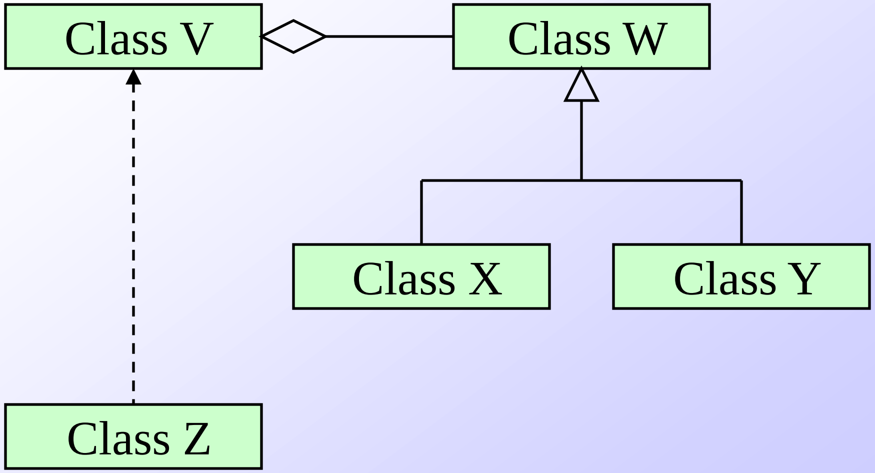
Data Flow Style (continued)

- 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 Style







Call-and-Return Style

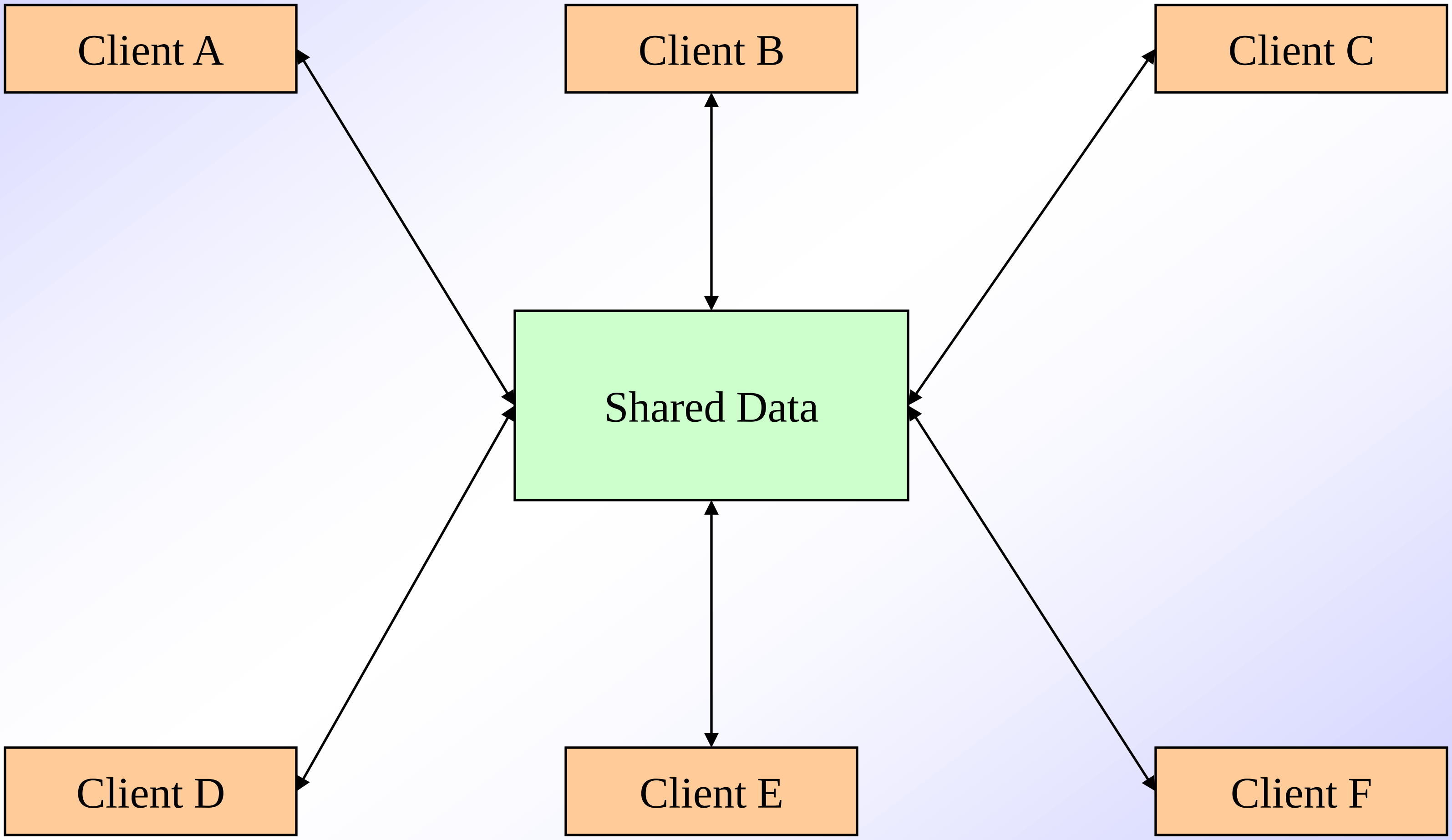
- Has the goal of modifiability and scalability
- Has been the dominant architecture since the start of software development
- Main program and subroutine style
  - Decomposes a program hierarchically into small pieces (i.e., modules)
  - Typically has a single thread of control that travels through 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

