ENSF 607

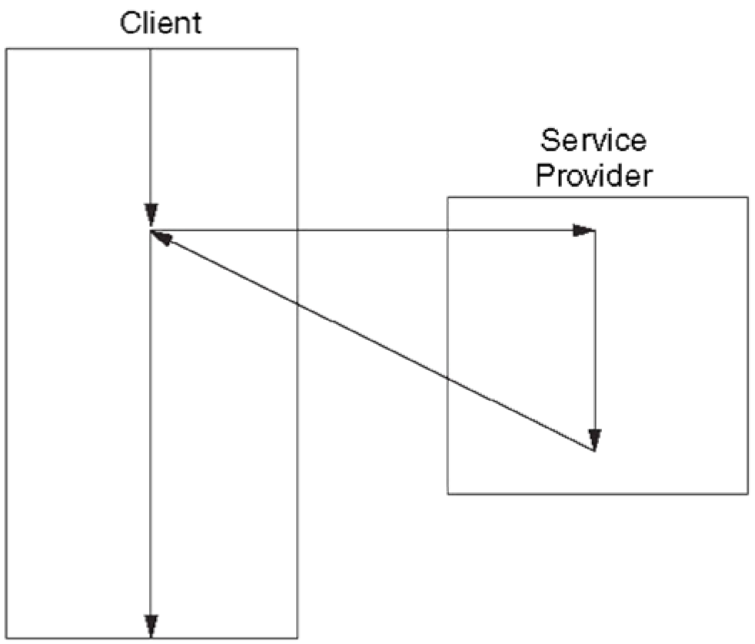
Architectural Patterns (Client-Server)

Architectural Styles

Architectural Style	*Architectural Idiom*
AP1: Data flow systems	Batch sequential Pipes and filters
AP2: Call-and-return systems	Main program and subroutines Client–server systems Object-oriented systems Hierarchical layers
AP3: Independent components	Communicating processes Event-based systems
AP4: Virtual machines	Interpreters Rule-based systems
AP5: Data-centered systems	Database systems Blackboards

Call & Return System

- A *Call & Return style* is a synchronous software architecture in which the client ceases execution while the service provider performs the service. Upon completing the service, the service provider may return a value to the client.
- *Call & Return* style is used when the order of computation is fixed. ← opposite to rule-based systems



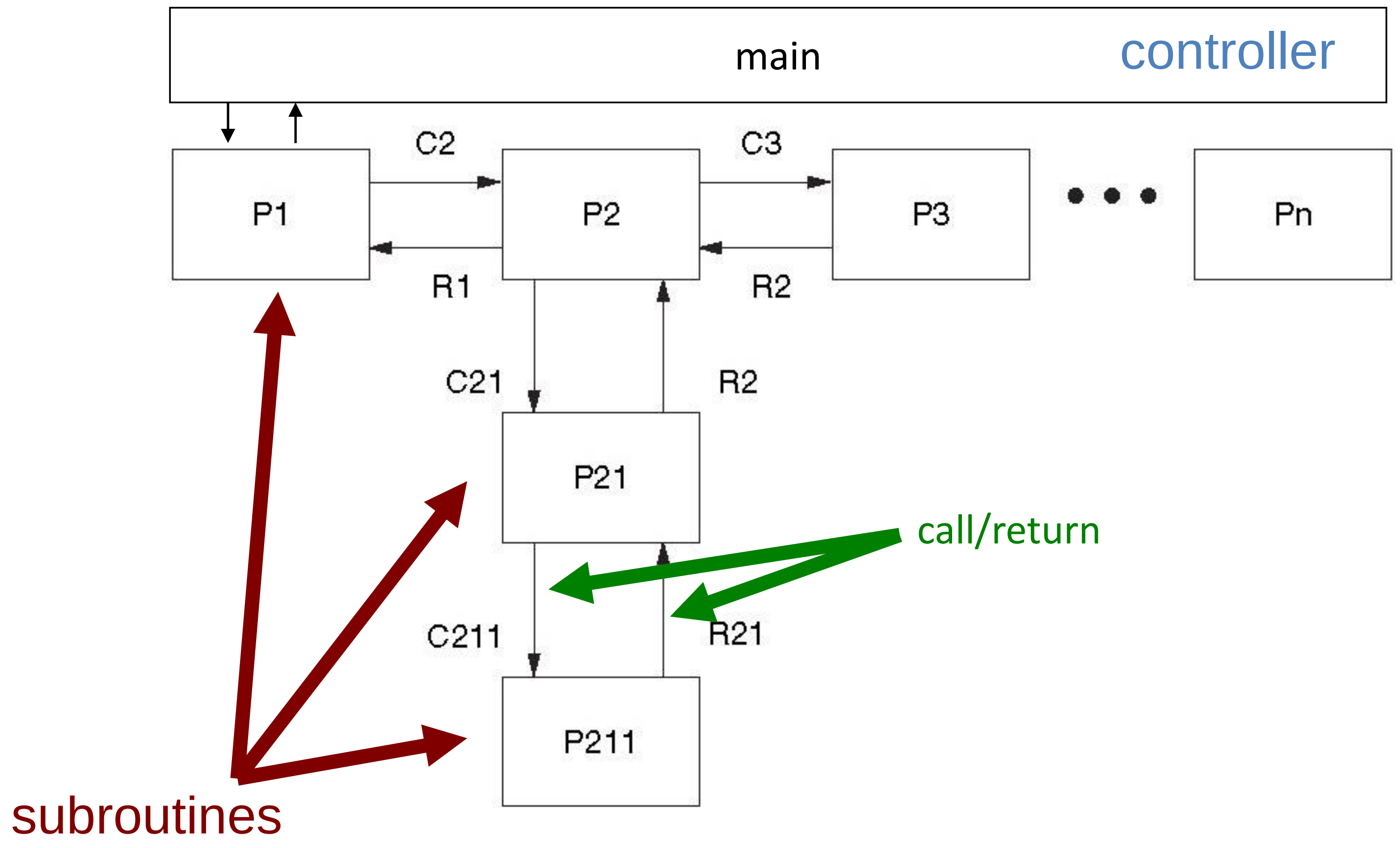
Call-and-return control flow

Ref: Stephen H. Kaisler: Software Paradigms, ch 13, Wiley, 2005

AP2.1 Main Program/Subroutine

- The *main program and subroutines* architecture consists of a main program that acts as a controller (or executive) and one or more subroutines that perform specific functions when called by the main program.
- This architecture is based on the definition–use relationship, which imposes a single thread of control.
- The correct operation of the main program and any subroutine is directly dependent on the correct operation of the subroutines it calls.

Main Program/Subroutine



Ref: Stephen H. Kaisler: Software Paradigms, ch 13, Wiley, 2005

Main Program/Subroutine: Essentials

- Hierarchical decomposition:
  - Based on definition-use relationship
- Single thread of control:
  - Supported directly by programming languages
- Subsystem structure implicit:
  - Subroutines typically aggregated into modules
- Hierarchical reasoning:
  - Correctness of a subroutine depends on the correctness of the subroutines it calls

Main Program/Subroutine: Advantages & Disadvantages

- Advantages:
  - Simple to visualize and easy to learn.
  - A single thread of control works its way through the various subroutines.
- Disadvantages:
  - Correctness of any job depends on all subroutines called.
  - Tendency to devolve into spaghetti code: several subroutines may do similar job due to similar requirements added at different times in the application’s lifetime.
  - Scalability & configuration control: one can lose visibility of the flow of control over several subroutines

Master–Slave Architecture

- Variant of the *main program/subroutines* architecture.
- **Definition:** A coordinating (control) process that distributes jobs to one or more slave processes. When a slave process finishes its job, it asks the master process for more jobs.
- **Requirement:** the master must know location, configuration and capabilities of the slaves.

Example: we could build the IS2000 example system's image processing unit as a master-slave instead of filter-pipe.

