Software Architecture and Design Documentation

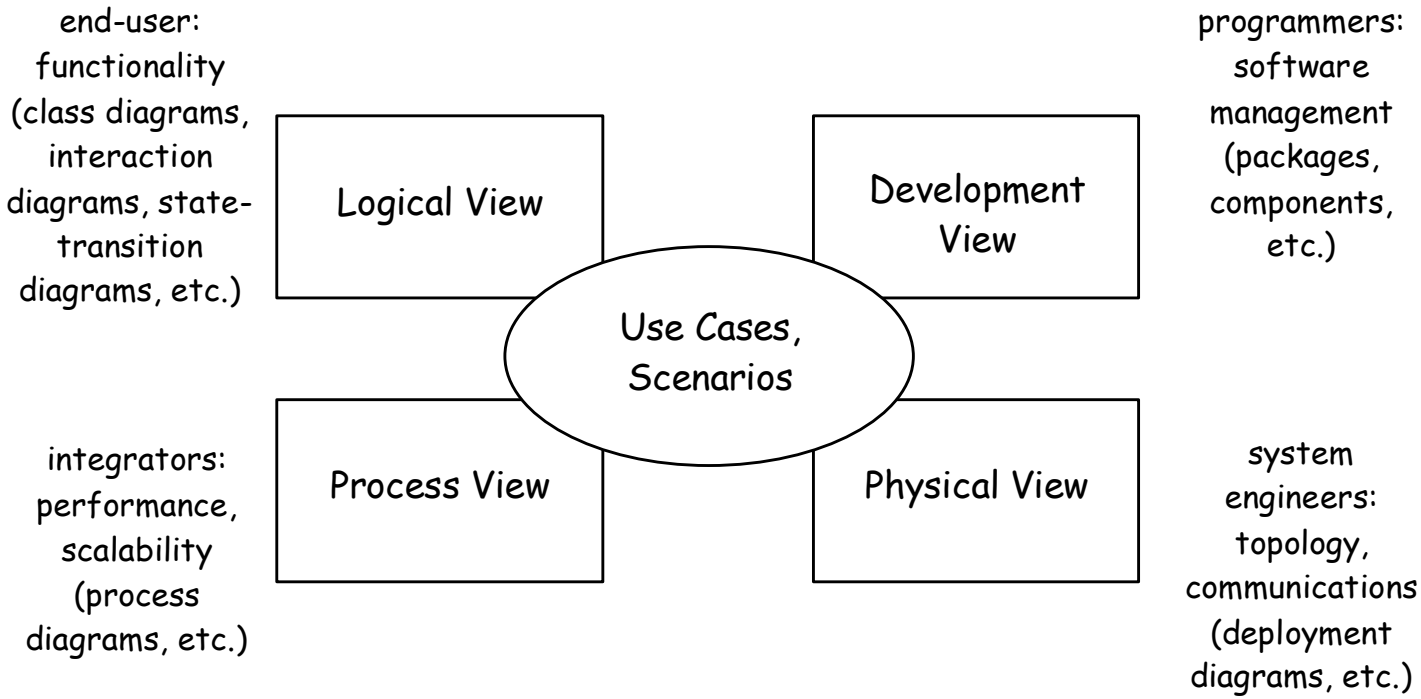
- The Software Design Document (SDD) captures the design of a system
- At minimum, create a document describing the **software architecture** of a system (or be able to produce the document from a tool)
- You may package it into a separate document called Software Architecture Document
- A reasonable size for a Software Architecture Document for even a large system is 50-100 pages
- Alternatively, you may produce a series of smaller working documents as you progress with the development

The Need For Multiple Architectural Views

- Architecture and design is about system structure
  - How the system is decomposed into parts
  - Components and interactions with appropriate properties, enabling appropriate analyses
- But this begs the question: what kinds of structure?
- Many possibilities:
  - Code structure
  - Run-time structure
  - Process structure
  - Work breakdown structure
- Each of these can be the basis of a Design View (or Architectural View)

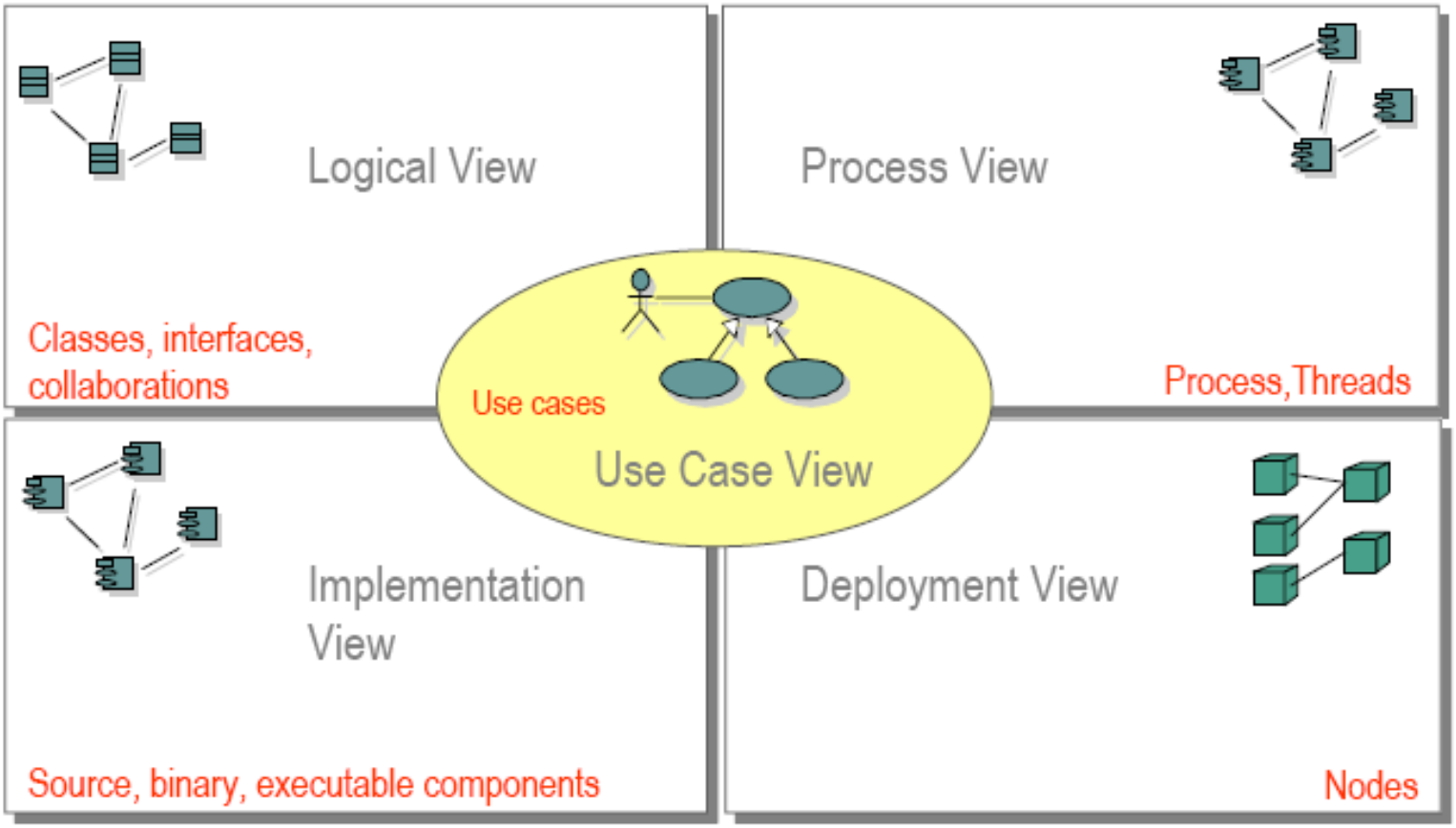
Kruchten's "4+1 View Model" of the Architecture