(More on next slide)

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 Style



Data-Centered Style (continued)

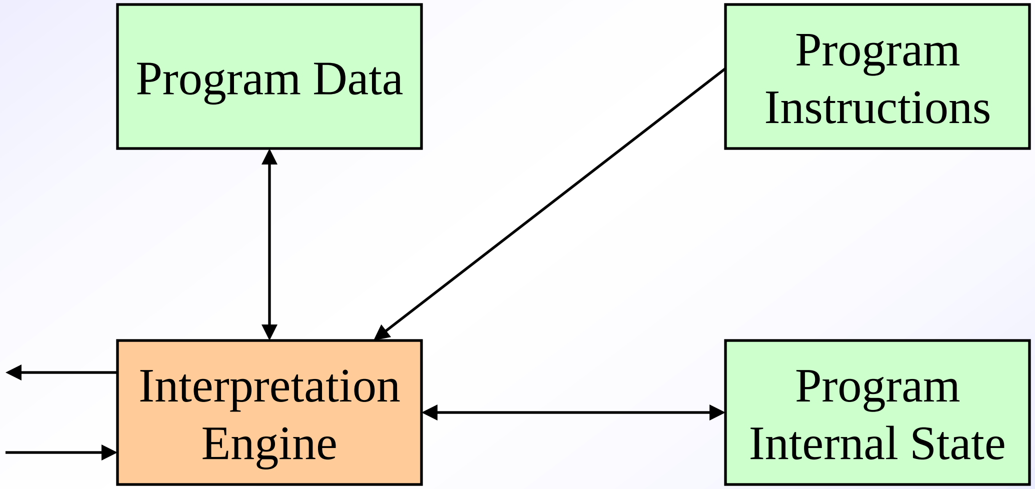
- Has the goal of integrating the data
- Refers to systems in which the access and update of a widely accessed data store occur
- A client runs on an independent thread of control
- The shared data may be a passive repository or an active blackboard
  - A blackboard notifies subscriber clients when changes 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

(More on next slide)