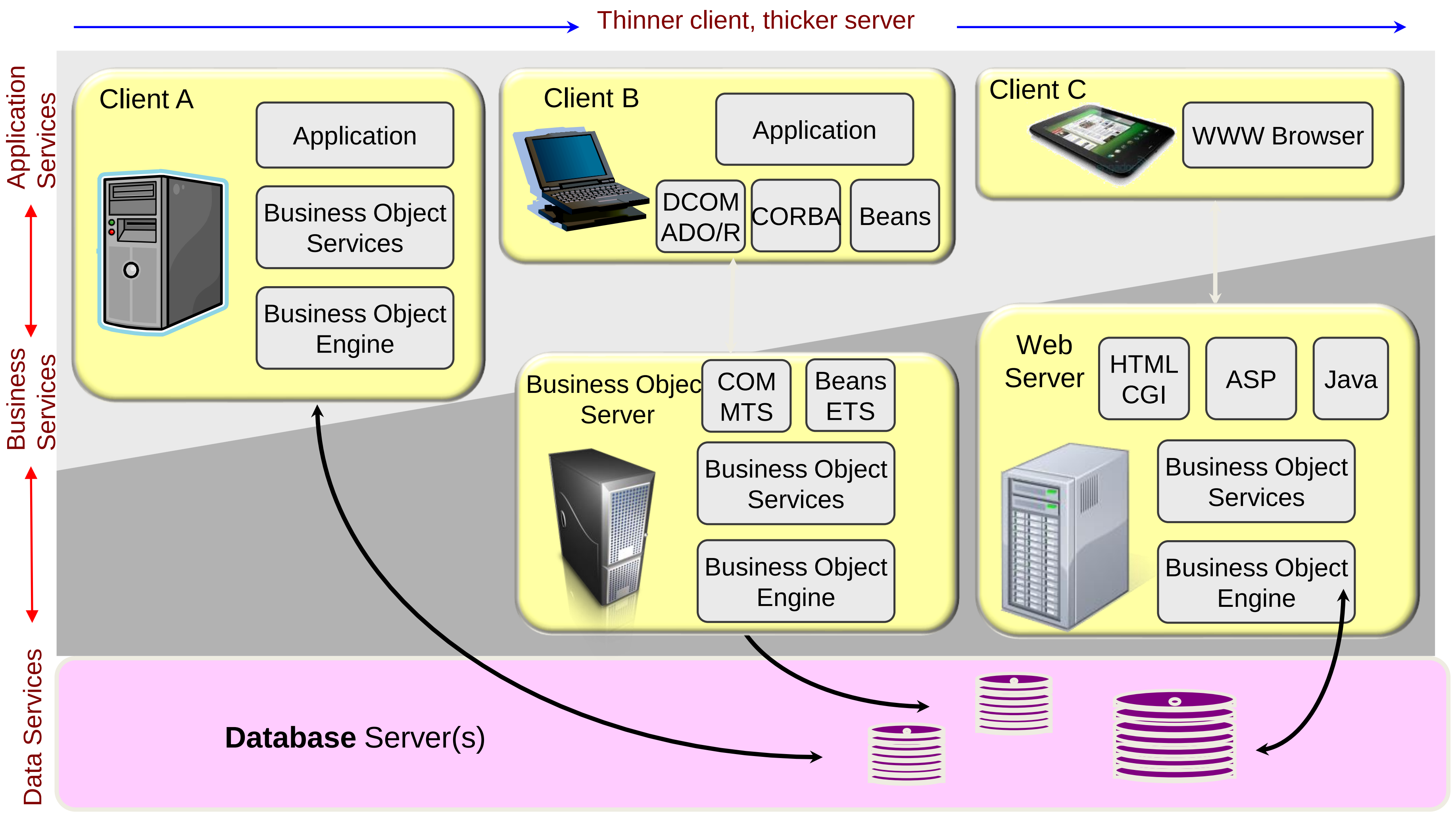
Client–Server & P2P Systems

- Both variant of the *main program/subroutines* architecture.
- The primary difference is the assumption that the client and server are located physically on different computer systems. ← requires remote access
- The *client* is an entity that makes a request for a service. The *server* is an entity that can provide that service upon request.
- Requires *existing implementation of protocols* between client and server that may restrict the flexibility of the interaction between the two.

Client–Server & P2P Systems

- Client/Server
  - **Typically 3-tier Systems:** Functionality is divided into three physical partitions: application, business, and data services.
  - **Fat Client:** More functionality is placed on the client.
  - **Thin Client:** More functionality is placed on the server.
  - **Distributed Client/Server**
- Peer-to-Peer
  - Any process or node in the system may be both client and server.

Client/Server Architectures



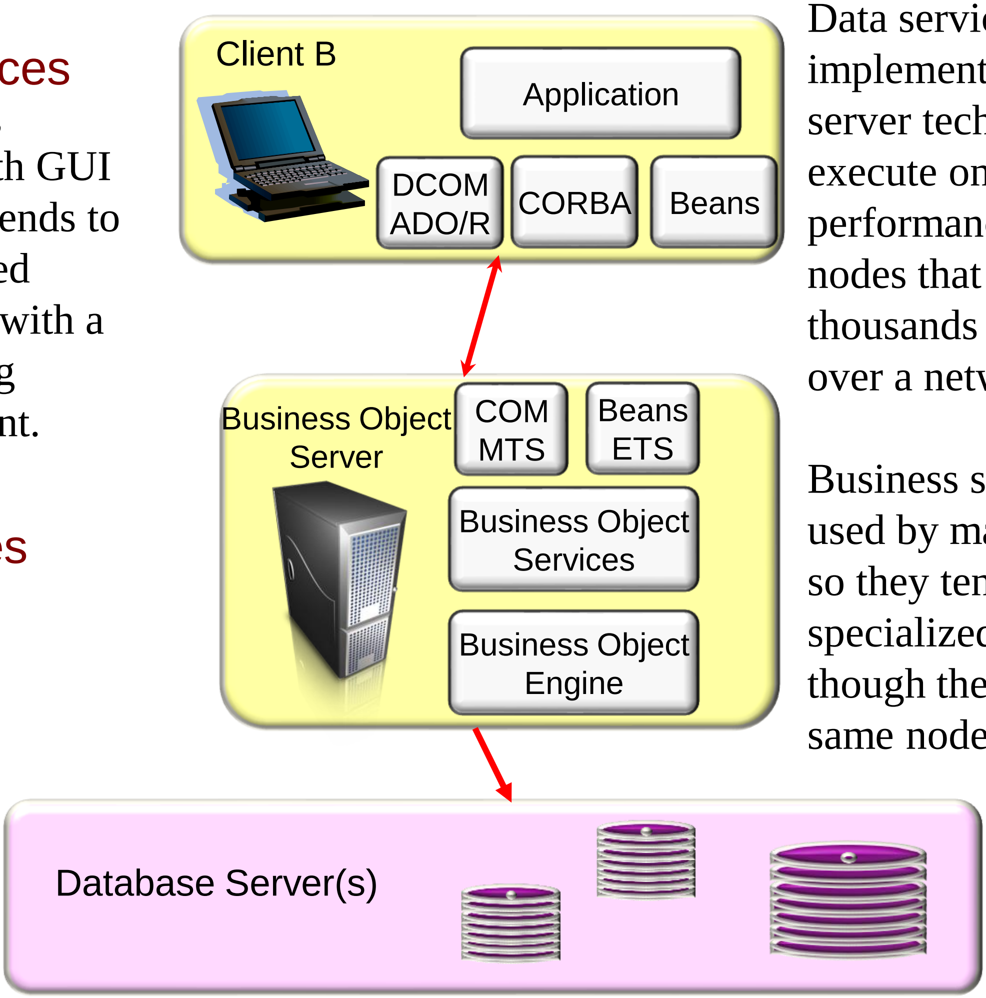
Client/Server: 3-Tier Style

Application Services

Application services, primarily dealing with GUI presentation issues, tend to execute on a dedicated desktop workstation with a graphical, windowing operating environment.

Business Services

Data Services



Data services tend to be implemented using database server technology, which tends to execute on one or more high-performance, high-bandwidth nodes that serve hundreds or thousands of users, connected over a network.

Business services are typically used by many users in common, so they tend to be located on specialized servers as well, though they may reside on the same nodes as the data services.

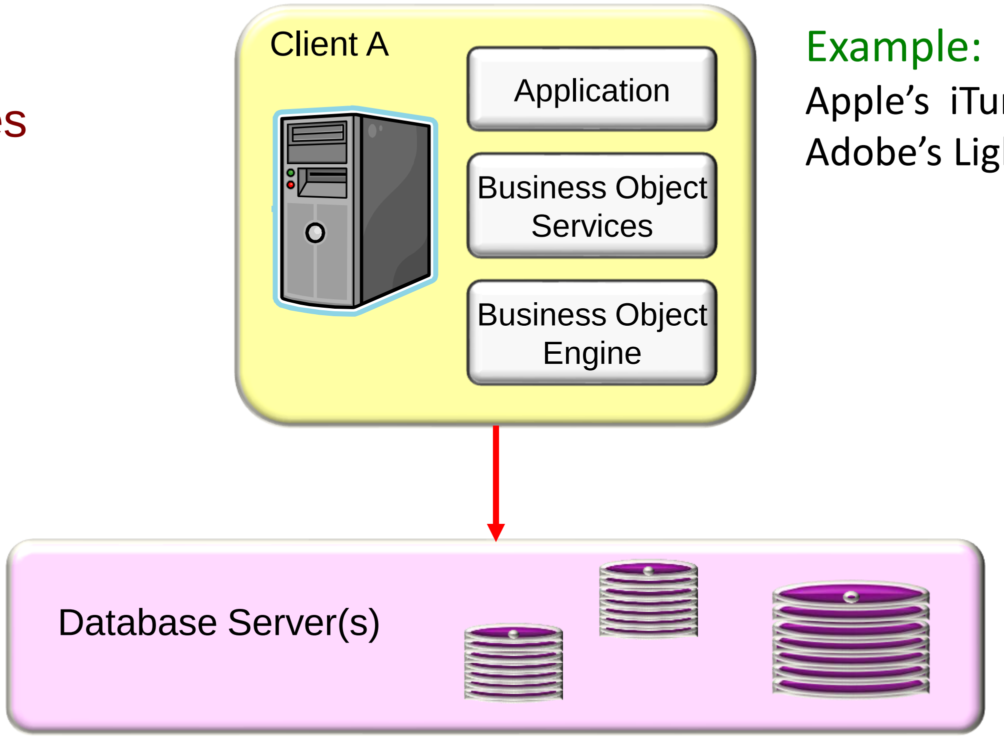
Client/Server: "Fat Client"

The client is "Fat" since nearly everything runs on it (except in a variation, called the "two-tier architecture," in which the data services are located on a separate node). Application Services, Business Services and Data Services all reside on client computers. The database server is usually on another computer.