- P. Kruchten. The 4+1 View Model of Architecture. In *IEEE Software*, vol. 12, no. 6, November 1995, pp. 42-50, © IEEE 1995



Note: The concrete selection of views for a given project may vary, but the main contribution of having to support multiple views and tie them back to the requirements remains.

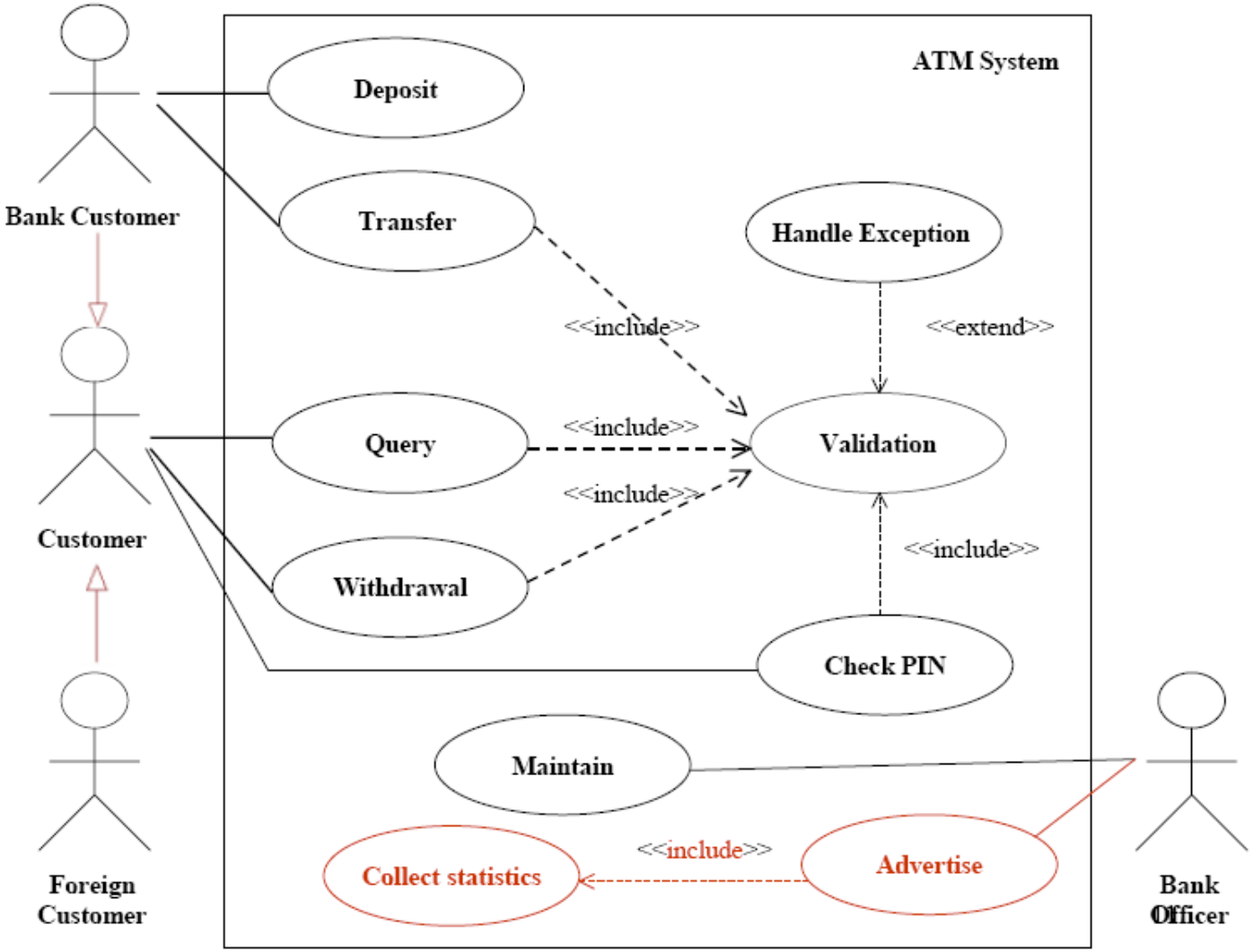
The “4+1” Views of Architecture



Use-case View

- Contains only architecturally significant use cases (whereas the final use case model contains all the use cases).
  - The *logical view* is derived using the use cases identified in the architectural view of the use case model.
- Architecturally significant use cases:
  - critical use cases, those that are most important to the users of the system (from a functionality perspective)
  - use cases that carry the major risks
  - use cases that have the most important quality requirements, such as performance, security, usability, etc.