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

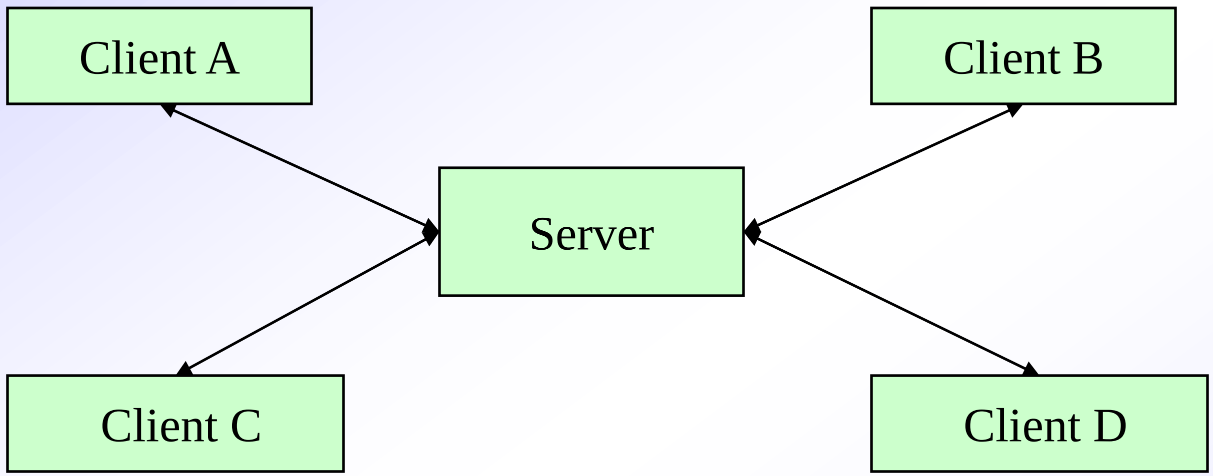
Virtual Machine Style

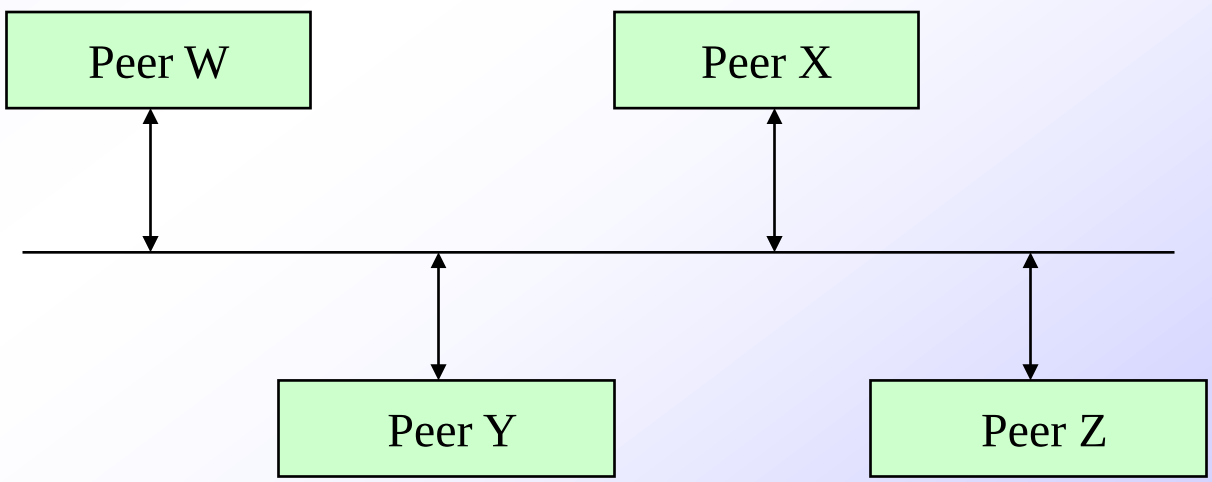


Virtual Machine Style

- Has the <u>goal</u> of portability
- Software systems in this style <u>simulate</u> some functionality that is not native to the hardware and/or software on which it is implemented
  - Can simulate and test hardware platforms that have not yet been built
  - Can simulate "disaster modes" as in flight simulators or safety-critical systems that would be too complex, costly, or dangerous to test with the real system
- Examples include interpreters, rule-based systems, and command language processors
- <u>Interpreters</u>
  - Add <u>flexibility</u> through the ability to interrupt and query the program and introduce modifications at runtime
  - Incur a <u>performance cost</u> because of the additional computation involved in execution
- Use this style when you have developed a program or some form of computation but have <u>no make of machine</u> to directly run it on