Application Services

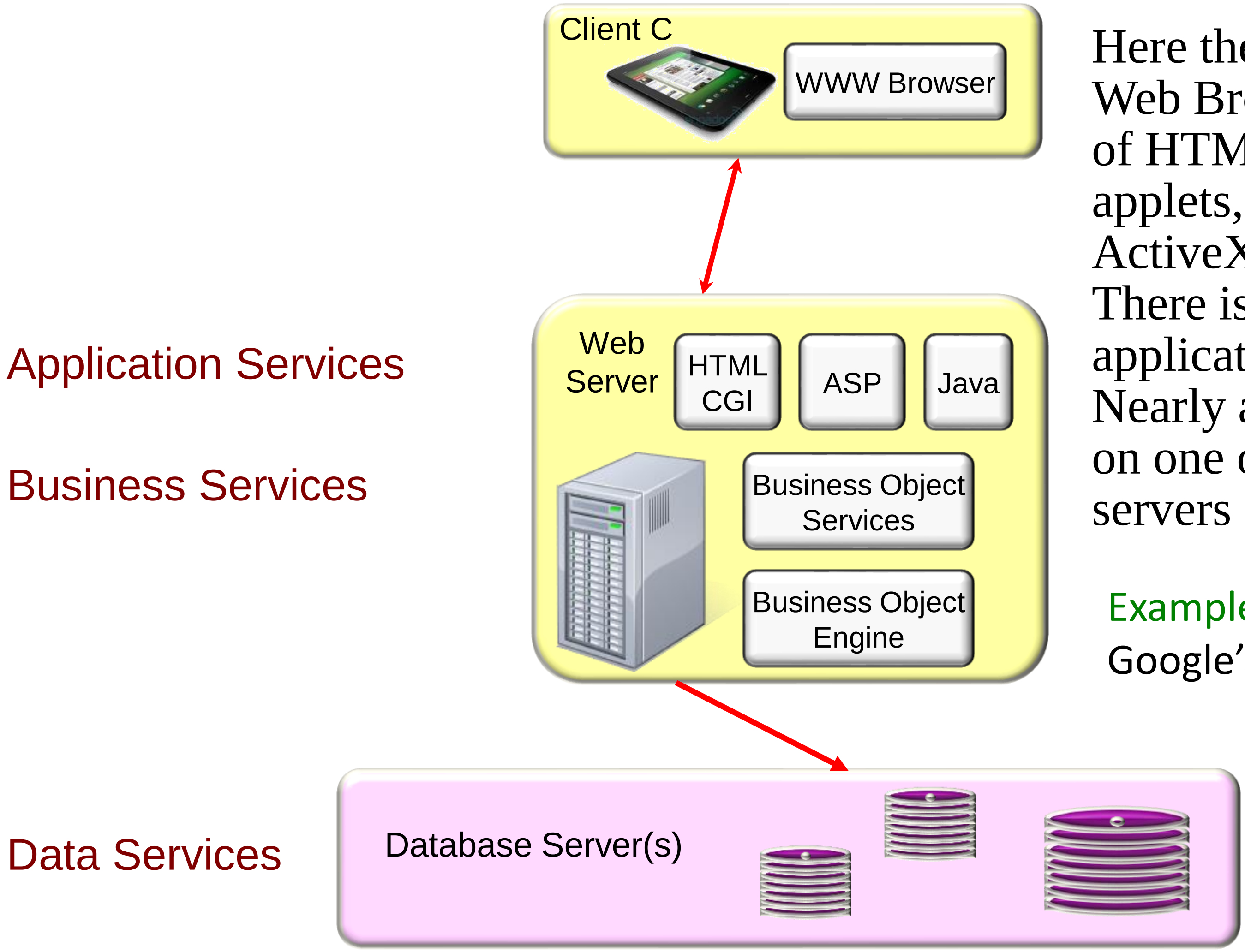
Business Services

Data Services



Example:
Apple's iTunes
Adobe's Lightroom

Client/Server: Web Appli Style



Here the client is simply a Web Browser running a set of HTML pages and Java applets, Java Beans or ActiveX components, etc. There is very little application there at all. Nearly all work takes place on one or more Web servers and data services.

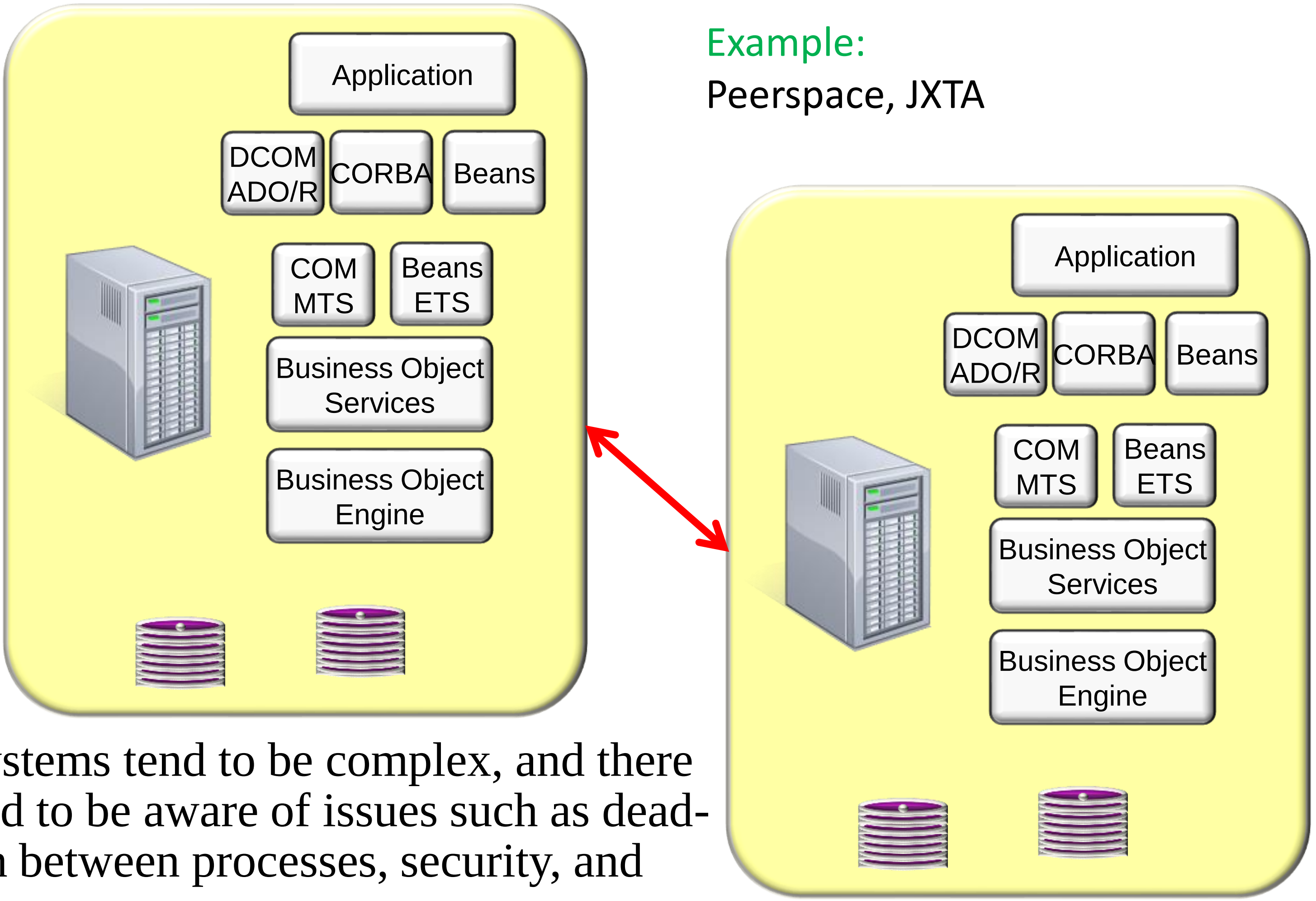
Example:
Google's Chromium

Peer-to-Peer Style

Application Services

Business Services

Data Services



Example:
Peerspace, JXTA

Peer-to-peer systems tend to be complex, and there is a greater need to be aware of issues such as dead-lock, starvation between processes, security, and fault handling.