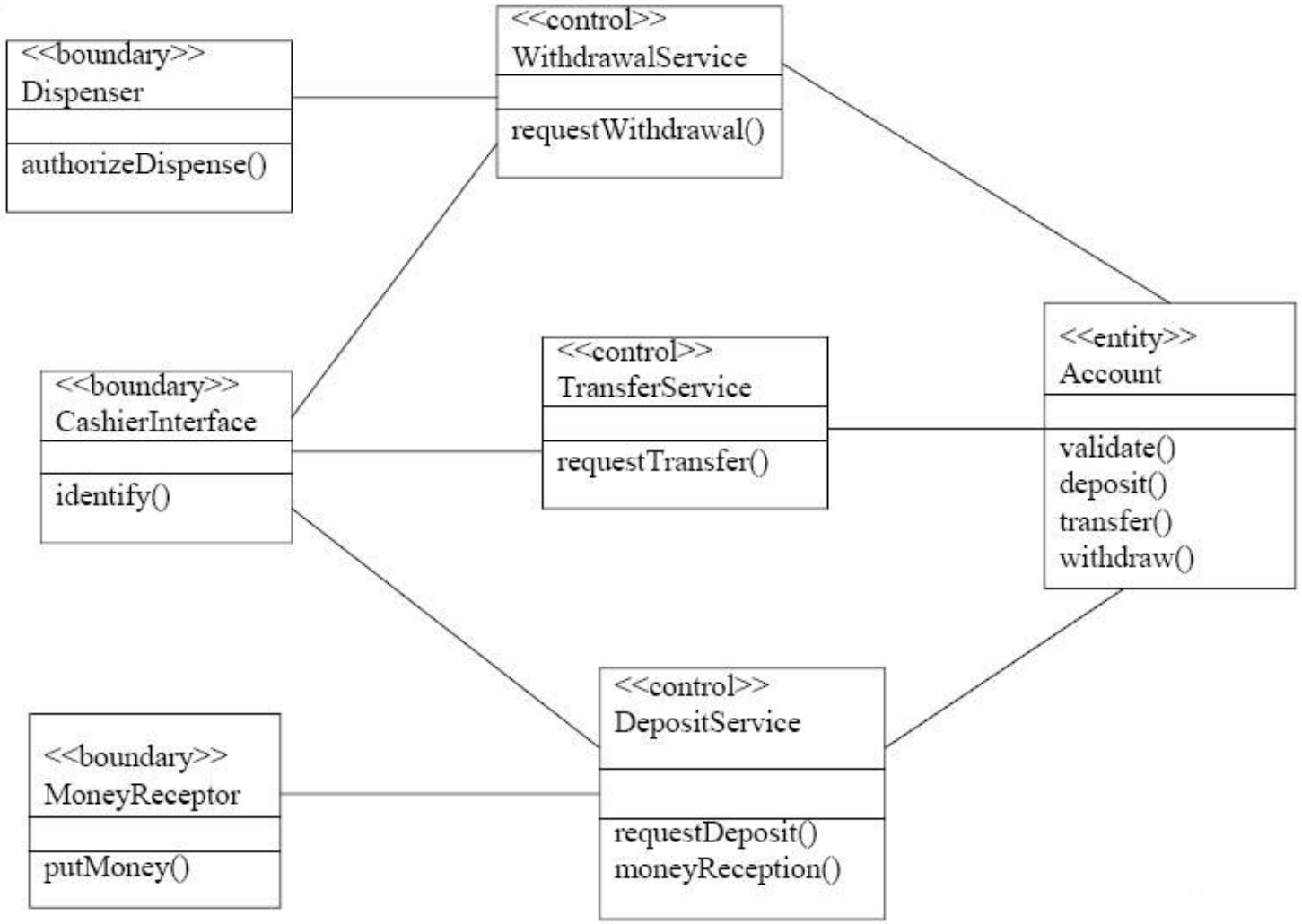
Use-case View : Example



Logical View

- The Logical View is a subset of the Design Model which presents architecturally significant design elements
- describes the most important classes
- their organization in packages and subsystems
- organization of these packages and subsystems into layers
- It also describes the most important use-case realizations, for example, the dynamic aspects of the architecture

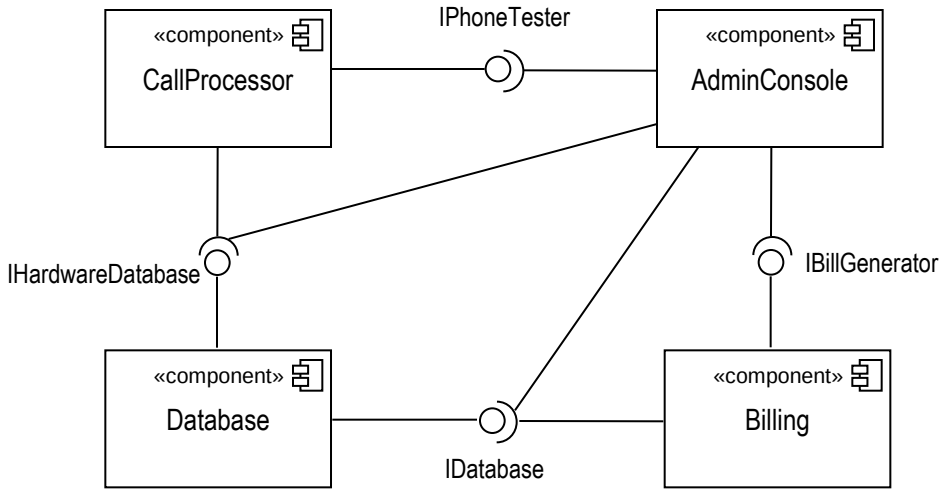
Logical View : Class Diagram



Logical View : Component Diagram

- Use a component diagram to show provided/required interfaces and decomposition of components
- Describe each component's responsibilities
- Other important information, e.g.,
  - Dependencies
  - Non-functional properties (e.g., maximum memory usage, reliability, required test coverage, etc.)
  - Mandated implementation technology (e.g., EJB, COM, etc.)

