Object Oriented Design - OOD

- The OOD phase consists of the following two sub-phases:
  - ***preliminary*** (or ***architectural***, or ***system***) ***OOD***: defines the overall strategy to build a solution that solves the problem specified at OOA time. Decisions are taken that deal with the overall organization of the software (*system architecture*)
  - ***detailed*** (or ***objects***) ***OOD***: provides the complete definition of classes and associations to be implemented, as well as the data structures and the algorithm of methods that implement class operations
- According to an iterative and incremental development approach, the OOA model is "transformed" into the OOD model, which adds the technical details of the *hardware/software solution* that defines ***how*** the software has to be implemented

System architecture

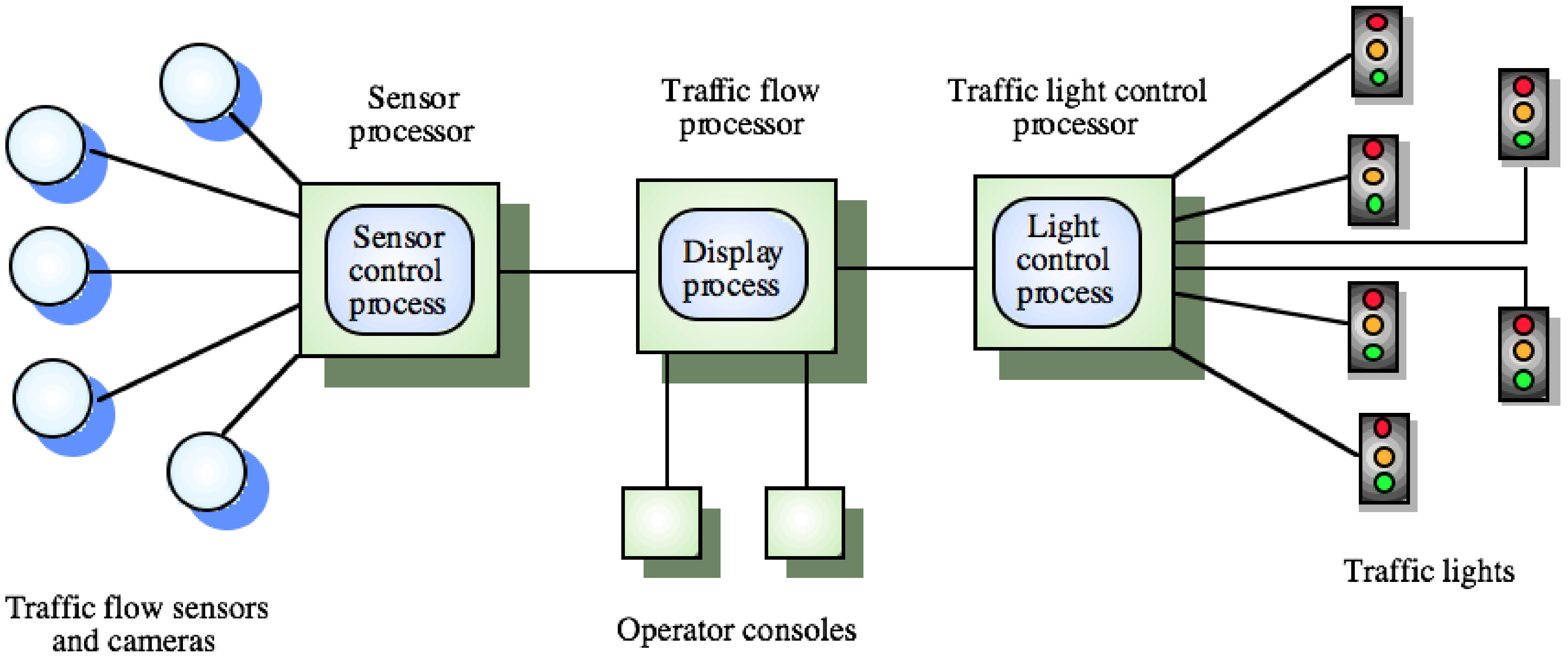
- A **system architecture** defines the structure of the software system components along with the relationships between such components and the principles driving design and system evolution
- Evolution of system architectures:
  1. Mainframe-based architectures
  2. File sharing architectures
  3. Client/server (C/S) architectures:
     - 3.1 two-tier (*thin client, fat client*)
     - 3.2 three-tier (*upper layer, middle layer, bottom layer*)
  4. Distributed objects architectures
  5. Component-based architectures
  6. Service-oriented architectures
- Architectures from 3 through 6 are denoted as distributed architectures, or architectures of ***distributed software systems***

Distributed software systems

- The processing of a **distributed software system** is distributed over a set of independent execution hosts, which are connected by a network infrastructure (of either *local area* or *wide area* type)
- The set of independent execution hosts is seen by users as a single execution host
- Middleware technology has played an essential role in the transition from centralized architectures to distributed ones
- **Middleware** refers to the software layer that provides connectivity
- Such layer is in between the *application* and *operating system* layers and provides a set of services to establish the required interaction among the various application processes executed by networked hosts (e.g., *TP monitor*, *RPC*, *MOM*, *ORB*)

Example distributed software system

Traffic control system: composed of multiple processes which are executed onto different processors (could be executed onto a single processor as well – it's the set of separate processes that makes distributed a software system)



Main features of distributed systems

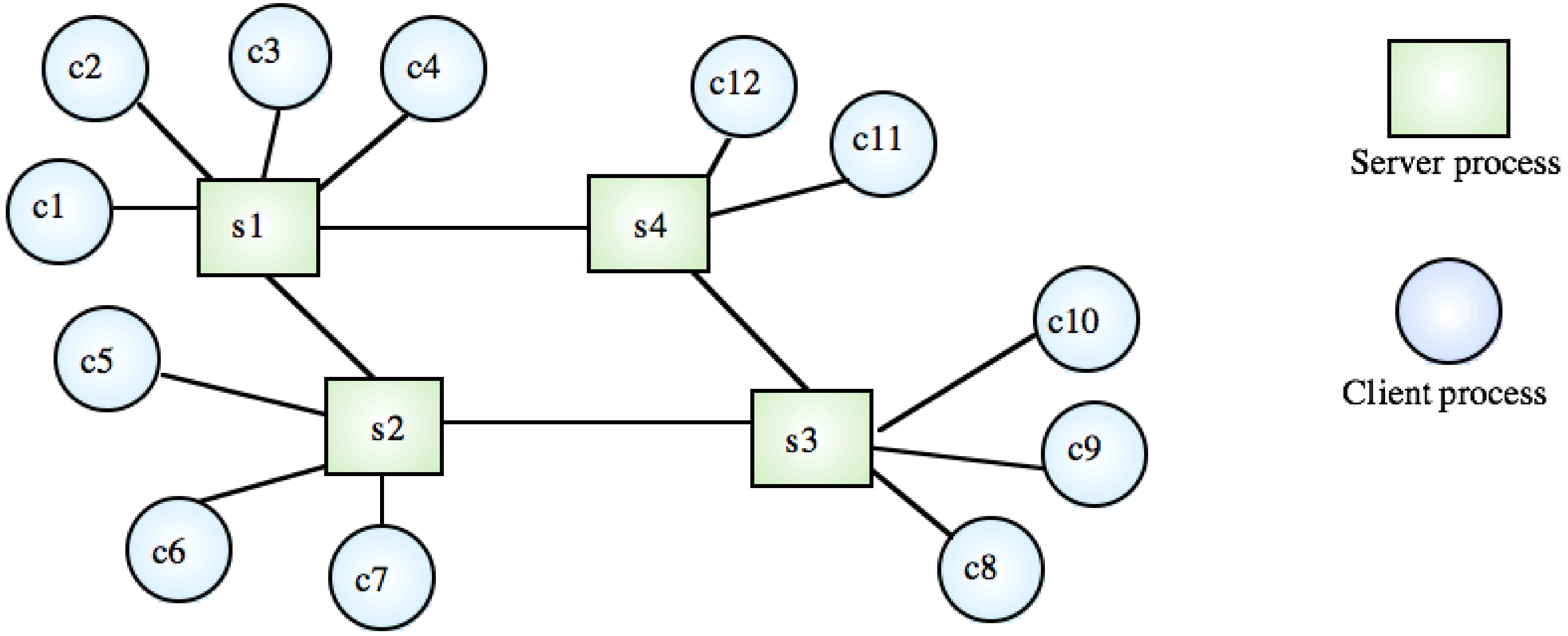
- Data and resource sharing
- Openness (ability to manage heterogeneous resources)
- Concurrency
- Scalability
- Load balancing
- Fault tolerance
- Transparency
- Adaptability to *enterprise computing* scenarios
- Critical factors
  - quality of service (performance, reliability, etc.)
  - interoperability
  - security

Client/server (C/S) architectures

- Each process plays the role of *client* or *server*
- The **client** process interacts with the user as follows:
  - provides the user interface to collect user requests
  - forwards requests to servers, by use of middleware technology
  - displays server responses back to the user through the user interface
- The **server** process (or the set of processes executed by a given host) provides services to the clients, as follows:
  - replies to client requests (it's not the server that initiates the conversation with the client)
  - hides the complexity of the entire C/S system to the user (a given server may in turn act as a client that forwards the initial request to a secondary server, without making the client and the user aware of the forwarding chain)

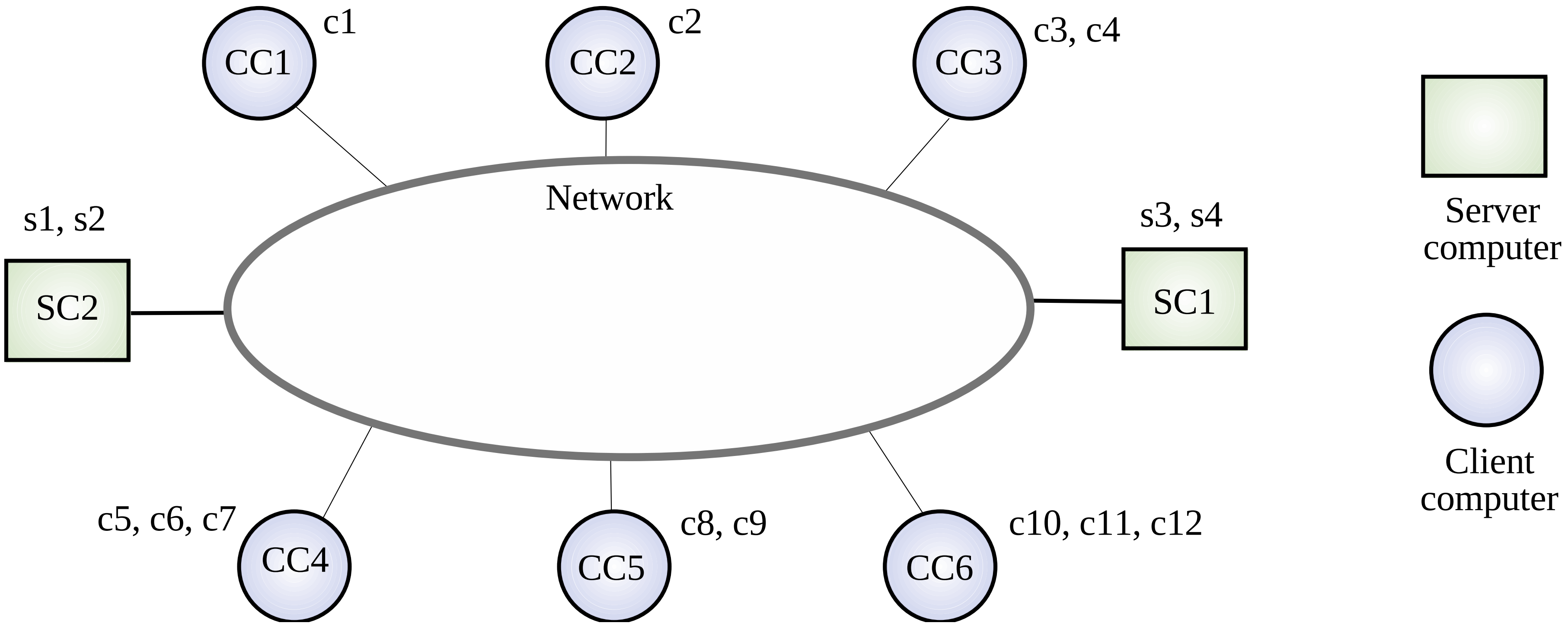
Client/server (C/S) architectures (2)