Independent Component Style





Independent Component Style

- Consists of a number of <u>independent</u> processes that communicate through messages
- Has the <u>goal</u> of modifiability by decoupling various portions of the computation
- Sends data between processes but the processes <u>do not</u> directly control each other
- <u>Event systems</u> style
  - Individual components <u>announce</u> data that they wish to share (<u>publish</u>) with their environment
  - The other components may <u>register</u> an interest in this class of data (subscribe)
  - Makes use of a message component that <u>manages</u> communication among the other components
  - Components <u>publish</u> information by <u>sending</u> it to the message manager
  - When the data appears, the subscriber is invoked and receives the data
  - <u>Decouples</u> component implementation from knowing the names and locations of other components

(More on next slide)

Independent Component Style (continued)

- Communicating processes style
  - These are classic multi-processing systems
  - Well-know subtypes are client/server and peer-to-peer
  - The goal is to achieve scalability
  - A server exists to provide data and/or services to one or more clients
  - The client originates a call to the server which services the request
- Use this style when
  - Your system has a graphical user interface
  - Your system runs on a multiprocessor platform
  - Your system can be structured as a set of loosely coupled components
  - Performance tuning by reallocating work among processes is important
  - Message passing is sufficient as an interaction mechanism among components

Heterogeneous Styles

- Systems are seldom built from a single architectural style
- Three kinds of heterogeneity
  - Locationally heterogeneous
    - The drawing of the architecture reveals different styles in different areas (e.g., a branch of a call-and-return system may have a shared repository)
  - Hierarchically heterogeneous
    - A component of one style, when decomposed, is structured according to the rules of a different style
  - Simultaneously heterogeneous
    - Two or more architectural styles may both be appropriate descriptions for the style used by a computer-based system