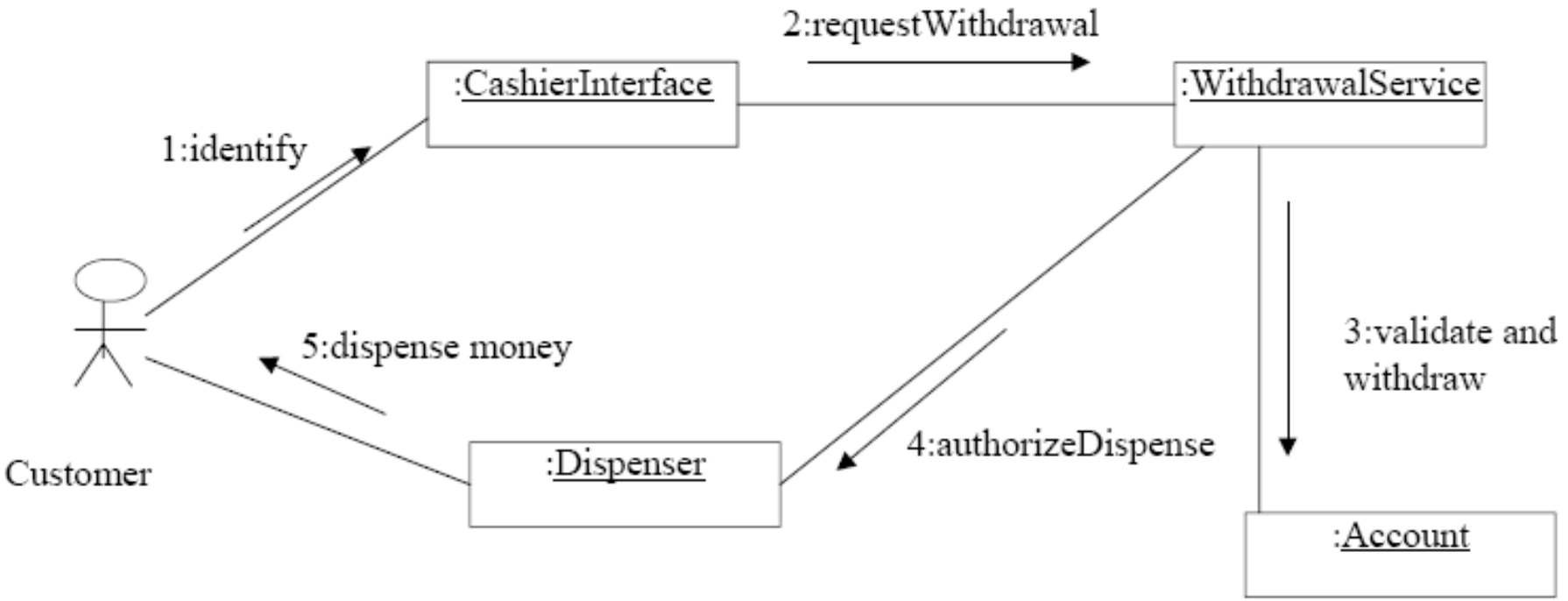
Logical View Example (using UML component diagram)



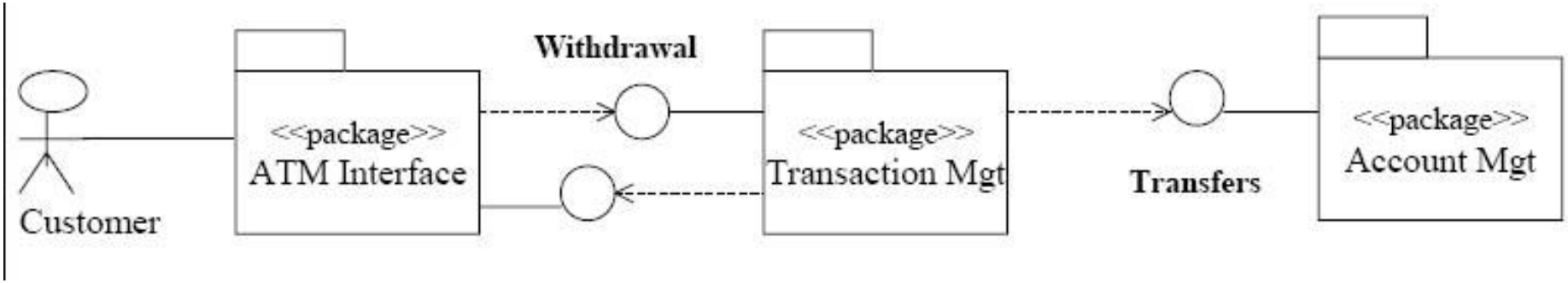
May represent the process view if components are separate processes

- Description
  - CallProcessor: interaction between phone processes
    - Provides an interface to test phones
  - AdminConsole: application logic and UI of admin application
  - Billing: bill generation and etc.
  - Database: persistence functionality
    - Provides a separate interface for hardware-related information
- Variability
  - IDatabase may be a subinterface of IHardwareDatabase

Logical View : Collaboration Diagram
Deposit Use case



Logical View : Package Diagram



Process View

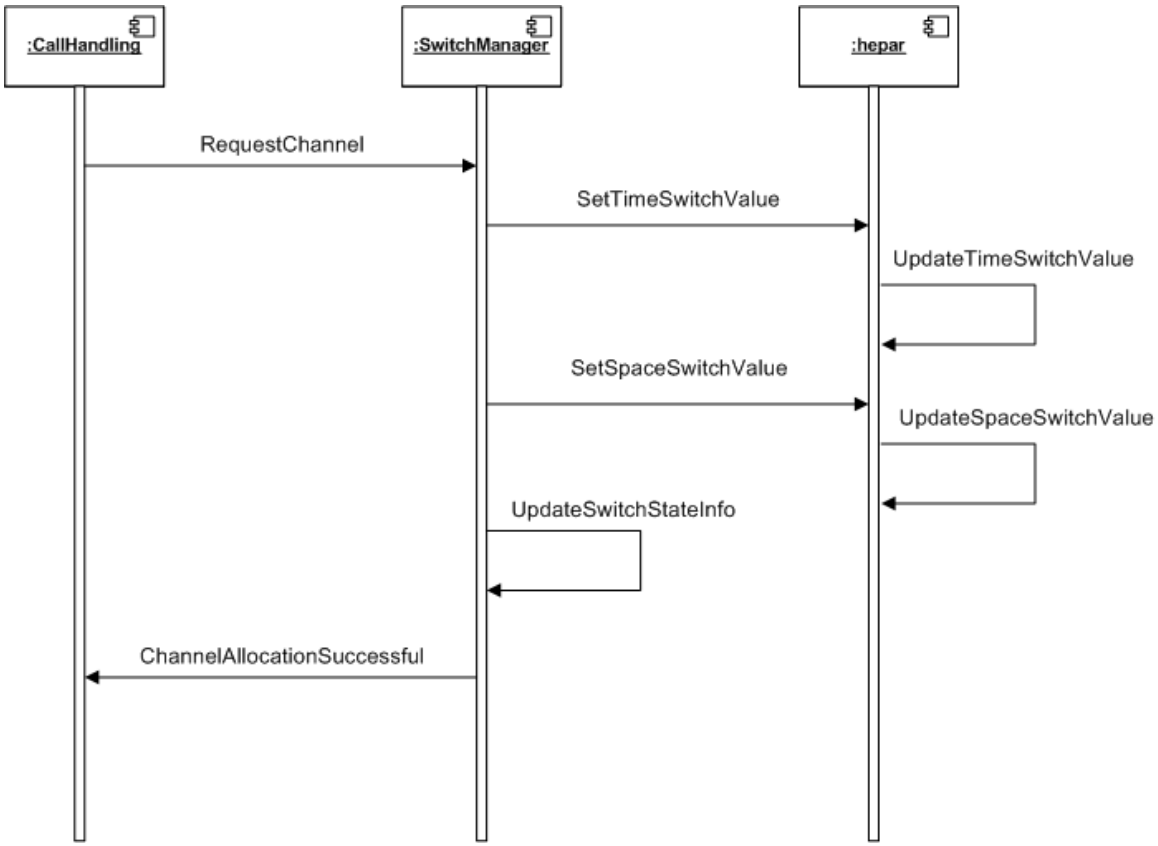
- Static and dynamic organization of executable program units (e.g. processes, threads)
- If each component has its own process, then detailed information (e.g. thread organization and interaction) should be deferred to section 6 (Components)