A **C/S architecture** partitions software applications in terms of a set of separate processes, each acting as a *client*, a *server* or *both*



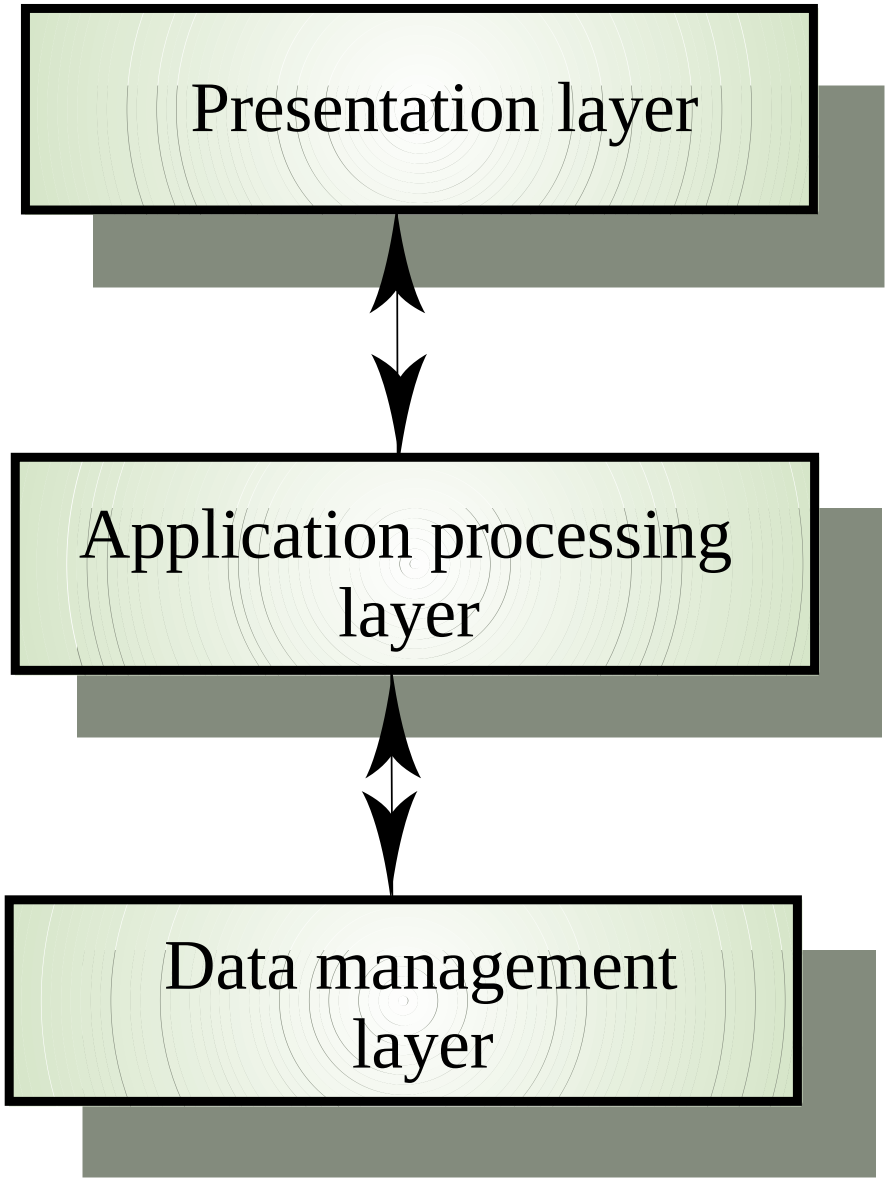
Client/server (C/S) architectures (3)

A **C/S architecture** partitions software applications in terms of a set of separate *processes* that are executed onto the same host (the *processor*) or on a group of networked hosts



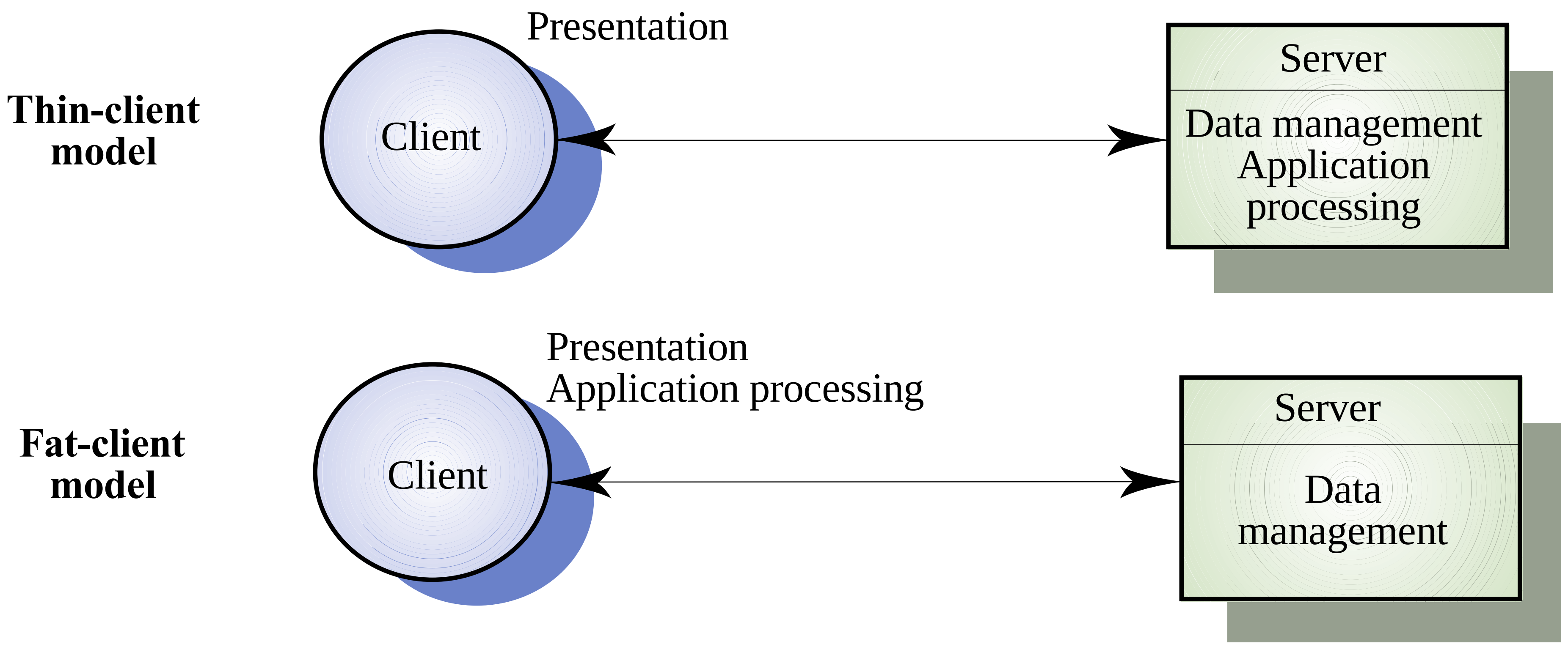
Application layers

- *Presentation* layer
  concerned with collecting user inputs and presenting the results of a computation to system users
- *Application processing* layer
  concerned with providing application specific functionality, e.g., in a banking system, banking functions such as open account, close account, etc.
- *Data management* layer
  concerned with managing access to application data



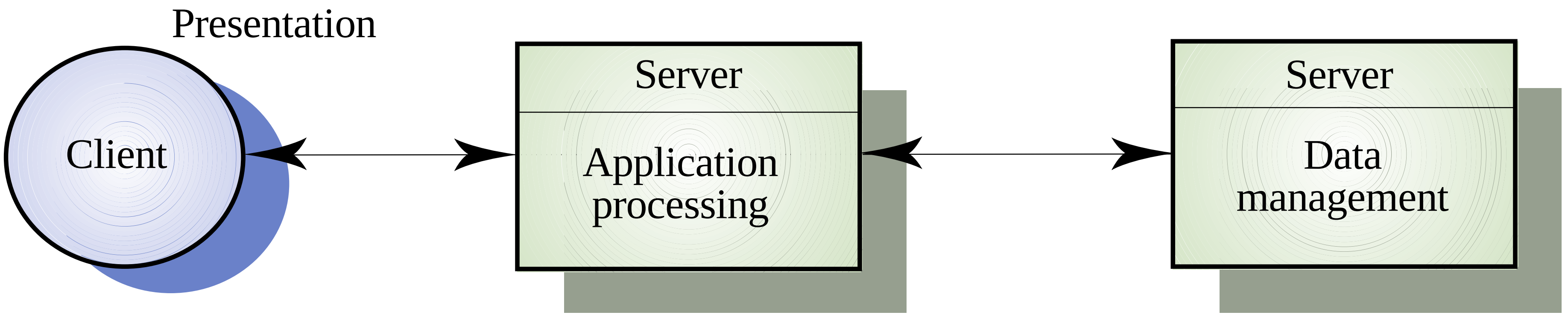
Two-tier C/S architectures

- **thin-client** model: all of the application processing and data management is carried out on the server; the client is simply responsible for running the *presentation* software
- **fat-client** model: the server is only responsible for data management; the software on the client implements the *application logic and* the *presentation* software.