Type of Connection Protocols

- ***Connectionless communication:*** e.g. via UDP (User Datagram Protocol), implements unreliable delivery of messages. Clients should use connectionless communication when the application service handles errors and participates in broadcast or multicast service, or performance requirements cannot tolerate virtual circuit overhead delays.
- ***Connection-oriented communication:*** e.g. via TCP/IP (Transfer Control Protocol/Internet Protocol), implements reliable delivery of messages. Connection oriented communication makes programming easier because the protocol includes mechanisms for detecting and handling errors and an acknowledgment mechanism between client and service.

Stateless vs. Stateful Servers

- ***State information:*** Information a server maintains about its clients. A server may or may not retain this information.
- State information may be required if the information exchange between the client and the application consists of multiple messages. State information may reduce the size of succeeding messages and may allow the server to respond more quickly. State information can become useless if messages are lost, duplicated, or delivered out-of-order, or if the server crashes and must be rebooted.
- *Stateless servers* rely on the application protocol to assume the responsibility for reliability of delivery and service. Thus, the application gives the same response no matter how many times a request arrives.
- Stateful designs can lead to complex application protocols and error handling mechanisms.

Client–Server: Issues

- In designing client-server systems the following issues must be considered:
  - ***Authentication:*** Verifying the identity of the client and its rights/privileges. Clients may be restricted to the systems they can access when multiple servers are available.
  - ***Authorization:*** Verifying the rights/privileges of the client. Clients may be restricted in the kinds of commands they may issue, the applications they can run, or the data they can retrieve.
  - ***Data Security:*** Protecting the data stored on the server. Servers must provide protection to prevent unauthorized release or modification of the data stored on them.
  - ***Protection:*** Protecting the system itself from errant applications. Servers must provide mechanisms for trapping errant applications and preventing them from damaging the system and its resources.
  - ***Middleware:*** selecting the right middleware for the task.

Data Abstraction Style

- Data Abstraction (Object-Oriented) style is a special case of the main program/subroutines architecture.
- key differences:
  - Objects are encapsulated so that communication occurs through a message-passing or procedure call mechanism.
  - Object systems also support inheritance and/or delegation mechanisms not necessarily available in program/subroutines architecture.

ADT & OO-Systems: Essentials

- Encapsulation:
  - "Manager" is responsible for preserving integrity of a resource
  - Restricted access to certain information
- Inheritance:
  - Share one definition of shared functionality
- Dynamic binding:
  - Determine actual operation to invoke at runtime
- Management of many objects:
  - Provide structure on large set of definitions
- Reuse and maintenance:
  - Exploit encapsulation and locality

ADT & OO Systems: Advantages

- Data hiding → easy maintenance of objects without affecting the clients that use these objects
- Supporting concurrent execution
- Natural problem decomposition → collection of interacting objects

ADT & OO Systems: Drawbacks

Main issue:

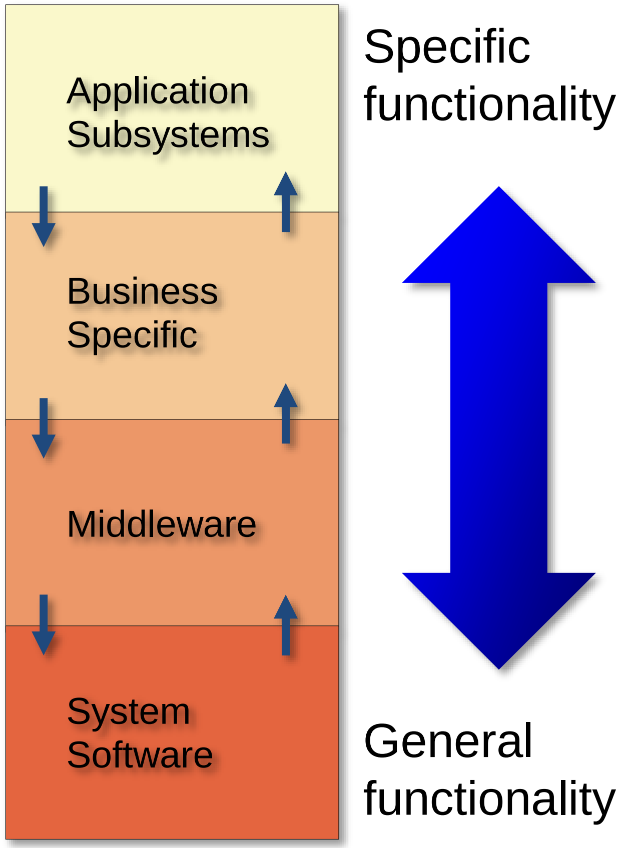
In order for one object to interact with another it must know the identity of that other object

- If identity of an object changes, it is necessary to modify all other objects that explicitly invoke it
- Potentially side-effect problems: if A uses object B and C also uses B, then C's effects on B look like unexpected side-effects to A, and vice versa

Note: both problems can be limited if discipline is used in implementation!

Layered Systems

- A *hierarchically layered system* consists of a large software application that is partitioned into layers.
- Each layer acts as a virtual machine for the layers above it, providing services to those layers. In turn, it acts as a client to the layers below it.
- Communication between layers occurs through well-defined APIs.



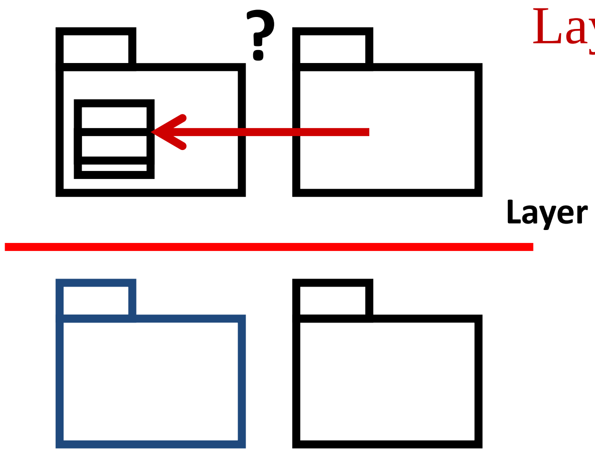
Layering Considerations

- Visibility
  - Dependencies only within current layer and the next lower layer
- Volatility
  - Upper layers affected by requirements changes
  - Lower layers affected by environment changes
- Generality
  - More abstract model elements in lower layers
- Number of layers
  - Small system: 3-4 layers
  - Complex system: 5-7 layers

Goal is to reduce coupling and to ease maintenance effort

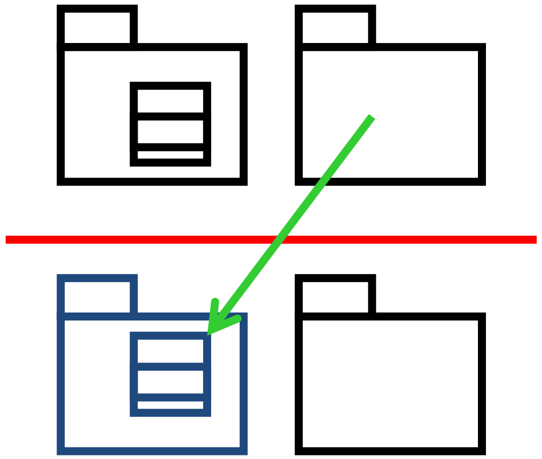
Layers: Reuse Opportunities

Layers can emerge from organizing reusable classes



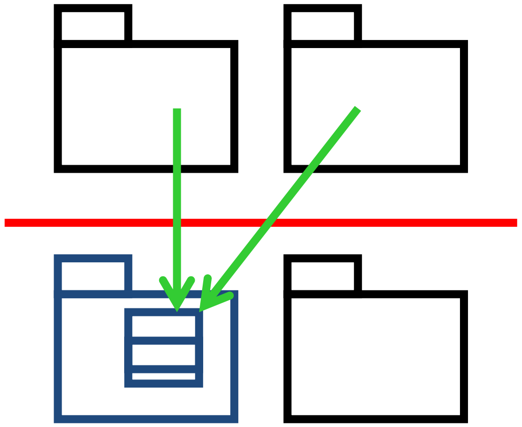
1st application

Build the first application and some general parts.