Process View

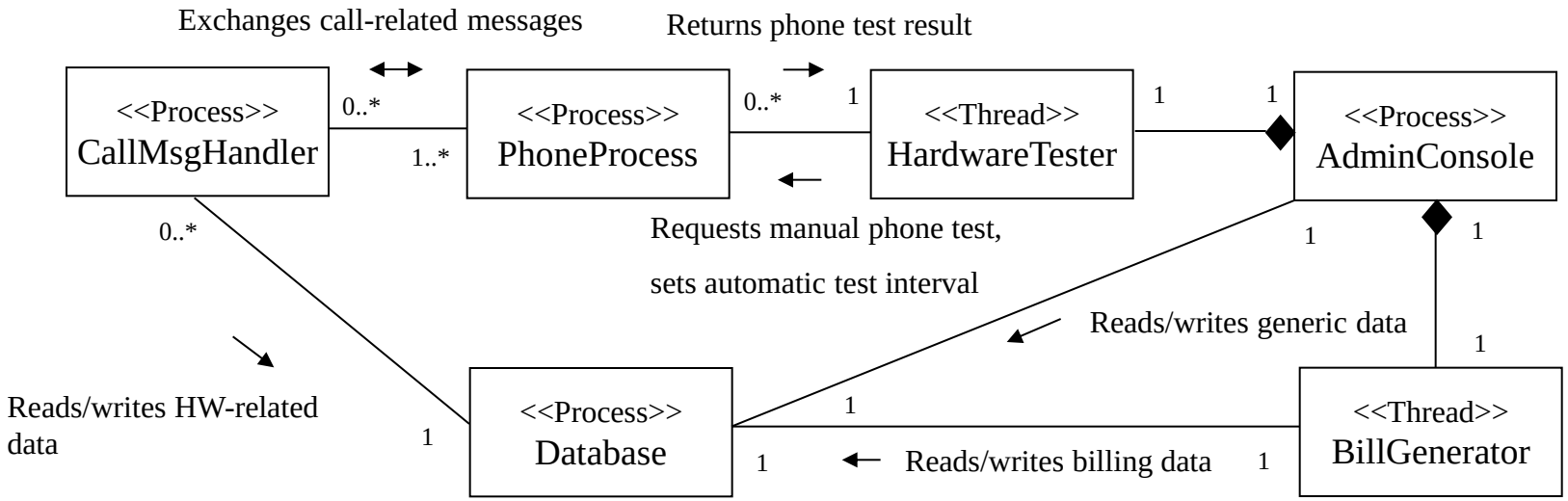
- Consists of the ***processes*** and ***threads*** that form the system's ***concurrency*** and ***synchronization*** mechanisms, as well as their ***interactions***

Example of a Scenario

A request is made by the Call Handler to the Database Manager to acquire a channel



Process View Example



- Note that despite the resemblance, the organization is different from the earlier component diagram
- Explanation parts where necessary (e.g. cardinality of association ends between CallMsgHandler and PhoneProcess)
- Dynamic organization (not shown)
  - Order of execution
  - A more detailed explanation of control/data dependency (e.g. through a scenario)
  - May use activity diagram / state chart