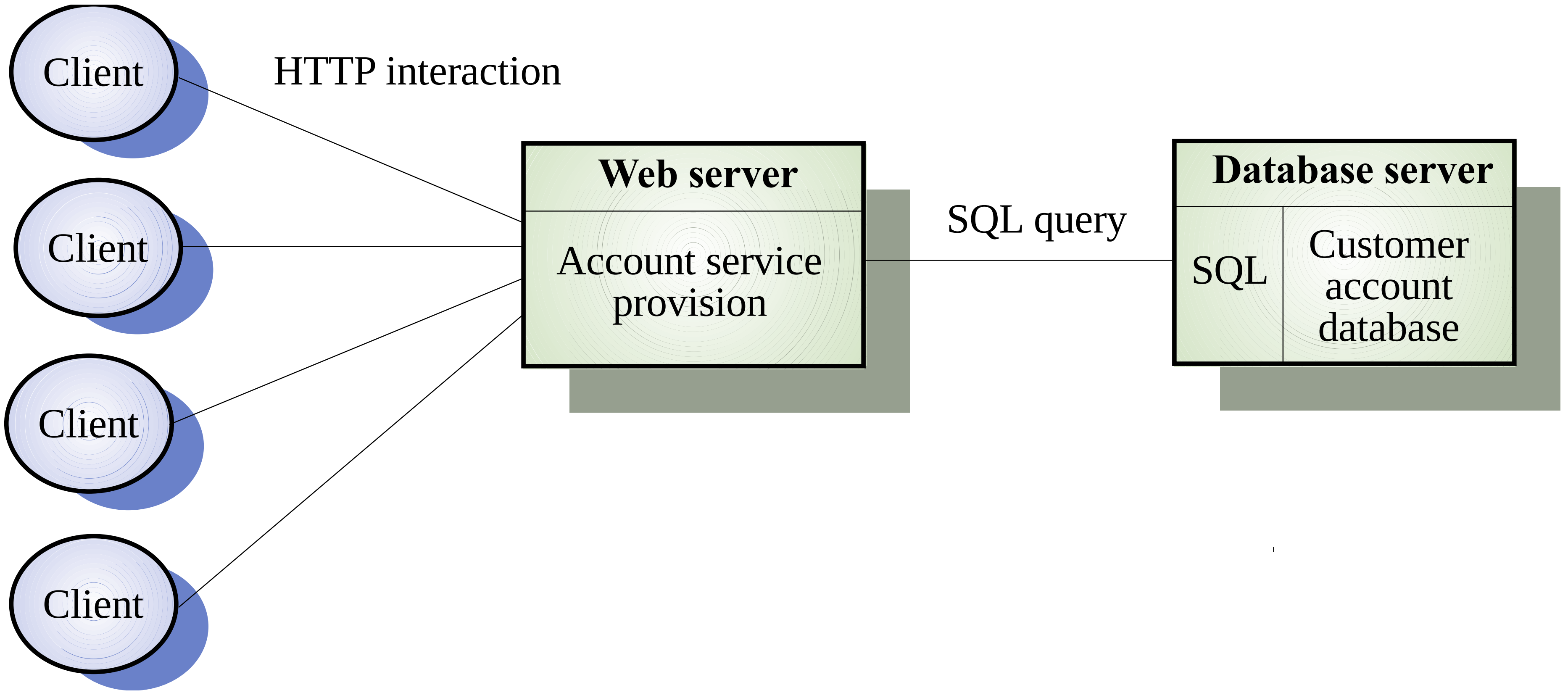
Three-tier C/S architectures

- Each of the application architecture layers executes on a *separate processor*
- Allows for *better performance* than a two-tier thin-client approach and is *simpler to manage* than a two-tier fat-client approach
- A *more scalable* architecture
  – as demands increase, extra servers can be added



Example C/S *three-tier* architecture

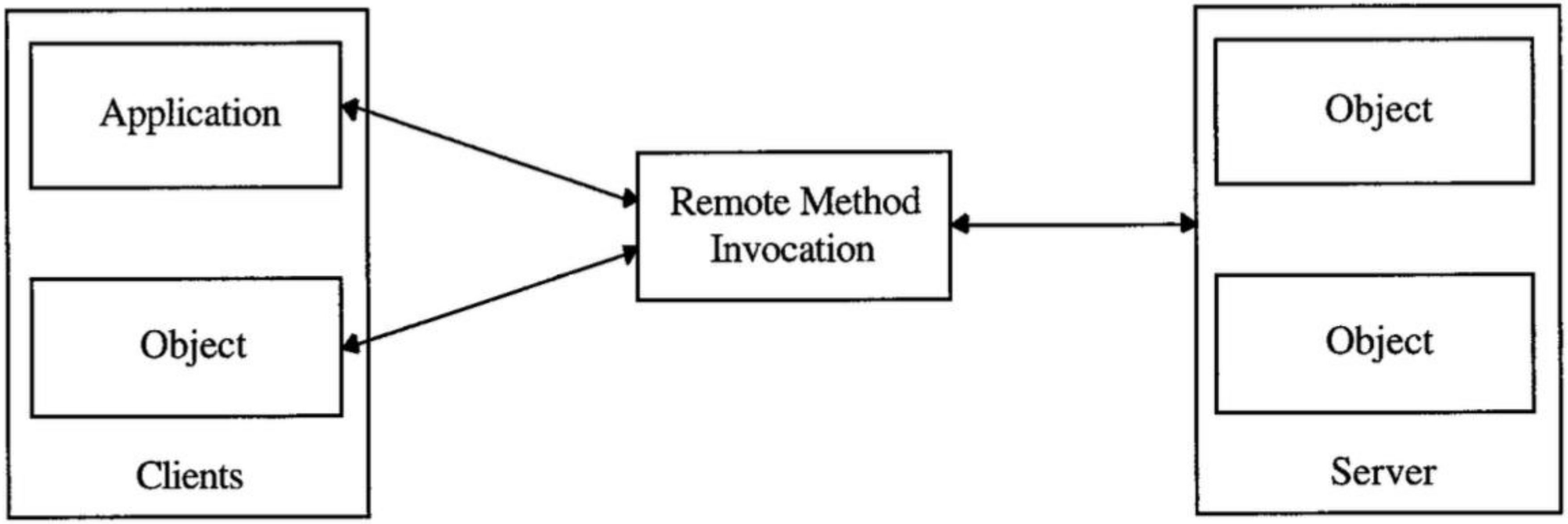
Internet Banking System



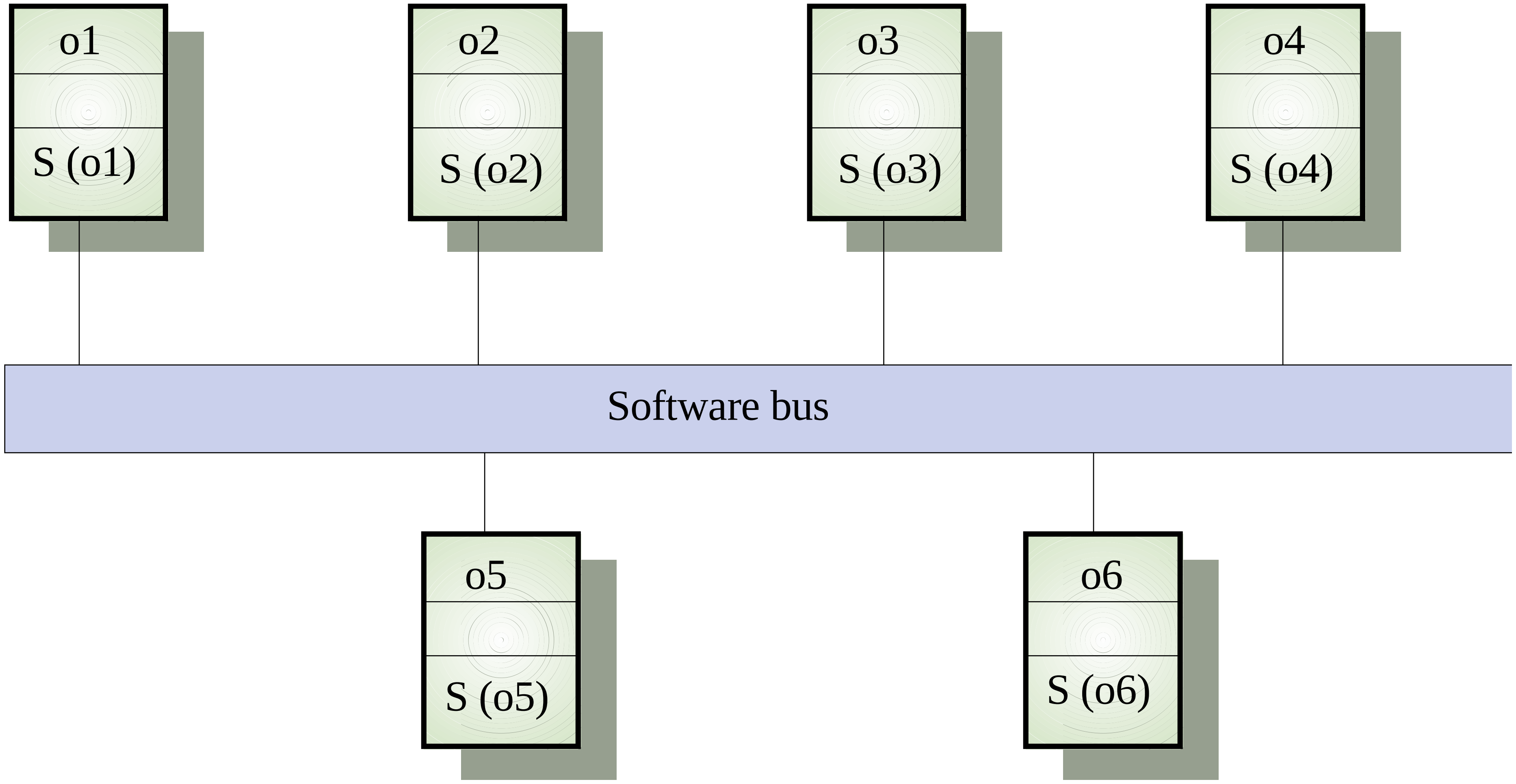
Distributed object architectures

- No distinction between client and server
- Each distributed object acts *both* as a *client* (by sending request messages, i.e., by invoking methods) *and* as a *server* (by providing response messages, i.e., by executing the invoked method)
- The remote communication between objects is made transparent by use of *middleware* based on the *software bus* concept (referred to as *object request broker*):
  - ***abstract bus****:* specification of the interface providing communication and data exchange services (control transfer model and type model for exchanged values)
  - ***bus implementation****:* implementation of the abstract bus for a given HW/SW platform ($\Rightarrow$ *separation between interface and implementation*)
- Applications based on distributed object architectures consists of a set of objects that are executed onto distributed and heterogeneous platforms and that communicates through remote method invocation

Distributed object architectures (2)



Example of *distributed object architecture*



Component-*based* architectures

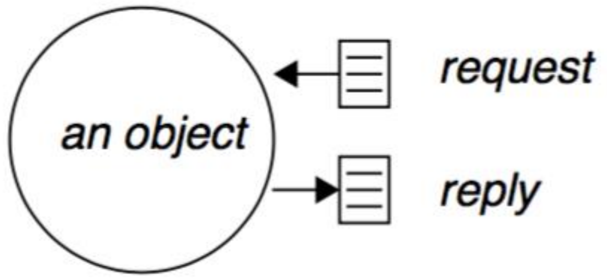
- ***Component-based architectures*** define software products assembled from a set of *software components*, which are designed to work together as part of a *component framework*
- A *component framework* makes use of generic software architectures to formalize given classes of applications
- *Component-based software systems* support the efficient development of software systems whose requirements exhibit significant levels of variability
- It is thus necessary to identify and implement *software abstractions* that encapsulate efficient and reliable solutions to standard coordination and synthesis problems
- These abstractions, or “**components**”, are used to build bigger systems, while hiding the implementation details of the smaller structure
- A components can be used across many different applications, and can be reconfigured when application requirements