2nd application

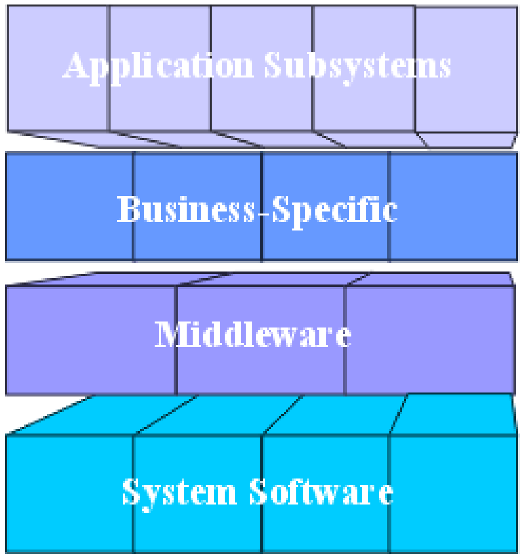
Build the second application and find which parts of the first can be reused.



Take the candidate reusable design elements (classes, packages, or subsystems) and make them reusable.

Layering Defects

•



Distinct Application subsystems that make up an application - contain the value adding software developed by the organization.

Business Specific- contains a number of reusable subsystems specific to the type of business.

Middleware - offers subsystems for utility classes and platform-independent services for distinguished object computing in hetero-geneous environments and so on.

System software - contains the software for the actual infrastructure such as operating systems, interfaces to specific hardware, device drivers and so on.

}

Layer Definitions and rationale defined

Examine the layers of the system using the following criteria:

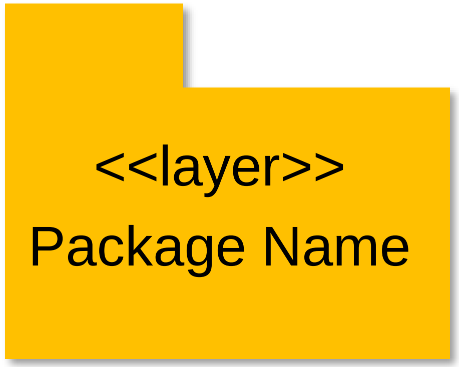
1. There are no more than seven (plus or minus two) layers.
2. The rationale for each layer definition is clearly presented and consistently applied.
3. Layer boundaries are respected within the design.
4. Layers are used to encapsulate conceptual boundaries between different kinds of services and provide abstractions to make the design easier to understand.

Modeling Architectural Layers

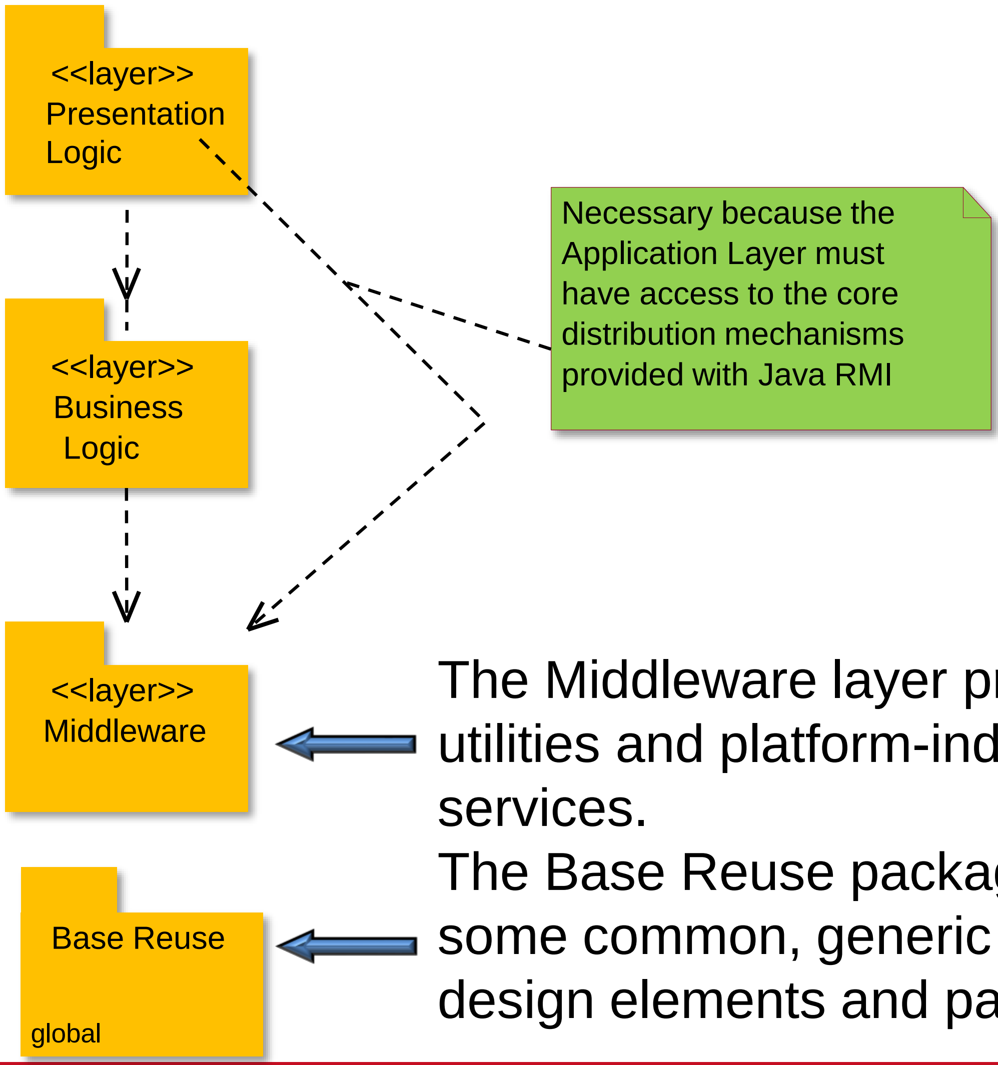
- Architectural layers can be modeled using stereotyped packages

  <<layer>> stereotype

- Layers can be represented in Rose as packages with the <<layer>> stereotype. The layer descriptions can be included in the documentation field of the specification of the package.
- Since a layer can be modeled as a package it can appear in a class diagram or a use-case diagram.



Layers Example /1

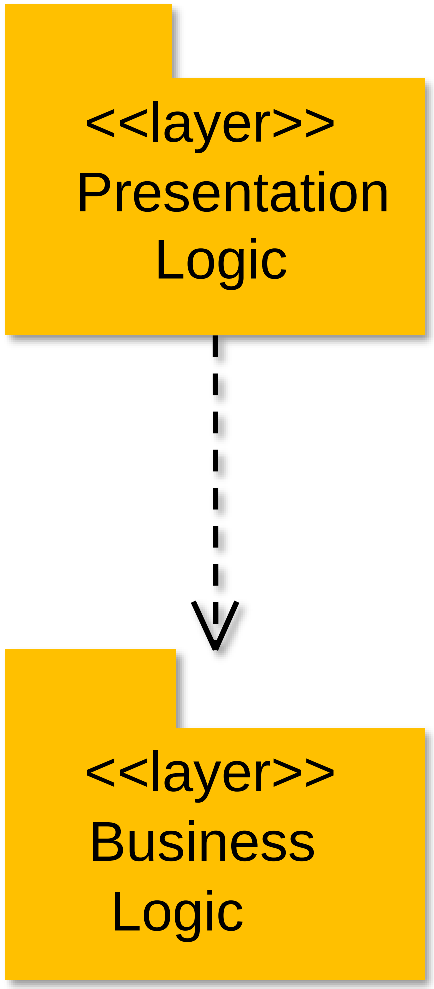


Necessary because the Application Layer must have access to the core distribution mechanisms provided with Java RMI