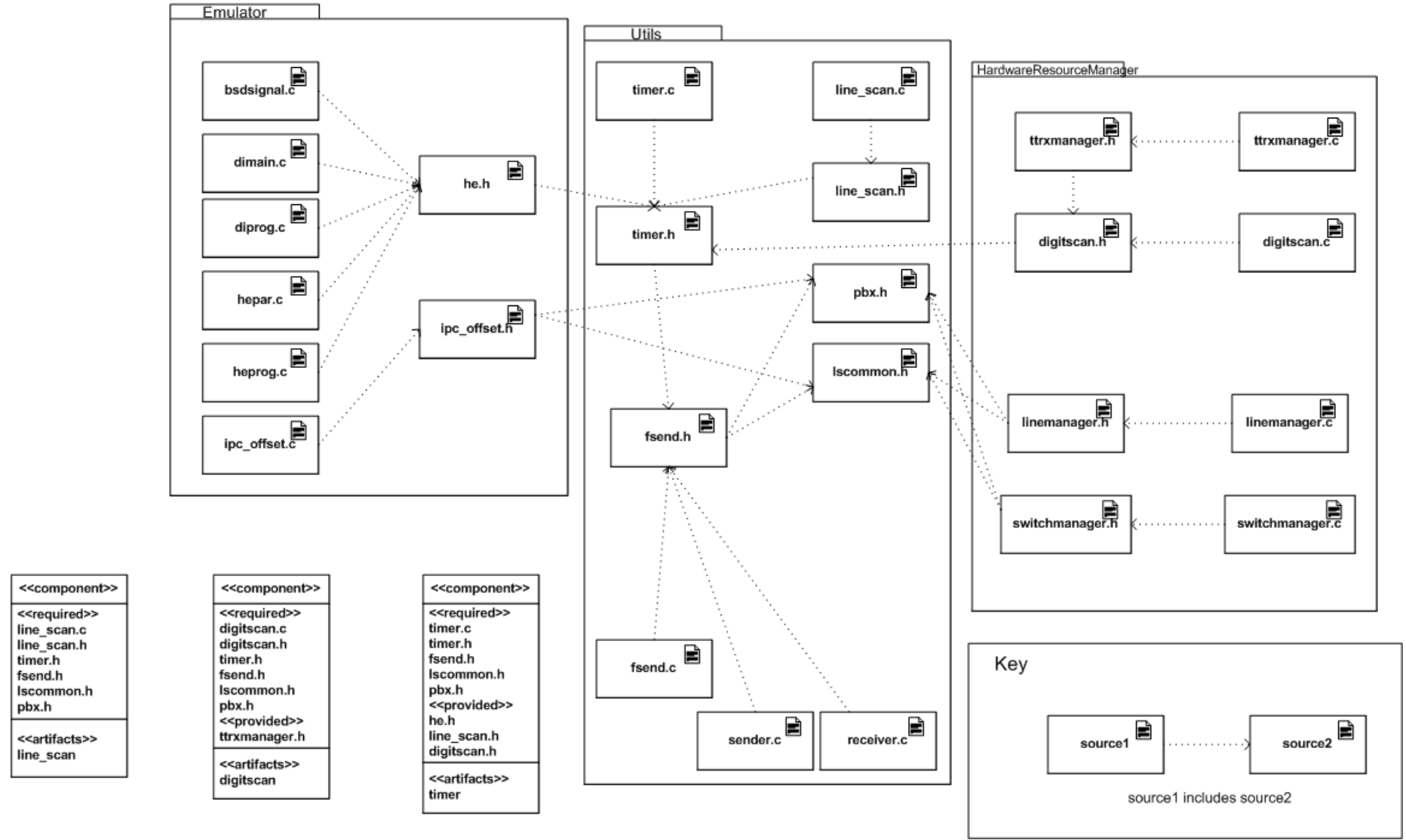
Data Schema View

- Provide a definition of all data structures referenced in the interfaces
- Specify database schemas
- May use UML class diagrams, ER diagrams, ASL, etc.
- Be precise, especially in relationships
  - Cardinality (what is “many”? 1..* or 0..*?)
  - Primary, foreign keys

Development View

- For each component, list
  - Source code artifacts (e.g., header file, C files, java files) used to build it
    - Note that a source artifact can be shared among more than one component (e.g., header files)
  - Deployment artifacts (e.g., exe file, jar archives, etc.) that represent the executable to be deployed on a computing node
  - May also list other artifacts such as documentation, test plans, etc.
- Specify dependencies between artifacts, e.g.,
  - Include dependencies between source artifacts
  - Make dependencies for deployment artifacts
- May use tables and/or UML notation for artifacts and dependencies

Development View Example



Deployment View

- Describes one or more physical network (hardware) configurations on which the software is deployed and run.

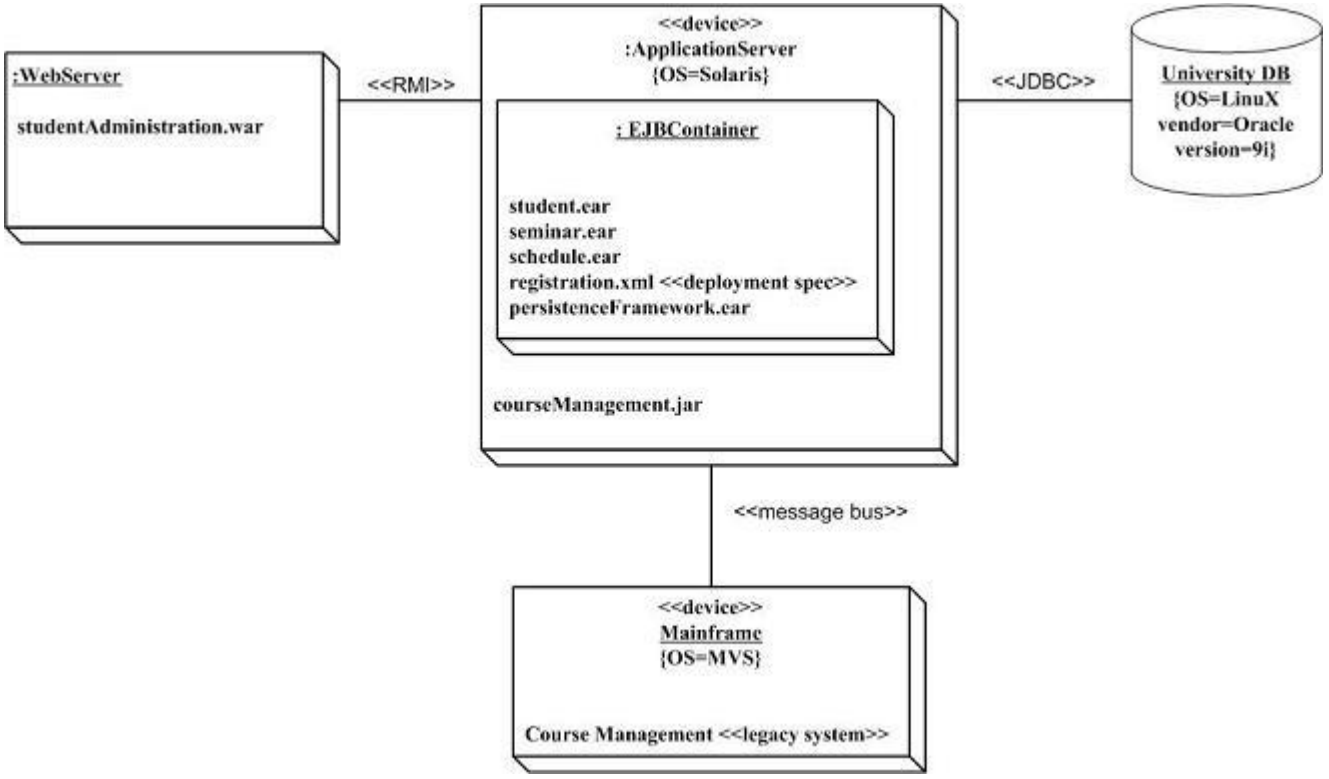
Implementation View

- Describes the **organization of static software modules** (source code, data files, executables, documentation etc.) in the development environment in terms of **Packaging and layering**
- Are modeled using UML Component Diagrams.
- UML components are physical and replaceable parts of a system that conform to and provide the realization of a set of interfaces