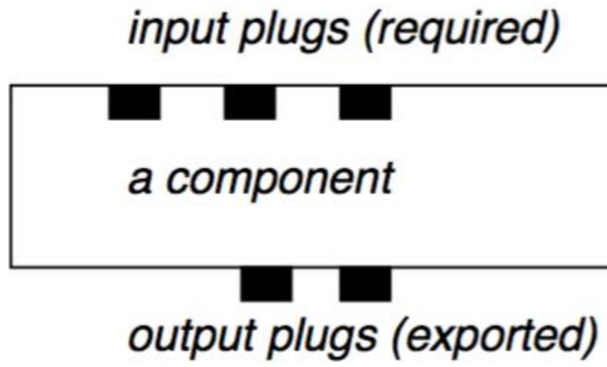
Component-based architectures (2)

- An essential element for building component-based software systems is the ***black-box reuse*** of software
- Putting together components is simple, since each component has a limited set of "***plugs***" with fixed rules specifying how it may be linked with other components
- Instead of having to adapt the structure of a piece of software to modify its functionality, a user plugs the desired behavior into the parameters of the component
- Important aspects of components:
  - encapsulation of software structures as abstract components (*variability*)
  - composition of components by binding their parameters to specific values, or other components (*adaptability*)

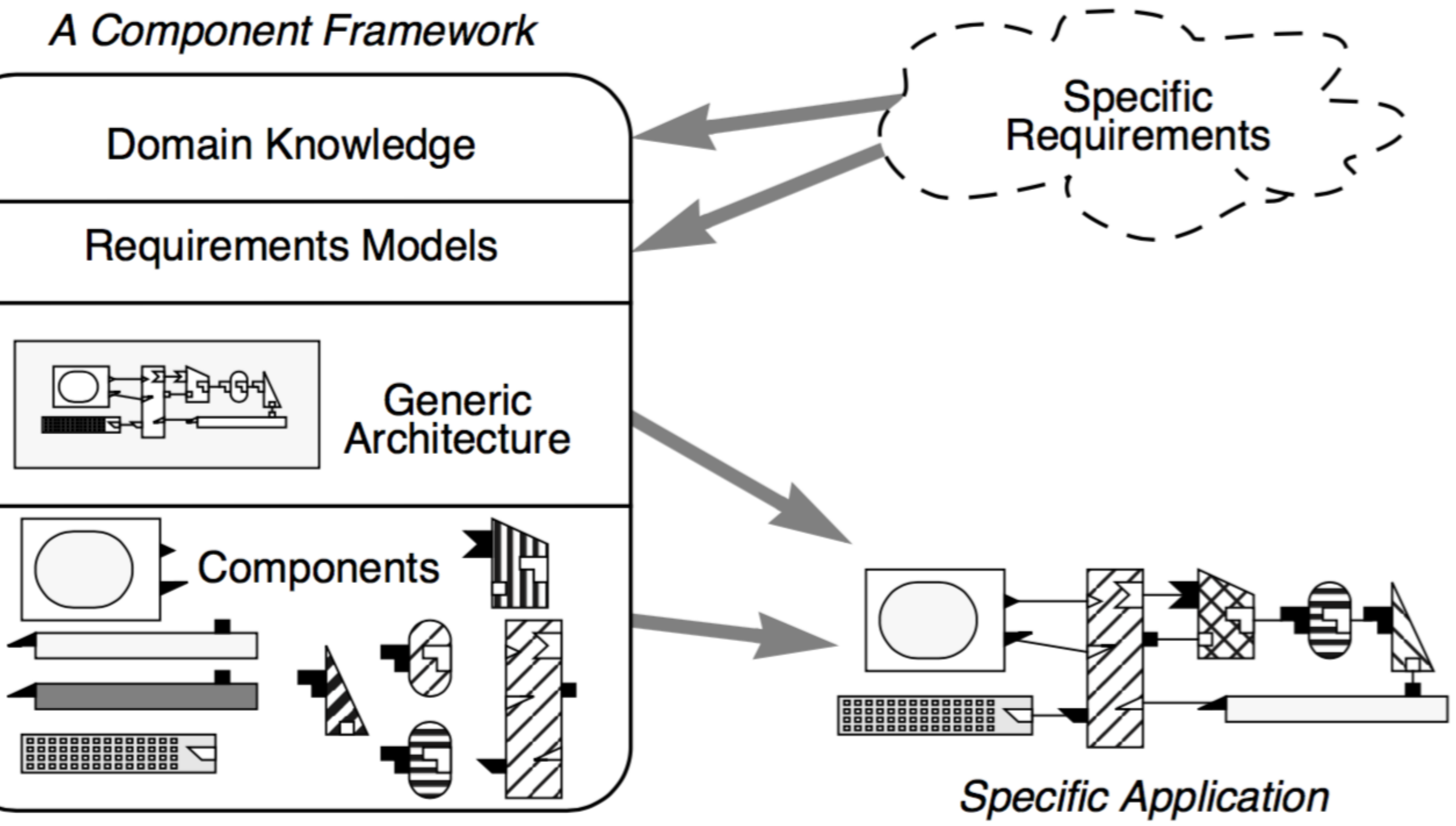
Component-*based* architectures (3)

- *Objects* vs. *components*:
  - objects encapsulate services, whereas components are abstractions (that can be used to construct object-oriented systems)
  - objects have identity, state and behavior, and are always *run-time* entities; components, on the other hand, are generally static entities that are needed at system *build-time* (and do not necessarily exist at run-time).
  - components may be of finer or coarser *granularity* than objects: e.g., classes, templates, mix-ins, modules; components should have an explicit *composition interface*, which is type-checkable

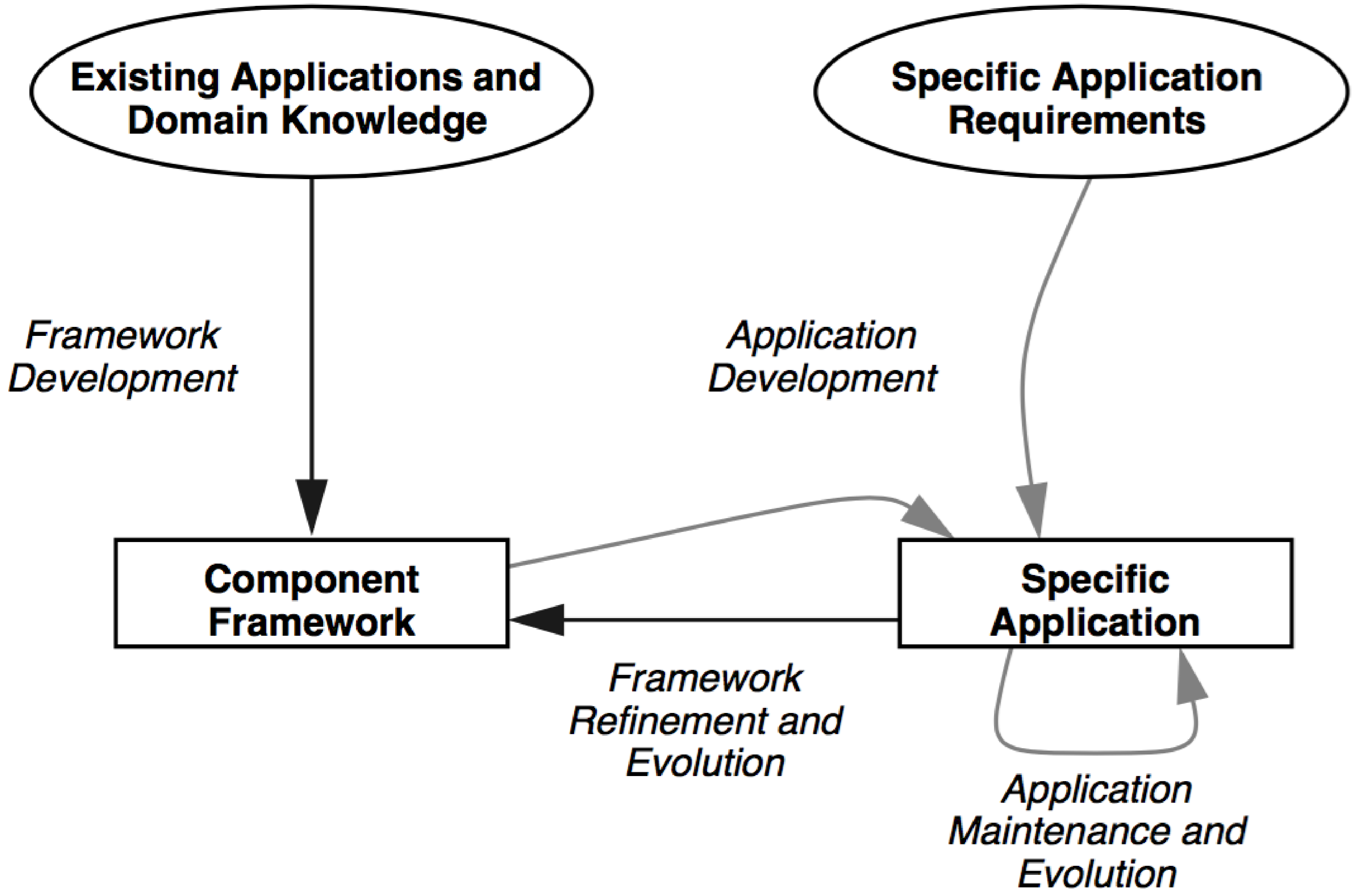




Component framework

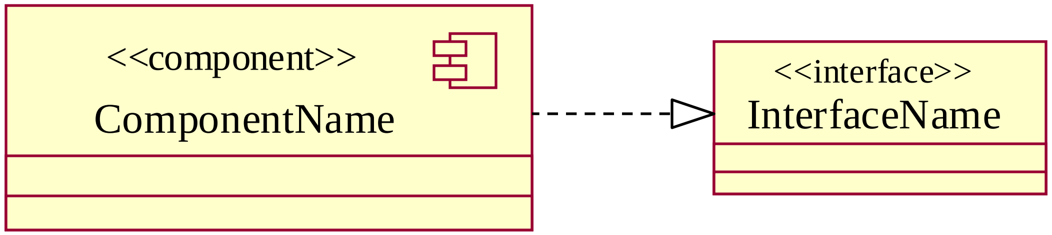


Framework dev vs. *Application* dev


Existing Applications and Domain Knowledge
Specific Application Requirements
Framework Development
Application Development
Component Framework
Specific Application
Framework Refinement and Evolution
Application Maintenance and Evolution

UML Components

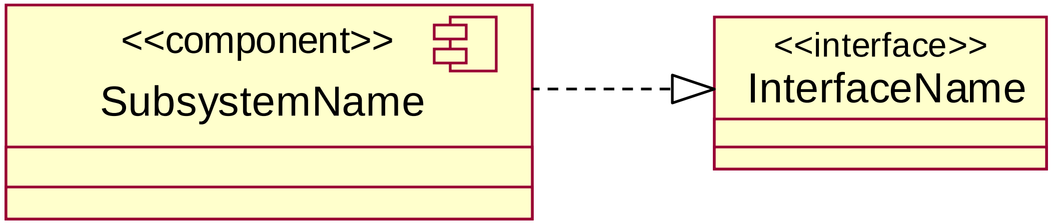
- A modular part of a system that hides its contents and whose appearance is replaceable within its environment
  - Defines its behavior in terms of provided and required interfaces that can be wired together
  - Can be replaced at design time or run-time by a component that offers equivalent functionality based on compatibility of its interfaces.