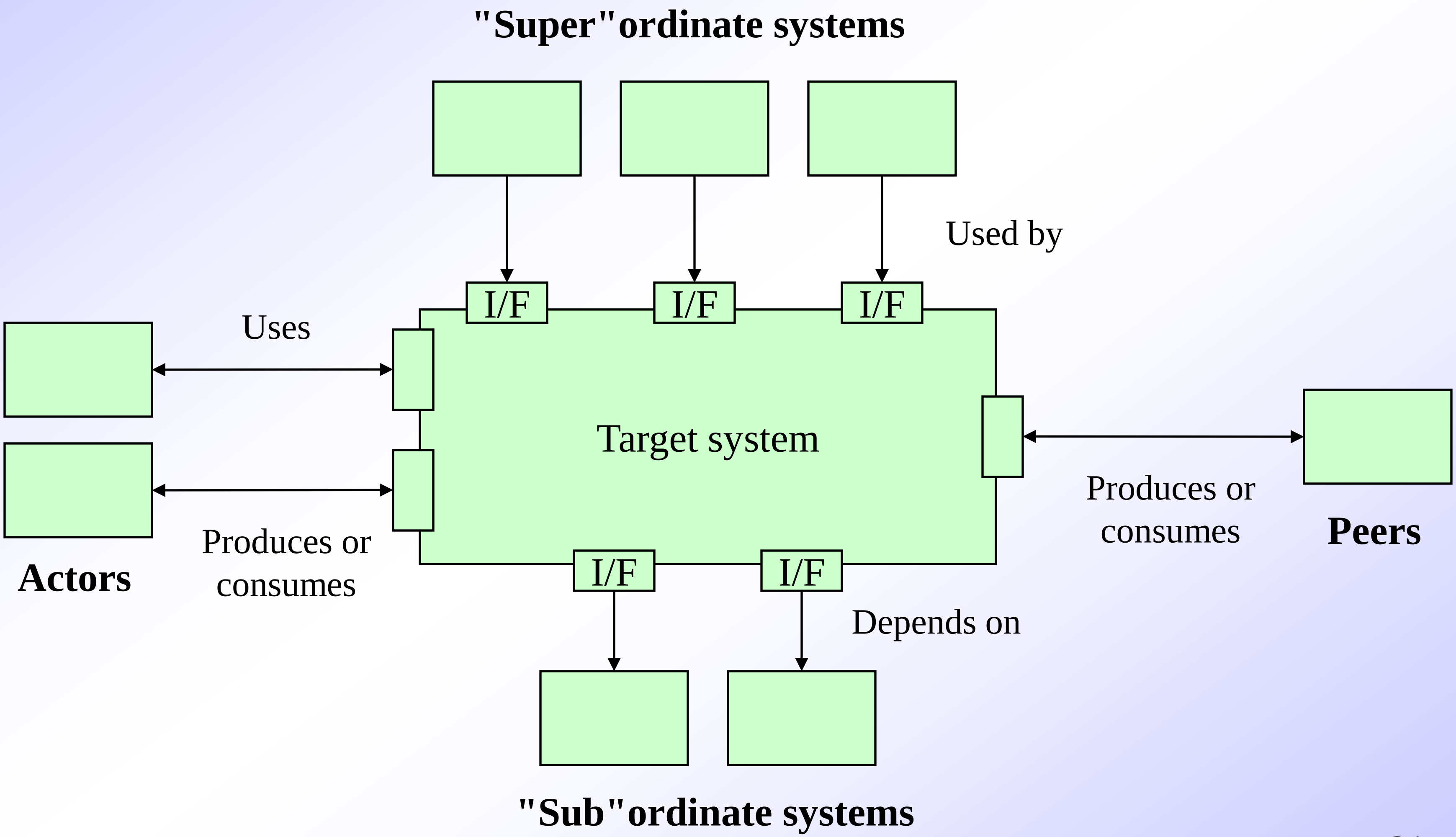
Architectural Design Process

Architectural Design Steps

1) Represent the system in context
2) Define archetypes
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



(More on next slide)

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 Humanity

- Addict/Gambler
- Amateur
- Beggar
- Clown
- Companion
- Damsel in distress
- Destroyer
- Detective
- Don Juan
- Drunk
- Engineer
- Father
- Gossip
- Guide
- Healer
- Hero
- Judge
- King
- Knight
- Liberator/Rescuer
- Lover/Devotee
- Martyr
- Mediator
- Mentor/Teacher
- Messiah/Savior
- Monk/Nun
- Mother
- Mystic/Hermit
- Networker
- Pioneer
- Poet
- Priest/Minister
- Prince
- Prostitute
- Queen
- Rebel/Pirate
- Saboteur
- Samaritan
- Scribe/Journalist
- Seeker/Wanderer
- Servant/Slave
- Storyteller
- Student
- Trickster/Thief
- Vampire
- Victim
- Virgin
- Visionary/Prophet
- Warrior/Soldier

(Source: http://www.myss.com/ThreeArchs.asp)