Physical View

- Give a list of deployment artifacts; for each artifact, specify any deployment constraints (e.g., required hardware platform and resources)
- Show how deployment artifacts are distributed over computing nodes
- May use UML deployment diagrams

Physical View Example (using UML Deployment Diagram)

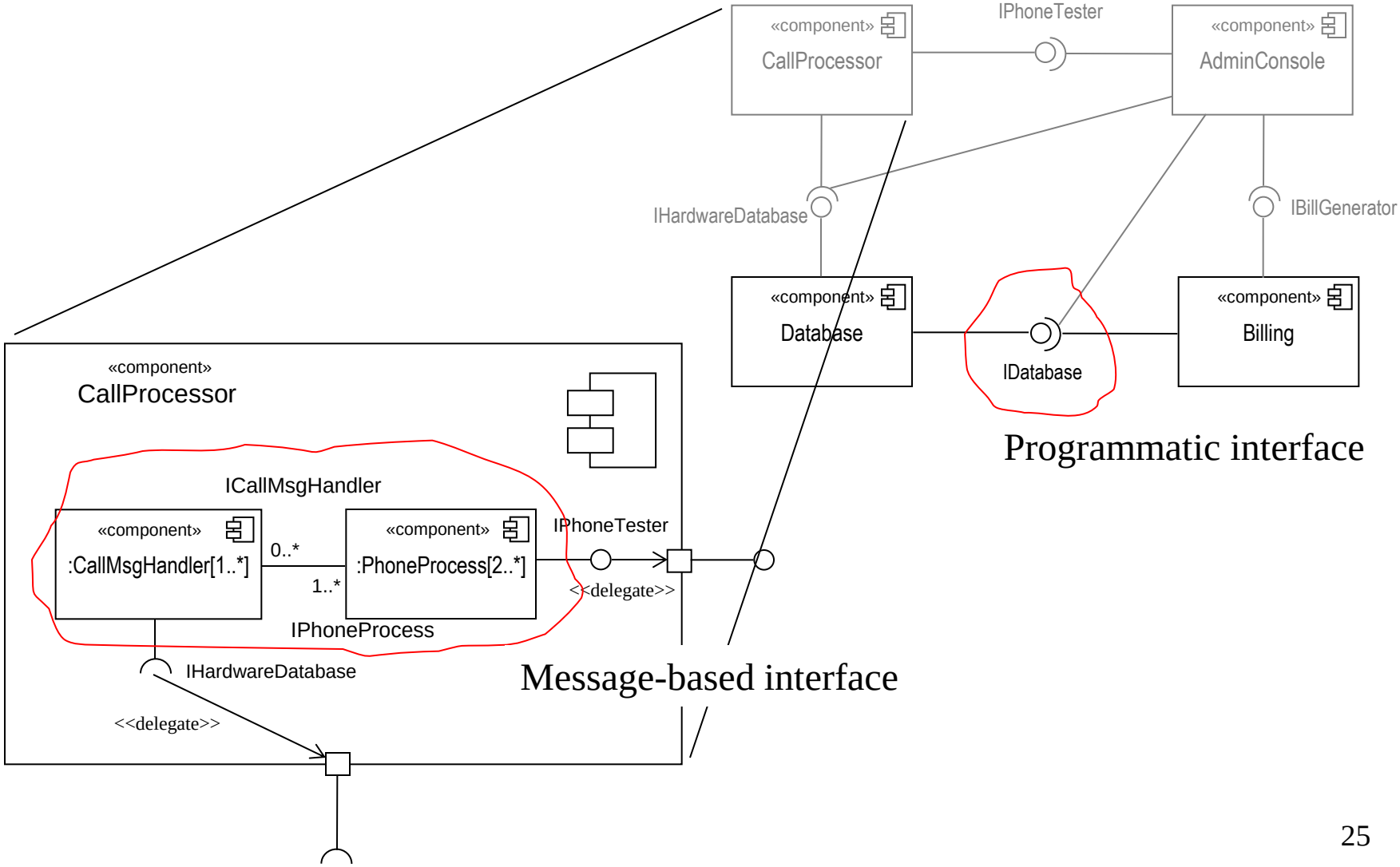


From: Scott Ambler, The Object Primer 3rd Edition: Agile Model Driven Development with UML 2, 2004

Interfaces

- Specification of observable interaction with the environment
- All components have interfaces
- A component's interface contains view-specific information
- Interfaces are two way: provided and required
- Spectrum of interaction constraints
  - Simple existential dependency (e.g. requires resource X) to complex behavioural semantics (e.g. a protocol imposes a state-machine)

Interface Example



Size and Performance

- The number of key elements the system will have to handle (the number of concurrent online users for an airline reservation system, …)
- The key performance measures of the system, such as average response time for key events

Most of these qualities are captured as requirements; they are presented here because they shape the architecture in significant ways and warrant special focus. For each requirement, discuss how the architecture supports this requirement.

Quality

- Operating performance requirements, such as **mean-time between failure** (MTBF).
- Quality targets, such as "no unscheduled down-time"
- Extensibility targets, such as "the software will be upgradeable while the system is running".
- Portability targets, such as hardware platforms, operating systems, languages

Keep in Mind…

- These are just some examples of useful views; however,…
- Some of these views may not be appropriate for a given system
  - The larger the system, the more dramatic the difference between these structures
  - For small systems, the module and conceptual structures may so similar they can be described together
  - E.g., no need for a separate process view if only one process in the system or one process per component
- Some other kinds of views may be required for the system
  - E.g., control flow, data flow, timing, domain-specific views, etc.
- You may also want to document mappings between individual views