OO Principles: Encapsulation and Modularity

What is a Subsystem?

- A part of a system that encapsulates behavior, exposes a set of interfaces, and contains other model elements.
  - Modeled as a component



OO Principles: Encapsulation and Modularity

Service Oriented Architecture (SOA)

- A SOA is a distributed software architecture that consists of multiple autonomous *services*
- The services are distributed such that they can execute on different nodes with different service providers
- With a SOA, the goal is to develop software *applications that are composed of distributed services*, such that individual services can execute on different platforms and be implemented in different languages

SOA protocols

- Standard *Internet-based protocols* are provided to allow services to communicate with each other and to exchange information
- Each service has a *service description*, which allows applications to discover and communicate with the service
- The service description defines the name of the service, the location of the service, and its data exchange requirements

Service providers and consumers

- A service *provider* supports services used by multiple clients
- Unlike client/server architectures, SOAs build on the concept of *loosely coupled services* that can be discovered and linked to by *clients* (also referred to as service *consumers* or service *requesters*) with the assistance of service *brokers*

SOA design concepts

- An important goal of SOA is to design services as *autonomous reusable components*
- Services are intended to be self-contained and loosely coupled, meaning that dependencies between services are kept to a minimum
- Instead of one service depending on another, *coordination services* are provided in situations in which multiple services need to be accessed and access to them needs to be sequenced
- Several software architectural patterns are described for service-oriented applications:
  - *Broker* patterns, including Service Registration, Service Brokering, and Service Discovery
  - *Transaction* patterns, including Two-Phase Commit, Compound, and Long-Living Transaction patterns
  - and *Negotiation* patterns

Services Design Principles

- Loose coupling
- Service contract
- Autonomy
- Abstraction
- Reusability
- Composability
- Statelessness
- Discoverability

Software Architectural Broker Patterns

- In a SOA, *object brokers* act as intermediaries between clients and services
- In the **Broker** pattern (which is also known as the *Object Broker* or *Object Request Broker* pattern), the **broker** acts as an intermediary between the clients and services
- Services register with the broker
- Clients locate services through the broker
- After the broker has brokered the connection between client and service, communication between client and service can be direct or via the broker

Transparency

- The broker provides both location transparency and platform transparency
- **Location transparency** means that if the service is moved to a different location, clients are unaware of the move and only the broker needs to be notified
- **Platform transparency** means that each service can execute on a different hardware/software platform and does not need to maintain information about the platforms that other services execute on

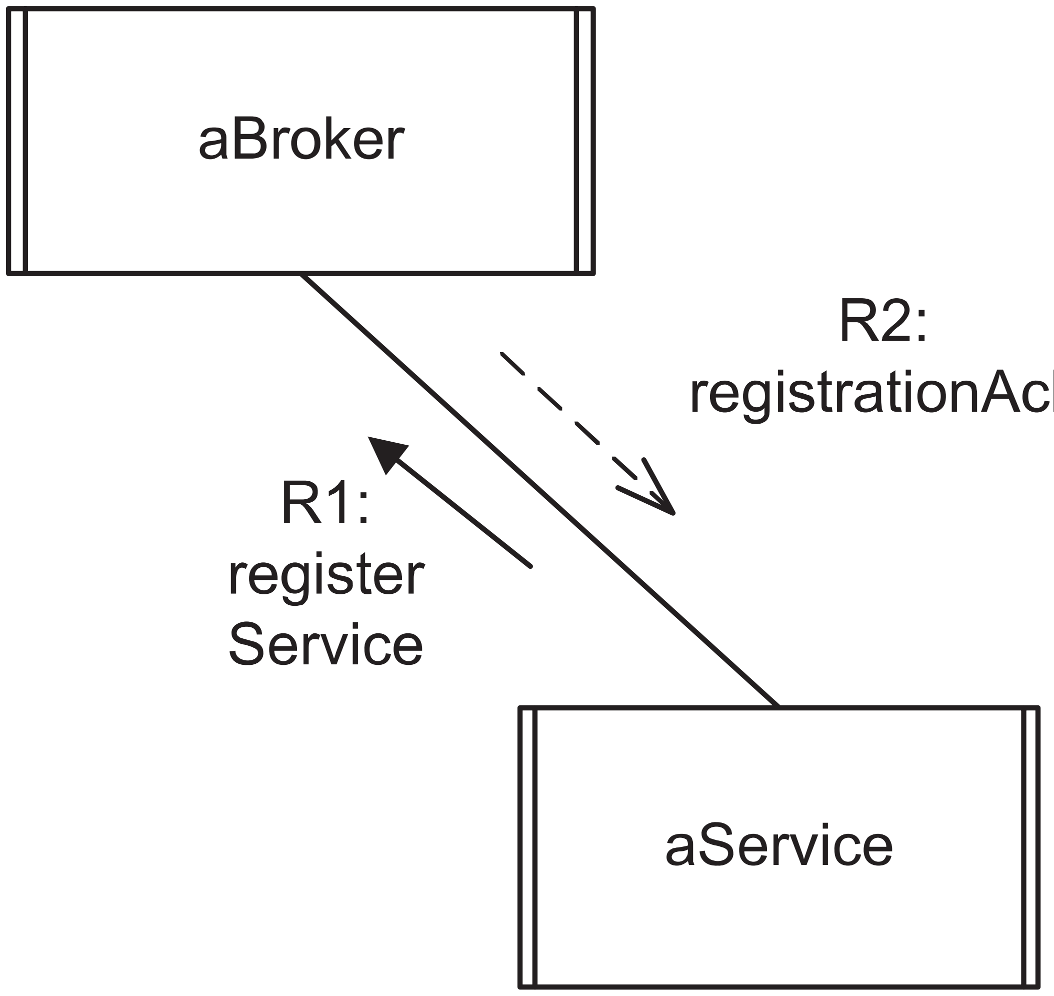
Brokered communication

- With brokered communication, instead of a client having to know the location of a given service, the client queries the broker for services provided
- First, the service must register with a broker as described by the *Service Registration pattern*

Service Registration Pattern

- The service needs to register service information with the broker, including the service name, a description of the service, and the location at which the service is provided
- Message sequence:
  1. the service sends a register service request to the broker
  2. the broker registers the service in the service registry and sends a registration acknowledgment to the service.

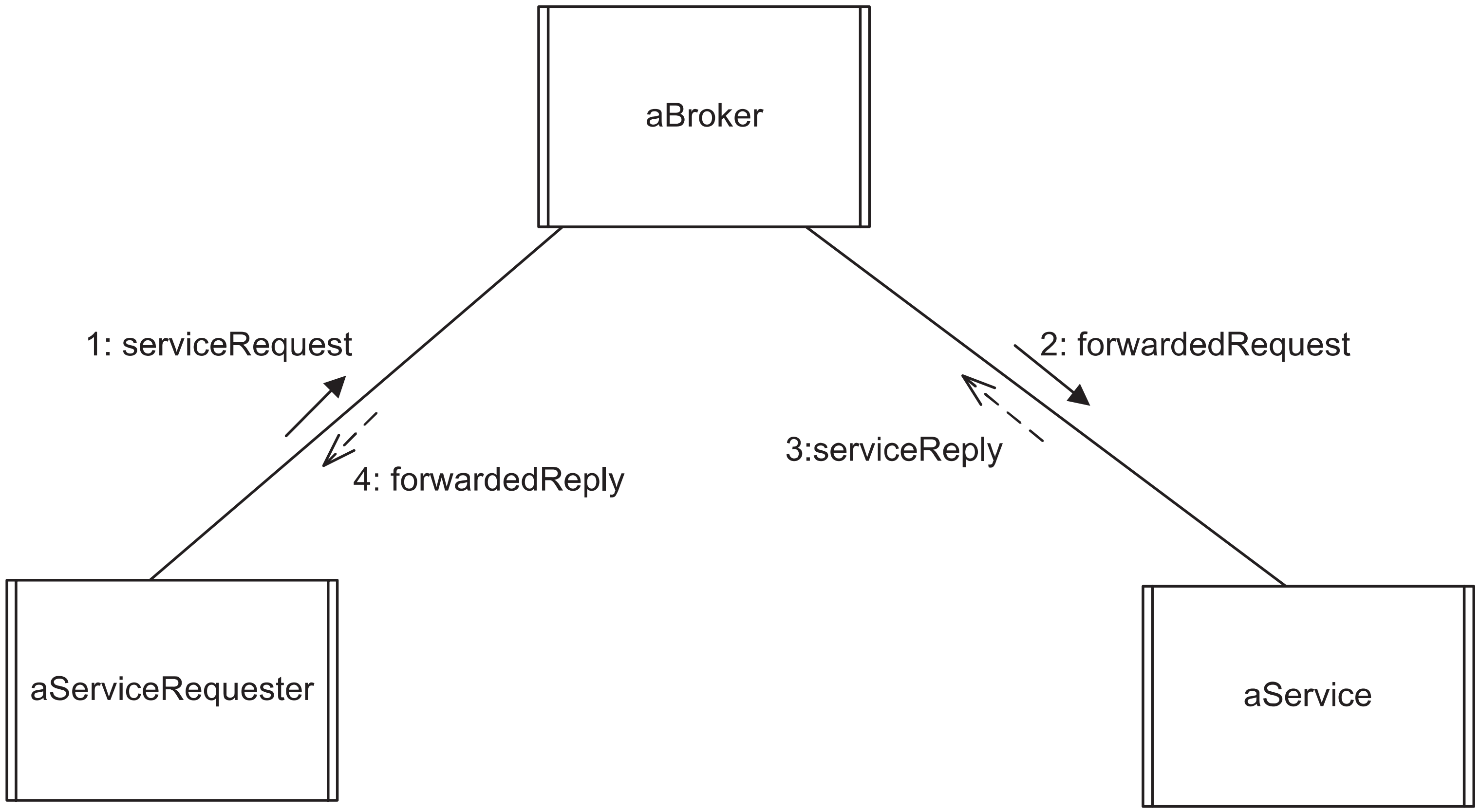
Service Registration Pattern



Broker Fowarding Pattern (white pages)

1. A client sends a message identifying the service required – for example, to withdraw cash from a given bank
2. The broker receives the client request, determines the location of the service (the ID of the node the service resides on), and forwards the message to the service at the specific location
3. The message arrives at the service, and the requested service is invoked
4. The broker receives the service response and forwards it back to the client