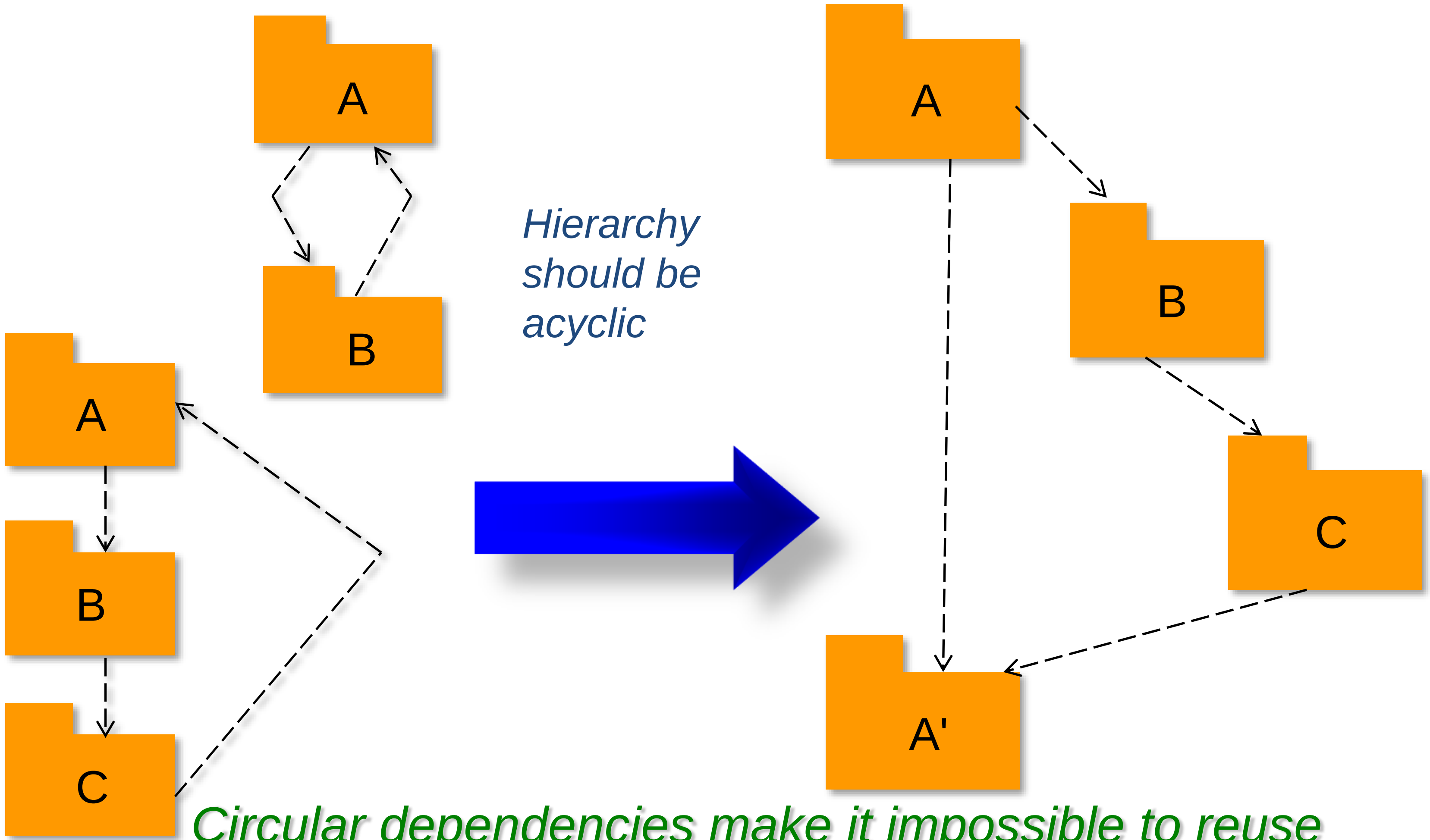
The Middleware layer provides utilities and platform-independent services.
The Base Reuse package contains some common, generic reusable design elements and patterns.

Layers Example /2

- The Presentation Logic layer contains the design elements that are specific to the presentation of the application.
- We expect that multiple applications will share some key abstractions and common services. These have been encapsulated in the Business Logic layer, which is accessible to the Presentation Logic layer.
- The Business Logic layer contains business-specific elements that can be used in several applications, not necessarily just this one.



Avoiding Circular Dependency

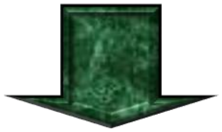


Hierarchy should be acyclic

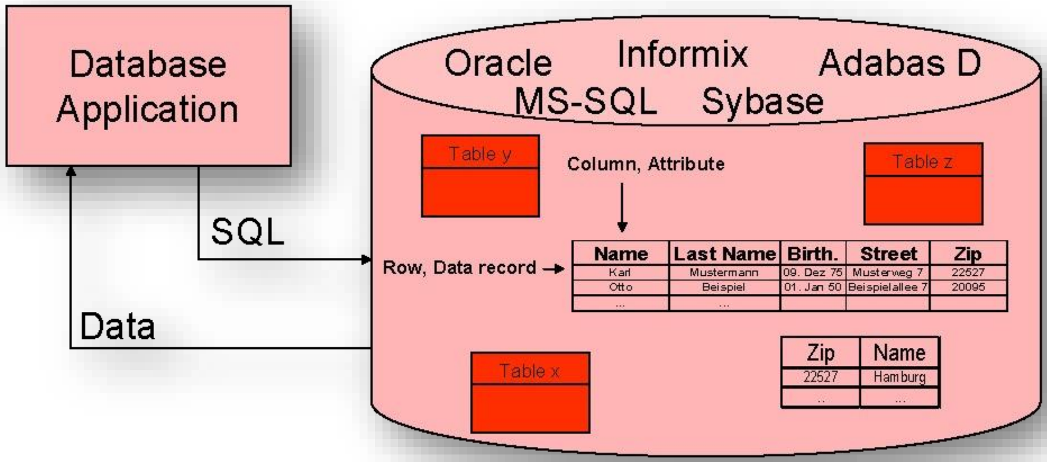
Circular dependencies make it impossible to reuse one package without the other

Layered System: Example /1

Access to a certain types of relational database from a certain application



Access to several types of relational databases from one (or more) application (for every operating system)



Name	Last Name	Birth.	Street	Zip
Karl	Mustermann	09. Dez 75	Musterweg 7	22527
Otto	Beispiel	01. Jan 50	Beispielallee 7	20095
...	...			

Zip	Name
22527	Hamburg
..	...

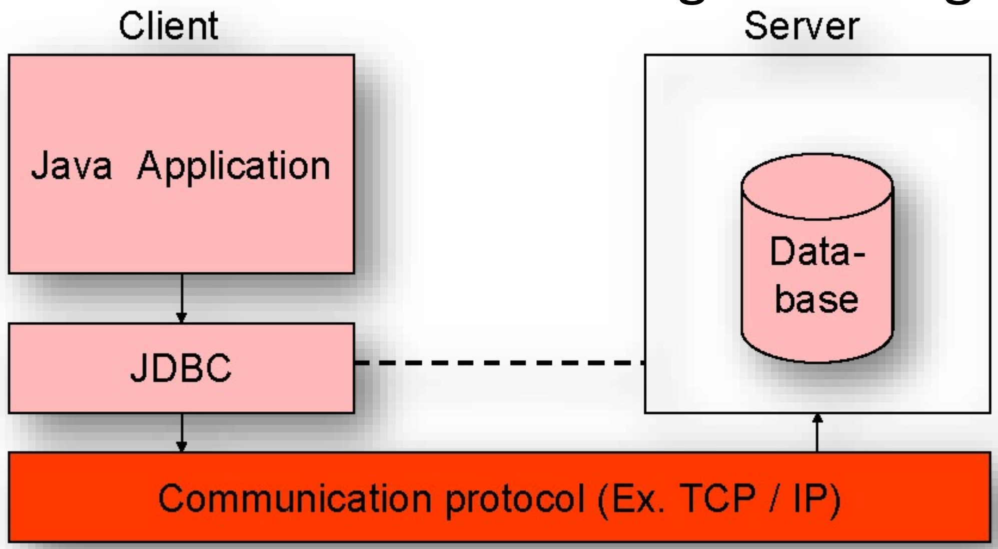
REF: Hans-Werner Sehring & Miguel Garcia: Arbeitsbereich Softwaresysteme (STS), TU Hamburg-Harburg

Layered System: Example /2

Access via Java Database Connectivity (JDBC)

- Class library for access to SQL databases; package java.sql
- SQL commands are sent to a database as a string with help of JDBC methods
- The database executes the SQL commands and delivers the requested data and success or failure messages
- Data delivered from the database is read again through JDBC methods

What if the database is not JDBC compliant?



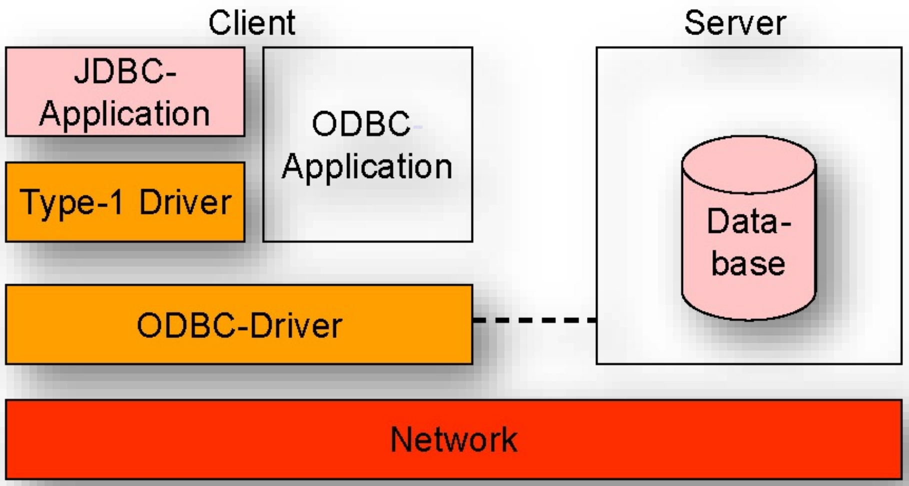
REF: Hans-Werner Sehring & Miguel Garcia: Arbeitsbereich Softwaresysteme (STS), TU Hamburg-Harburg