Architectural Patterns : Definition

An architectural style or pattern is

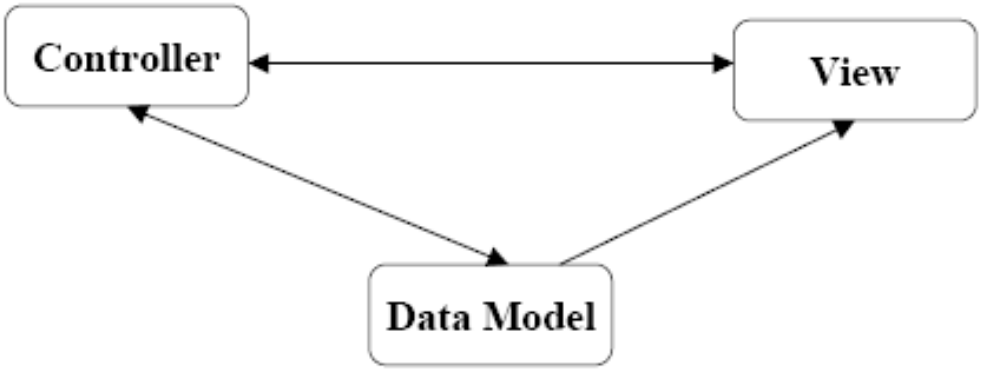
- a description of the component and connector types involved in the style
- the collection of rules that constrain and relate them

Model-View-Controller

- **Context**
  - Provides a flexible structure for developing interactive applications.
- **Problem**
  - User interfaces are subject to changes. As new features are added to the system, the UI must provide appropriate command and menus to handle them.
  - Different platforms support different ‘look and feel’ standards; the UI must be portable.
  - Different kind of users may expect different data format from the UI (bar char, spreadsheet etc.).
- **Solution**
  - Divide the system into three parts: processing, output, and input:
  - ***Model:*** contains the processing and the data involved.
  - ***View:*** presents the output; each view provides specific presentation of the same model.
  - ***Controller:*** captures user input (events-> mouse clicks, keyboard input etc.). Each view is associated to a controller, which captures user input.

Model-View-Controller

- **Main goal**:
  - facilitate and optimize the implementation of interactive systems, particularly those that use multiple synchronized presentations of shared information.
- **Key idea:**
  - separation between the data and its presentation, which is carried by different objects.
- Controllers typically implement event-handling mechanisms that are executed when corresponding events occur.
- Changes made to the model by the user via controllers are directly propagated to corresponding views. The change propagation mechanism can be implemented using the ***observer (design) pattern.***



Layered Pattern

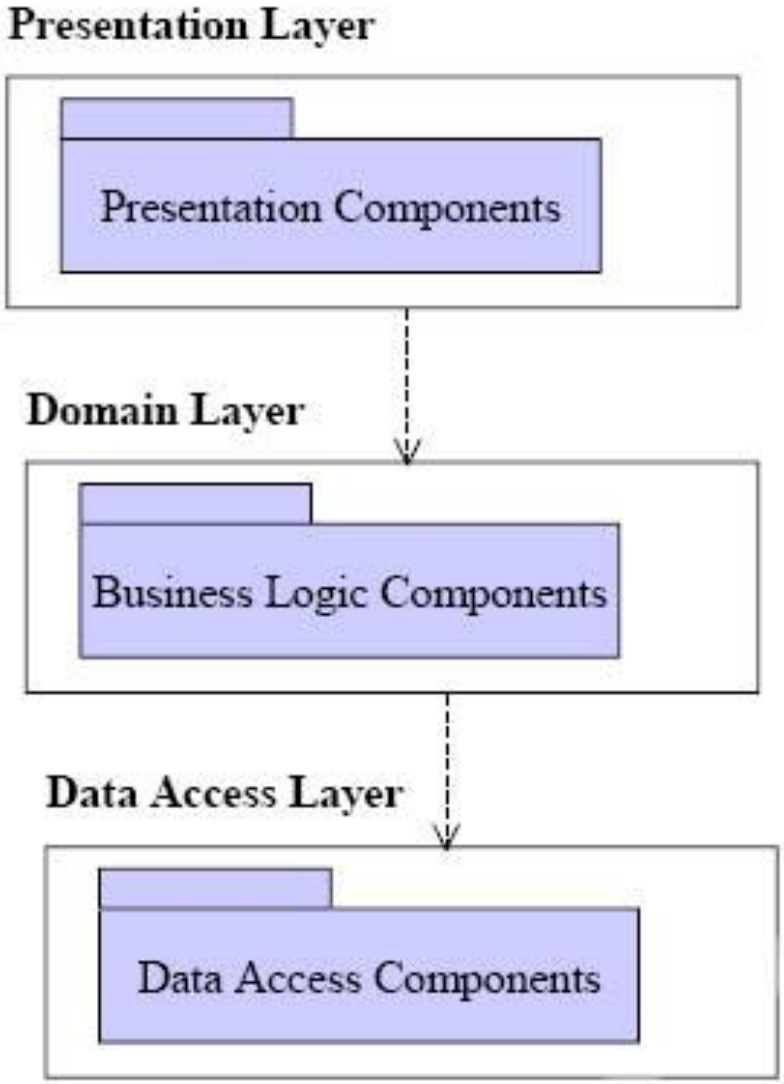
- **Context**
  - You are working with a large, complex system and you want to manage complexity by decomposition.
- **Problem**
  - How do you structure an application to support such operational requirements as maintainability, scalability, extensibility, robustness, and security?
- **Solutions**
  - Compose the solution into a set of layers. Each layer should be cohesive and at Roughly the same level of abstraction. Each layer should be loosely coupled to the layers underneath.

Layered Pattern

- Layering consists of a ***hierarchy of layers***, each ***providing service to the layer above*** it and ***serving as client to the layer below***.
- Interactions among layers are defined by suitable communication protocols.
- Interactions among non-adjacent layers must be kept to the minimum possible.
- Layering is different from composition
  - higher-layers do not encapsulate lower layers
  - lower layers do not encapsulate higher layers (even though there is an existence dependency)

Three-Layered Pattern

- **Context**
  - You are building a business solution using layers to organize your application.
- **Problem**
  - How do you organize your application to reuse business logic, provide deployment flexibility and conserve valuable resource connections?
- **Solutions**
  - Create three layers: ***presentation, business logic*** and ***data access***.
  - Locate all database-related code, including database clients access and utility components, in the data access layer.
  - Eliminate dependencies between business layer components and data access components.
  - Either eliminate the dependencies between the business layer and the presentation layer or manage them using the ***Observer*** pattern.


Presentation Layer
Presentation Components
Domain Layer
Business Logic Components
Data Access Layer
Data Access Components