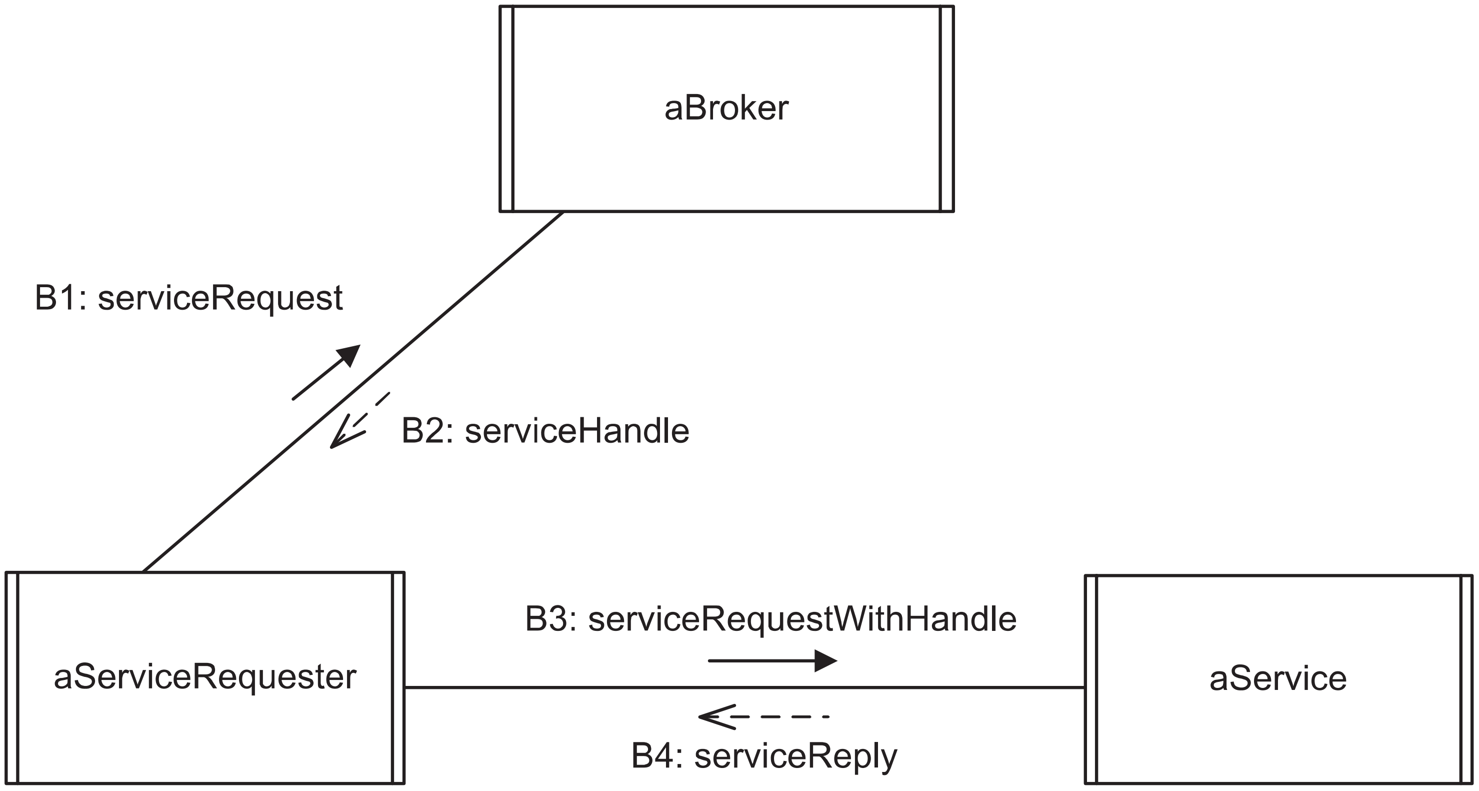
Broker Fowarding Pattern (white pages)



Broker Handle Pattern (white pages)

- The **Broker Handle** pattern keeps the benefit of location transparency while adding the advantage of reducing message traffic
- Instead of forwarding each client message to the service, the broker returns a service handle to the client, which is then used for direct communication between client and service
- This pattern is particularly useful when the client and service are likely to have a dialog and exchange several messages between them.

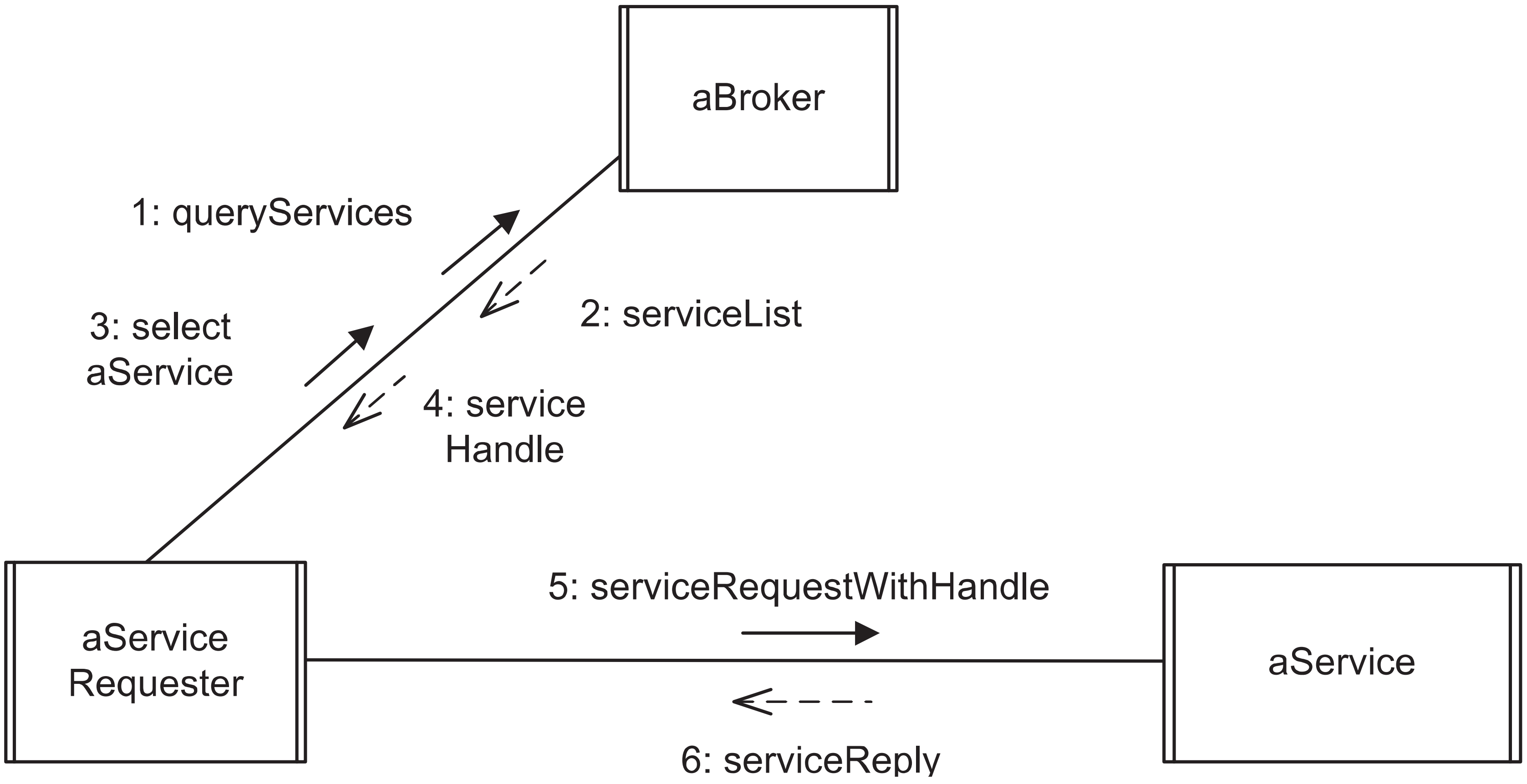
Broker Handle Pattern (white pages)



Service Discovery Pattern (yellow pages)

- In *white pages brokering* the client knows the service required but not the location
- A different brokering pattern is *yellow pages brokering***,** analogous to the yellow pages of the telephone directory, in which the client knows the type of service required but not the specific service
- Also known as the **Service Discovery** pattern because it allows the client to discover new services:
    1. The client sends a query request to the broker, requesting all services of a given type
    2. The broker responds with a list of all services that match the client's request
    3. The client, possibly after consultation with the user, selects a specific service
    4. The broker returns the service handle, which the client uses for communicating directly with the service

Service Discovery Pattern (yellow pages)



Technology Support for SOA

- Although SOAs are conceptually platform-independent, they are currently provided very successfully on **Web Services** technology platforms
- A *web service* is a service that is accessed using standard Internet and XML-based protocols

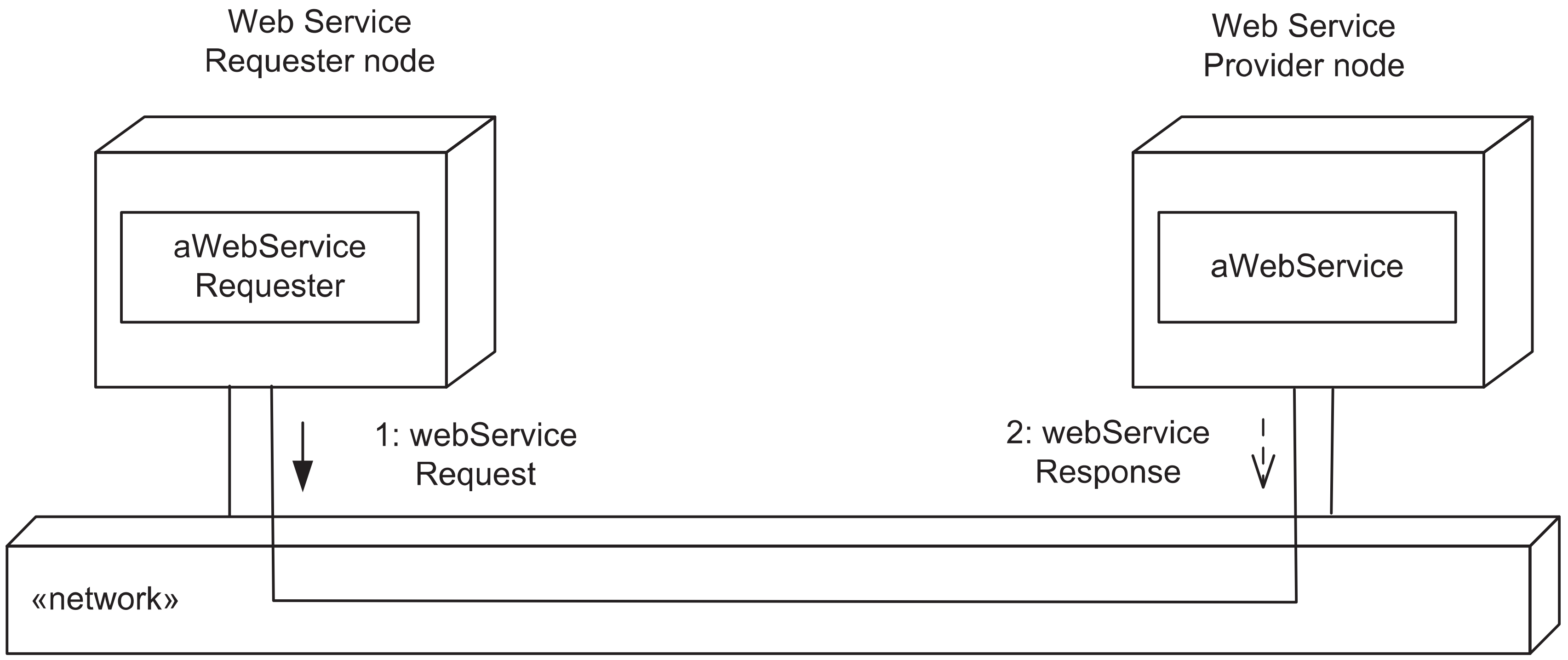
Web Service Protocols

- Application clients and services need to have a communication protocol for inter-component communication
- Extensible Markup Language (XML) is a technology that allows different systems to interoperate through exchange of data and text
- The **Simple Object Access Protocol (SOAP)**, which is a lightweight protocol developed by the World Wide Web Consortium (W3C), builds on XML and HTTP to permit exchange of information in a distributed environment
- SOAP defines a unified approach for sending XML-encoded data and consists of three parts:
  - an envelope that defines a framework for describing what is in a message and how to process it
  - a set of encoding rules for expressing instances of application-defined data types, and
  - a convention for representing remote procedure calls and responses

Web Services

- Applications provide services for clients
- One example of application services is **Web services,** which use the World Wide Web for application-to-application communication
- From a software perspective, Web services are the application programming interfaces (APIs) that provide a standard means of communication among different software applications on the World Wide Web
- From a business application perspective, a Web service is business functionality provided by a company in the form of an explicit service over the Internet for other companies or programs to use
- A Web service is provided by a service provider and may be composed of other services to form new services and applications.