Example Archetypes in Software Architecture

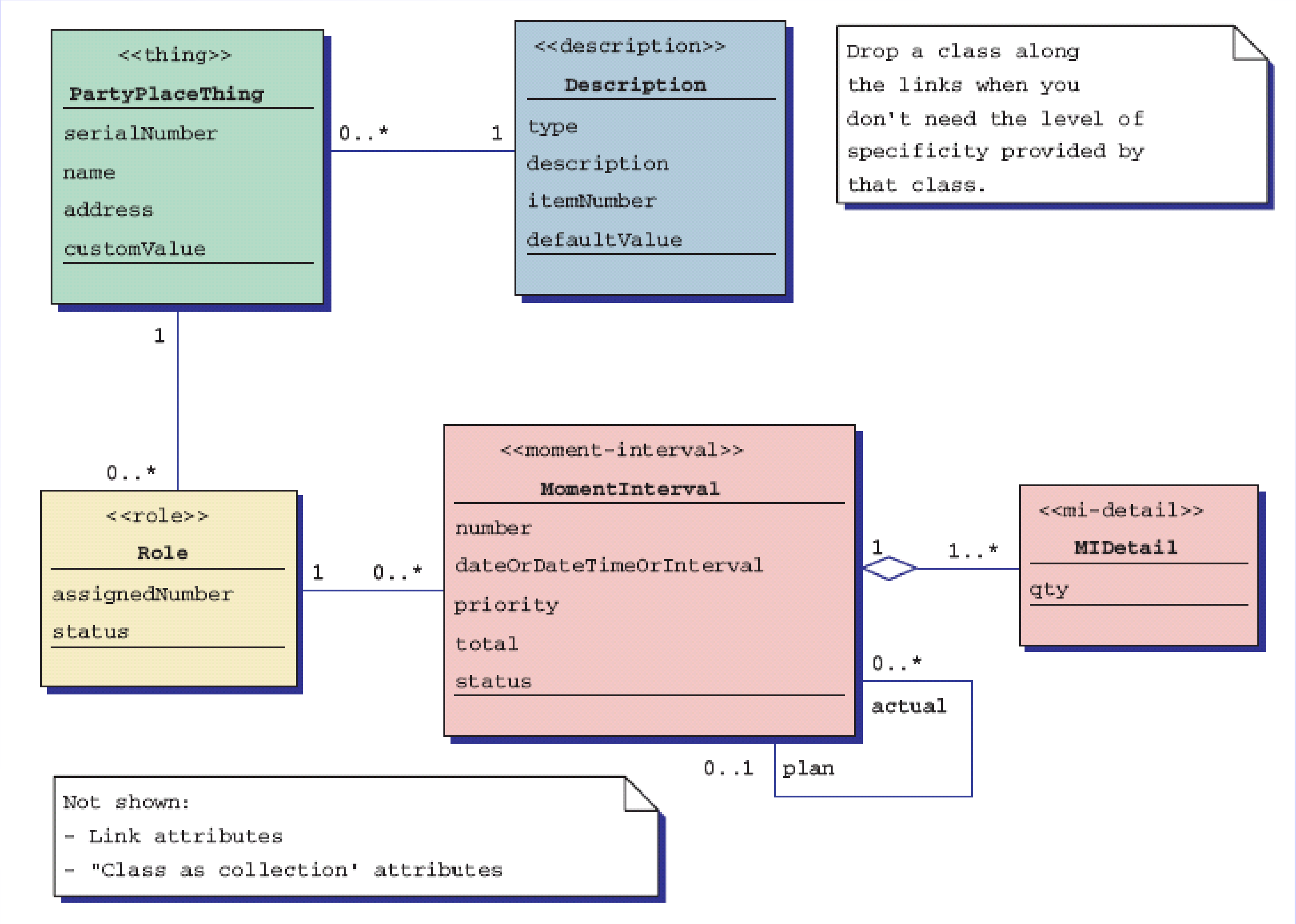
- Node
- Detector/Sensor
- Indicator
- Controller
- Manager

(Source: Pressman)