Layered System: Example /3

Access via Type-1 Driver (JDBC-ODBC Bridge)

- Used for communication with the ODBC-Driver available to the database
- **Advantage**:
  - Same Type-1 JDBC driver can communicate with every database system, for which a ODBC driver is available

- Disadvantages:
  - ODBC driver used by the JDBC driver depends on the operating system on which the database application runs and on the database system used
  - If the same JDBC application is deployed on another operating system, an ODBC driver for this operating system is needed



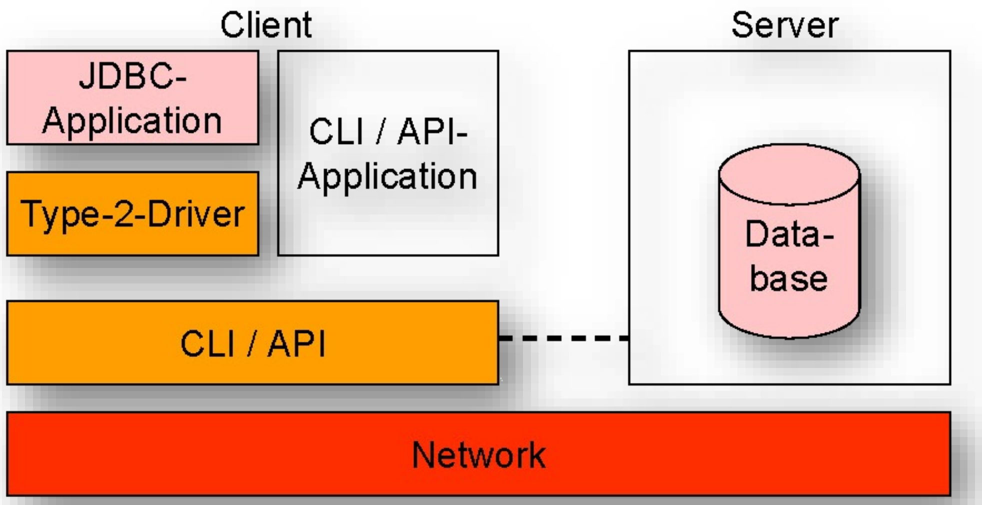
REF: Hans-Werner Sehring & Miguel Garcia: Arbeitsbereich Softwaresysteme (STS), TU Hamburg-Harburg

Layered System: Example /4

Access via Type-2 Driver

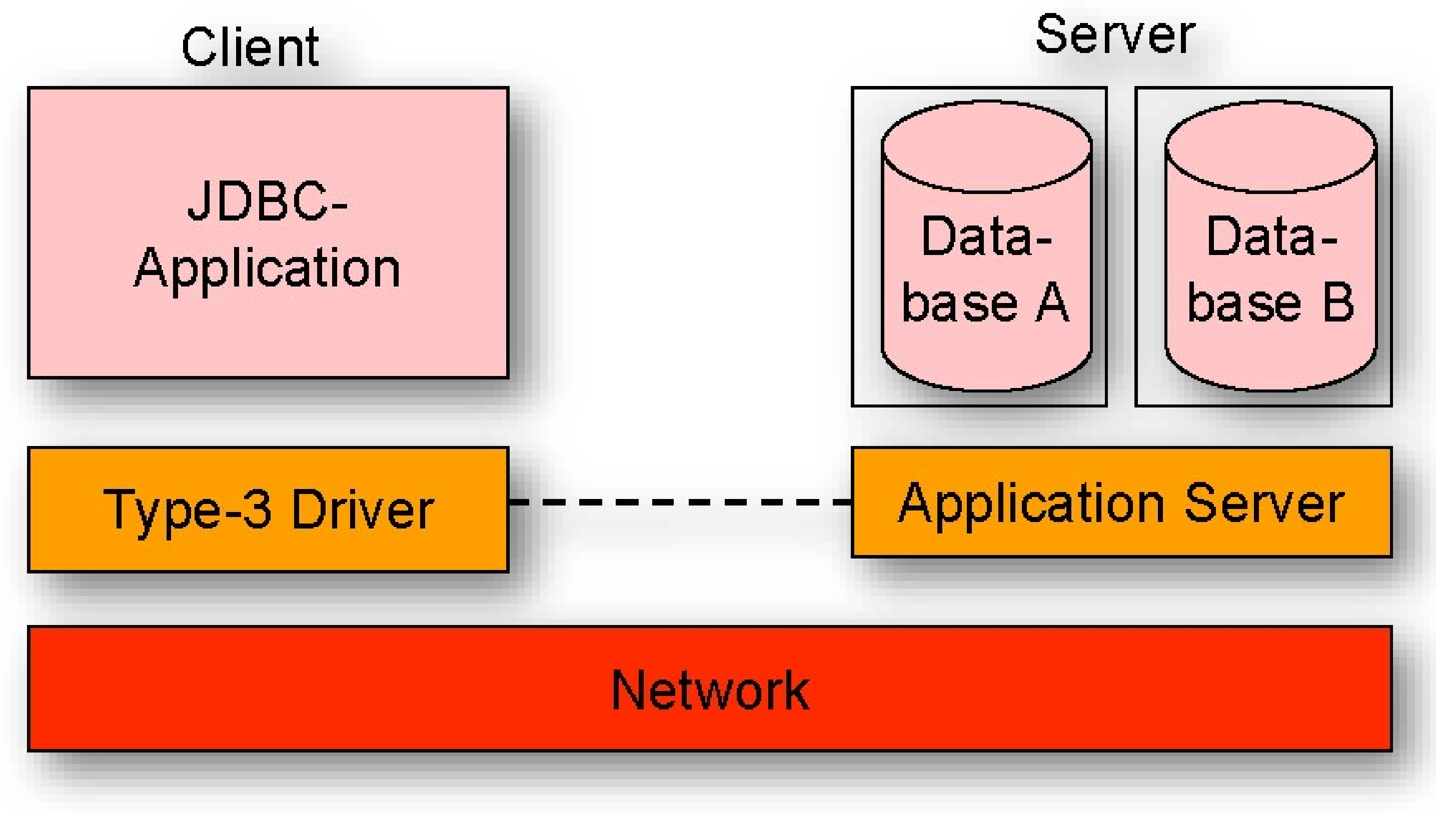
- Access to communication with the database on a programming interface provided by the database manufacturer (e.g., Oracle Call Interface, Informix-CLI)
- **Advantage:**
  - The JDBC Driver can be adapted by the database manufacturer to the specific capabilities of the database

- Disadvantages:
  - Type-2 JDBC drivers can only communicate with the database system, for which it was programmed



REF: Hans-Werner Sehring & Miguel Garcia: Arbeitsbereich Softwaresysteme (STS), TU Hamburg-Harburg

Layered System: Example /5a



REF: Hans-Werner Sehring & Miguel Garcia: Arbeitsbereich Softwaresysteme (STS), TU Hamburg-Harburg