Web Service example



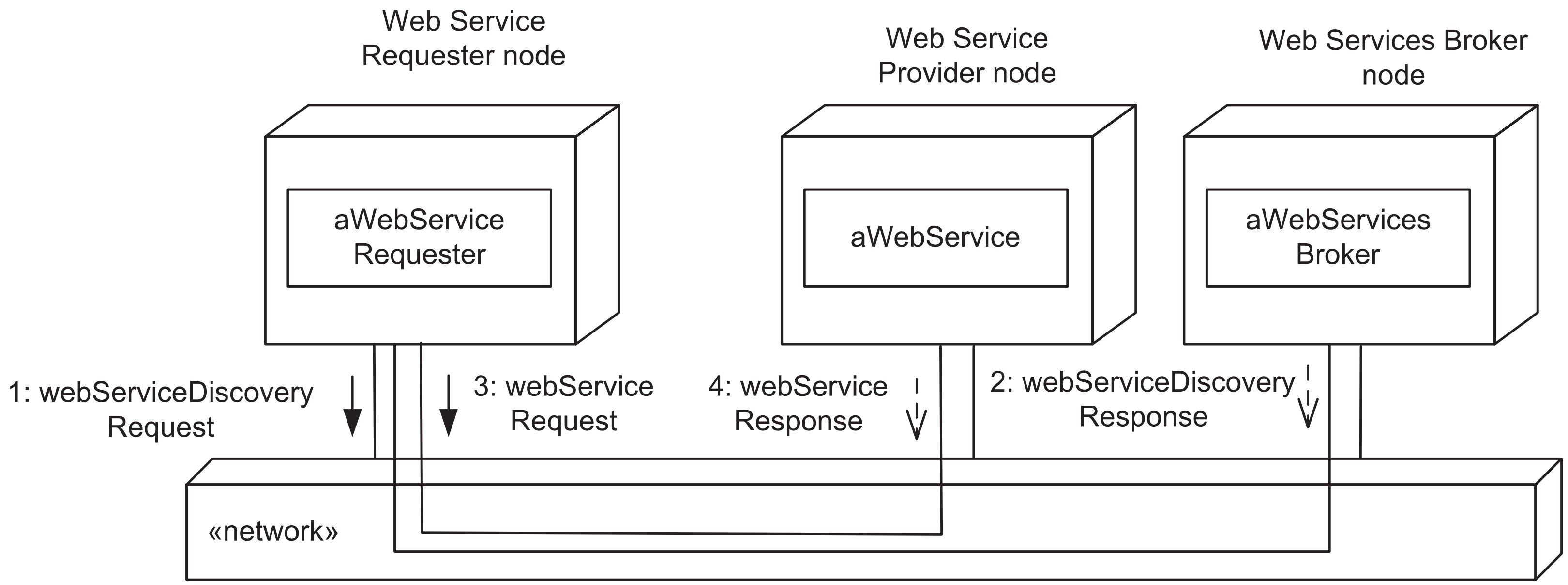
Registration Services

- A registration service is provided for services to make their services available to clients
- Services register their services with a registration service – a process referred to as *publishing* or *registering* the service
- Most brokers, such as CORBA and Web service brokers, provide a registration service
- For Web services, a **service registry** is provided to allow services to be published and located via the World Wide Web.
- Service providers register their services together with service descriptions in a service registry
- Clients searching for a service can look up the service registry to find a suitable service
- The **Web Services Description Language** (**WSDL**) is an XML-based language used to describe what a service does, where it resides, and how to invoke it

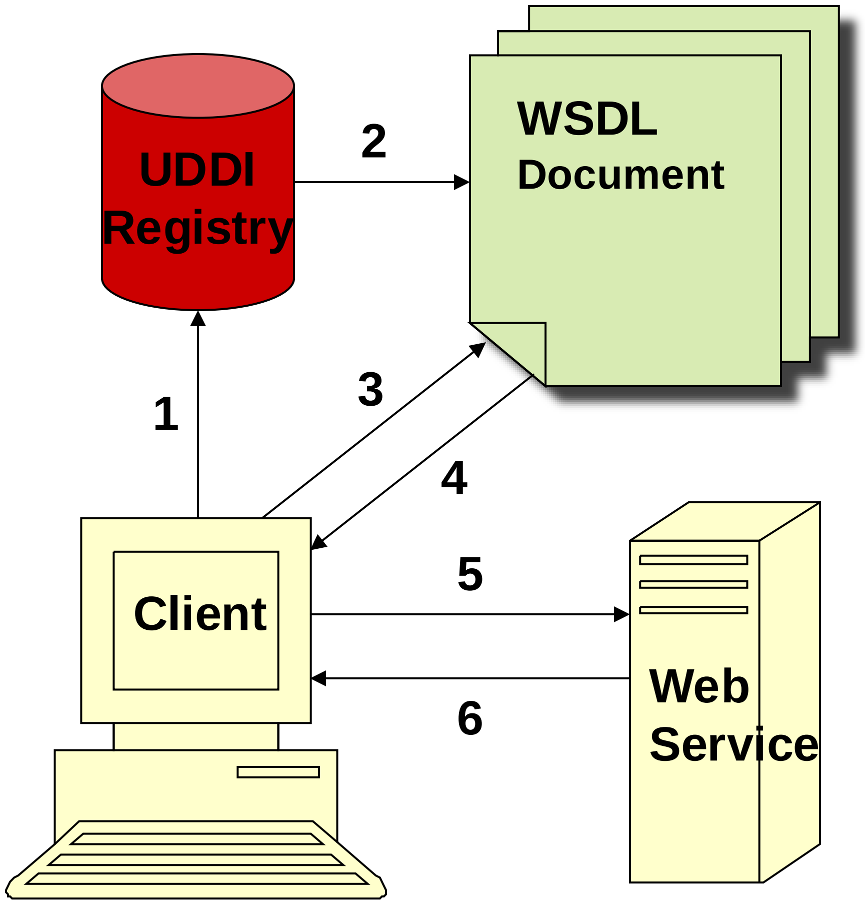
Brokering and Discovery Services

- In a distributed environment, an **object broker** is an intermediary in interactions between clients and services
- An example of brokering technology is a Web services broker
- Information about a Web service can be defined by the **Universal Description, Discovery, and Integration (UDDI)** framework for Web services integration
- A UDDI specification consists of several related documents and an XML schema that defines a SOAP-based protocol for registering and discovering Web services
- A Web services broker can use the UDDI framework to provide a mechanism for clients to dynamically find services on the Web

Web Service Broker Example

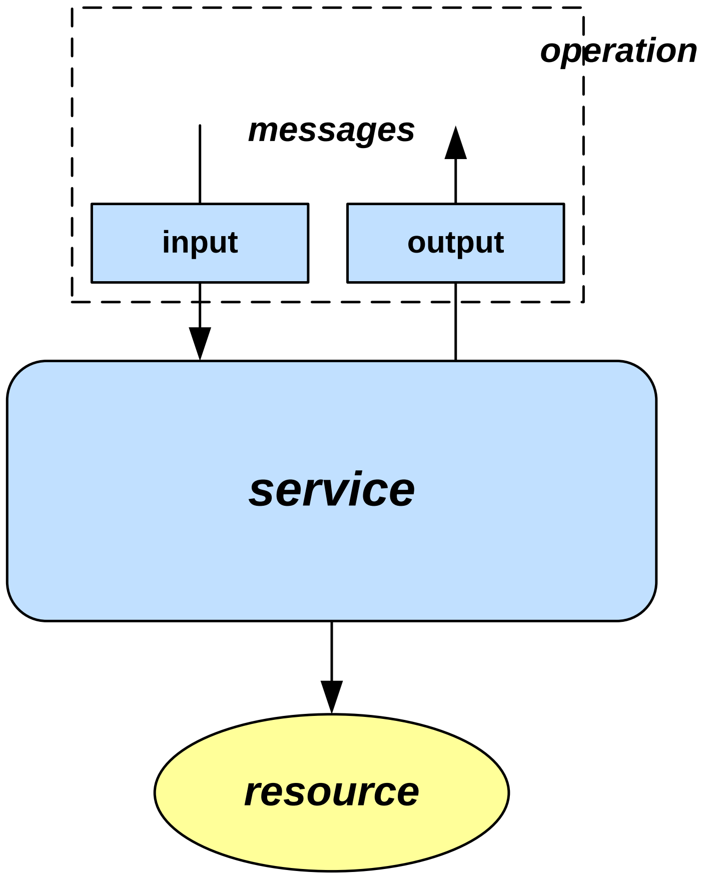


Web Service Protocols and Standards

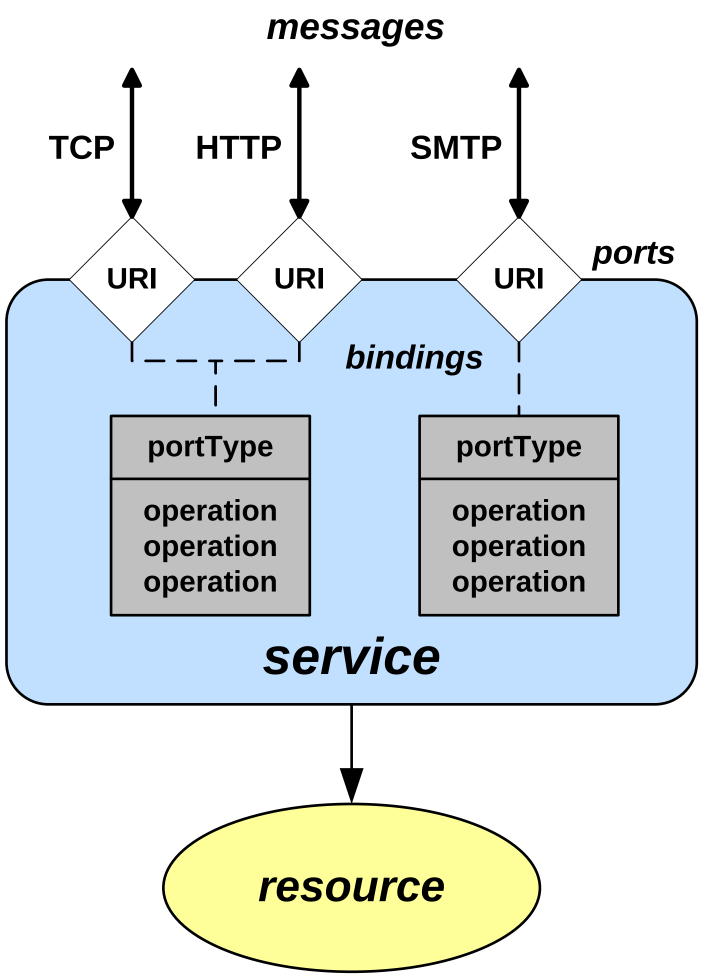


1. Client queries UDDI registry to locate service
2. Registry refers client to WSDL document
3. Client accesses WSDL document
4. WSDL provides data to interact with web service
5. Client sends SOAP-message request
6. Web service returns SOAP-message response