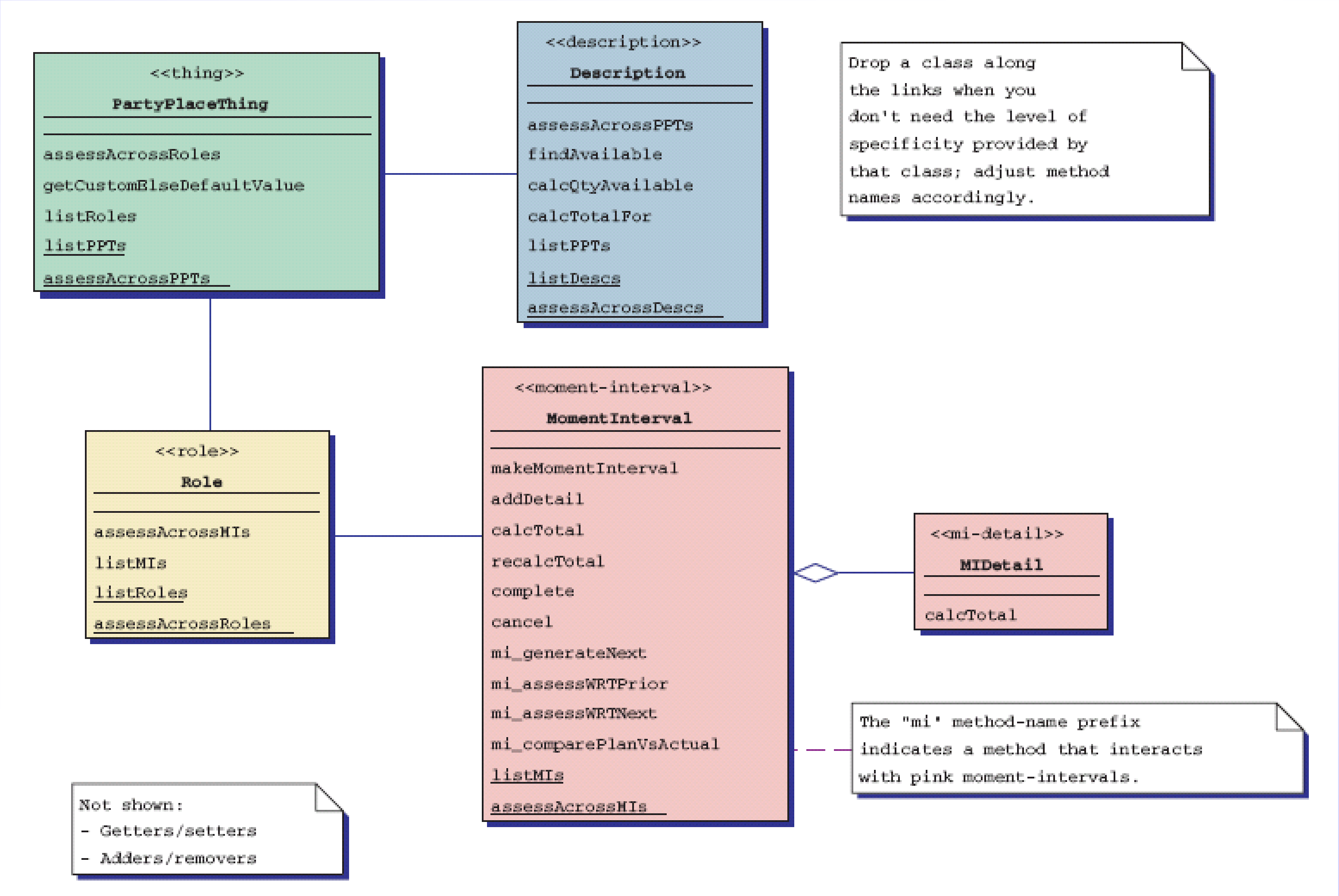
- Moment-Interval
- Role
- Description
- Party, Place, or Thing

(Source: Archetypes, Color, and the Domain Neutral Component)

Archetypes – their attributes



Archetypes – their methods



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

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 <u>components</u>
  - The <u>properties</u> of the components
  - The <u>relationships</u> (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 <u>early</u> design decisions and provides a mechanism for considering the <u>benefits</u> of alternative architectures
- Data design <u>translates</u> 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

☺