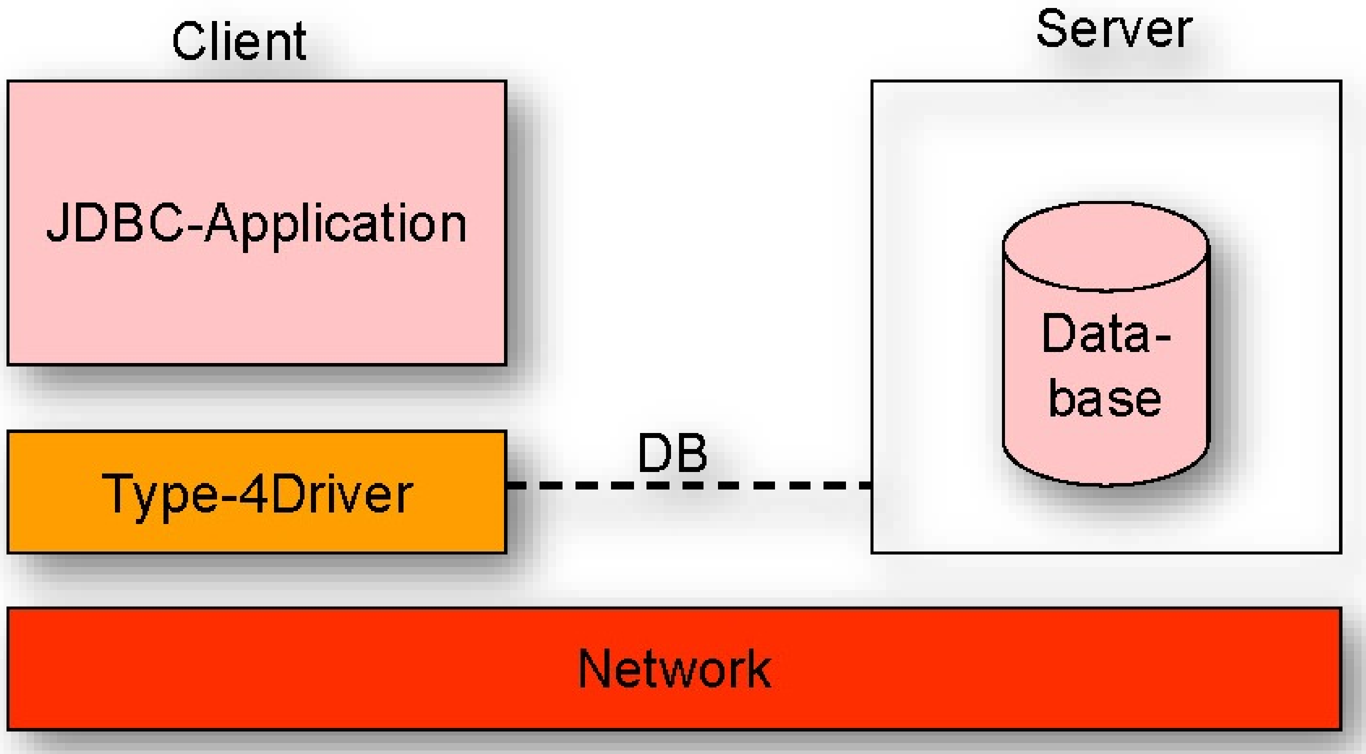
Layered System: Example /5b

Access via Type-3 Driver (net-protocol all-Java)

- The JDBC driver does not communicate directly with a database system, but with an application server over a database-independent protocol
- Application server translates messages of the client into messages that can be processed by a specific database system
- Application server can run either on the client computer, the server computer or on a third computer (Three Tier Architecture)
- **Advantage:**
  - The driver is programmed 100% in Java and therefore works with every operating system, on which Java programs can be run.
  - The driver uses a database-independent communication protocol and can therefore be used together with every database system, supported by a given application server.
- **Disadvantages:**
  - The driver works together exclusively with application server of the particular middleware manufacturer.
  - Only database systems supported by the application server can be used.

REF: Hans-Werner Sehring & Miguel Garcia: Arbeitsbereich Softwaresysteme (STS), TU Hamburg-Harburg

Layered System: Example /6a



REF: Hans-Werner Sehring & Miguel Garcia: Arbeitsbereich Softwaresysteme (STS), TU Hamburg-Harburg

Layered System: Example /6b

Access via Type-4 Driver (Native Protocol all-Java)

- The driver is 100% programmed in Java and therefore works with every operating system, on which Java programs can be run.
- The driver communicates direct with the database system over the communication protocol of the database manufacturer.
- **Advantages**:
  - OS dependent ODBC driver or CLIs not needed on the client.
  - Performance losses caused by the conversion of JDBC calls to other programming interfaces (e.g. Types 1/2) or from one communication protocol to another (e.g. Type 3) are avoided.
- **Disadvantages**
  - The driver can only communicate with the database system, for which it was developed.
  - When changing to another database system a new driver is needed.
  - Not all database manufacturers offer Type-4 drivers.

REF: Hans-Werner Sehring & Miguel Garcia: Arbeitsbereich Softwaresysteme (STS), TU Hamburg-Harburg

Example: Comparison

	Application	OS	DB
Type 0	All Java	any	DB Must support JDBC
Type 1	Java + API for ODBC	mix	DB Must support ODBC
Type 2	Java + API for supported db	mix	Supported DB only
Type 3	Java + API for Application server	any	Any DB supported by Application server
Type 4	All Java	any	Supported DB only

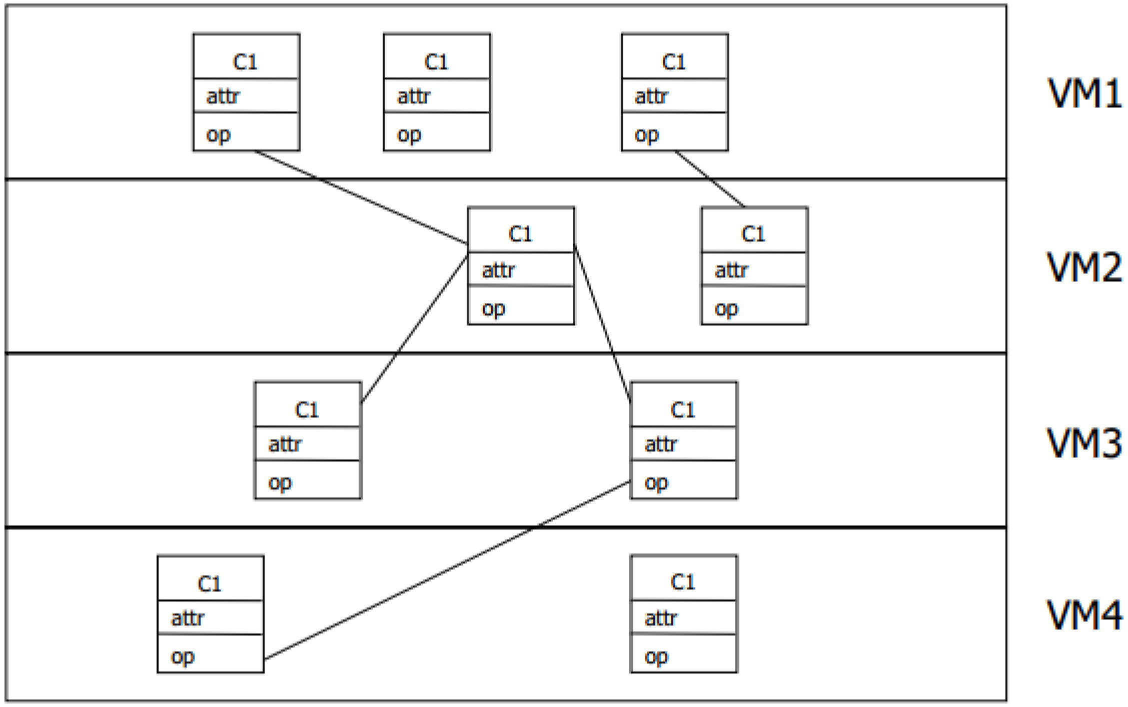
Which one you select then?
The answer depends on the trade-off among several factors and stakeholders’ concerns!

Styles of Layering

- Closed Architecture (Opaque Layering)
- Open Architecture (Transparent Layering)

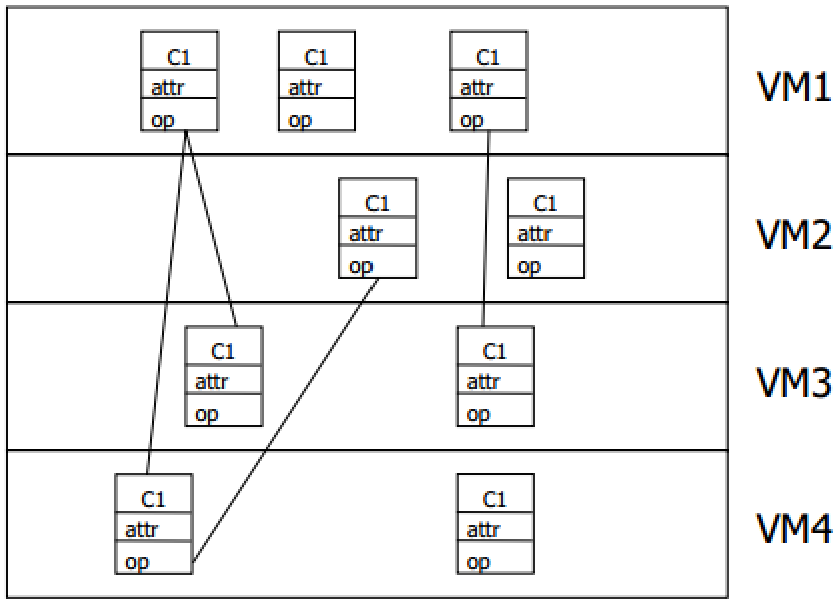
Closed Architecture (Opaque Layering)

- Any layer can only invoke operations from the immediate layer below
- Design goal: High maintainability, flexibility



Open Architecture (Transparent Layering)

- Any layer can invoke operations from any layers below
- Design goal: Runtime efficiency



Layered Systems: Summary

- Each layer provides certain facilities
  - hides part of lower layer
  - provides well-defined interfaces
- Serves various functions
  - kernels: provide core capability, often as a set of procedures
  - shells, virtual machines: support for portability
- Various scoping regimes
  - Opaque versus translucent layers

Layered Systems: Advantages

- **Better Abstraction**: Support designs based on increasing levels of abstraction → allows designer to partition a complex problem into a sequence of incremental steps
- **Better Maintenance**: each layer interacts with at most the layers below and above
- **Better Reuse**: allows multiple implementation of the same layer to be used interchangeably → possibility to standardize layer interfaces; e.g., APIs, OSI ISO model