WSDL



Operations



Ports and Bindings

REST

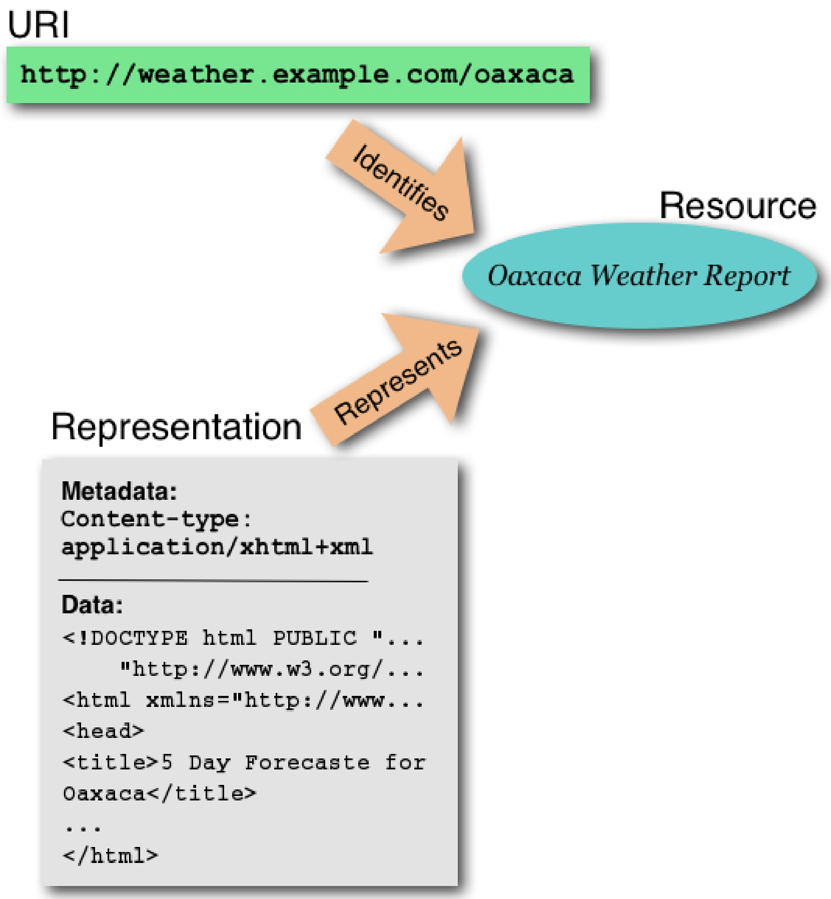
- REST stands for *Representational State Transfer*
- REST is a term coined by Roy T. Fielding to describe an architecture style of networked systems
- RESTful API
  - A resource-based API that uses the HTTP protocol

REST-based network characteristics

- *Client-Server*: a pull-based interaction style
- *Stateless*: the client-server communication is constrained by no client context being stored on the server
- *Cache*: clients and intermediaries can cache responses
- *Uniform interface*: all resources are accessed with a generic interface (e.g., HTTP GET, POST, PUT, DELETE), thus simplifying and decoupling the architecture
- *Named resources:* the system is comprised of resources which are named using a URL (or URI)
- *Interconnected resource representations:* the *representations* of the resources are interconnected using URLs, thereby enabling a client to progress from one state to another

Resources

- Resources
  - every distinguishable entity is a resource.
  - a resource may be a Web site, an HTML page, an XML document, a Web service, a physical device, etc.
- URLs Identify Resources
  - Resources are is uniquely identified by a URL (Axiom 0 of Tim Berners-Lee Web Design)

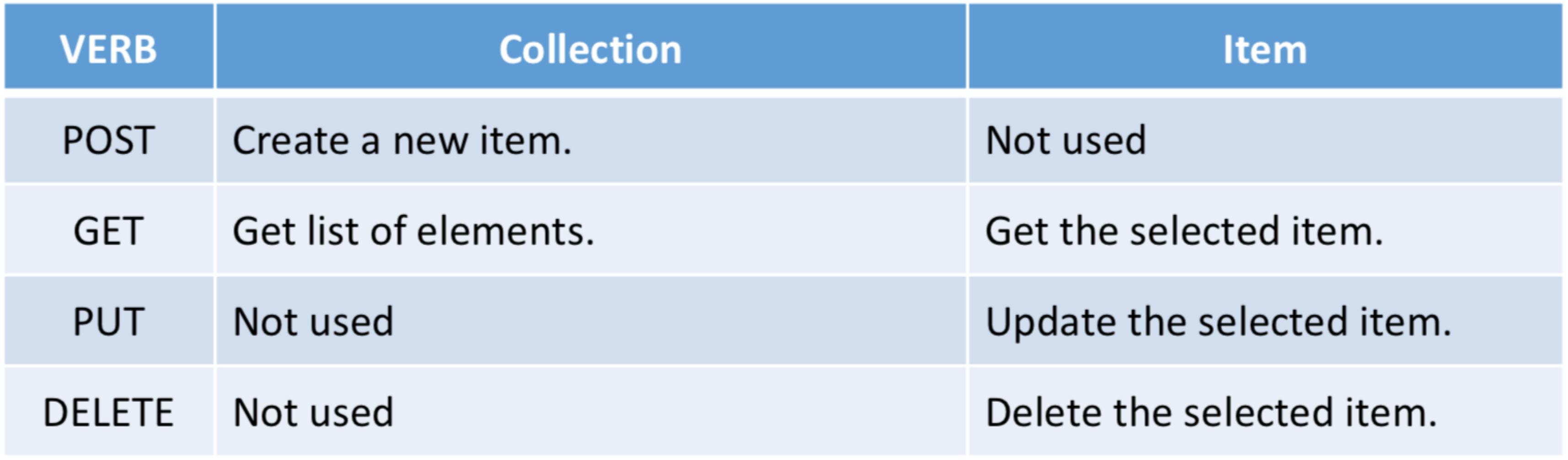


```
Metadata:
Content-type:
application/xhtml+xml
```

```
Data:
<!DOCTYPE html PUBLIC "...
    "http://www.w3.org/...
<html xmlns="http://www...
<head>
<title>5 Day Forecaste for
Oaxaca</title>
...
</html>
```

RESTful API

- The RESTful API uses the available HTTP verbs to perform CRUD operations based on the "context":
  - *Collection*: A set of items (e.g.: /users)
  - *Item*: A specific item in a collection (e.g.: /users/{id})



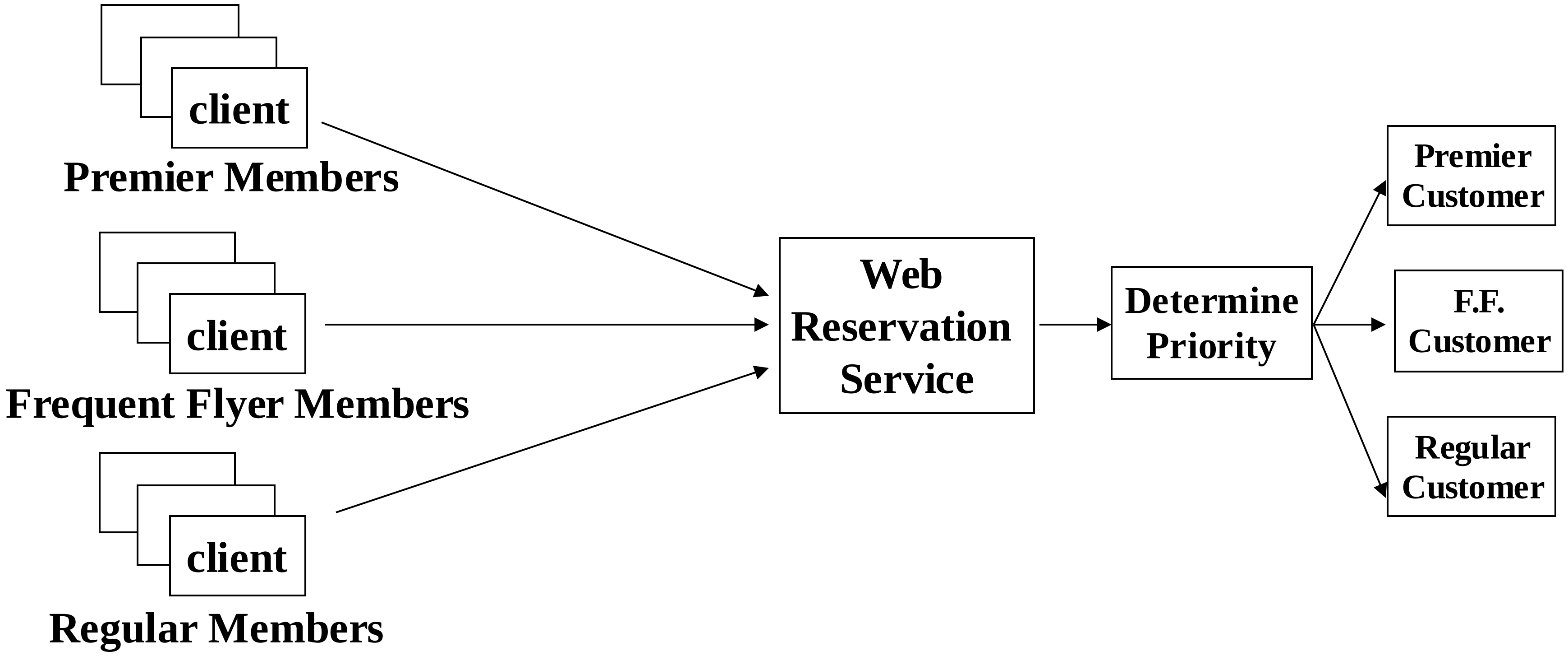
VERB	Collection	Item
POST	Create a new item.	Not used
GET	Get list of elements.	Get the selected item.
PUT	Not used	Update the selected item.
DELETE	Not used	Delete the selected item.

Conventional vs. REST-based design

- Example scenario
  - an airline wants to provide a Web reservation service for customers to make flight reservations through the Web.
  - the airline wants to ensure that its premier members get immediate service, its frequent flyer members get expedited service, all others get regular service.
- Two main approaches to design and implement the Web reservation service
  - Single URL approach: based on conventional web service design
  - Multiple URLs approach: exploits REST-based design

Single URL approach

- The Web service is responsible for examining incoming client requests to determine their priority and process them accordingly

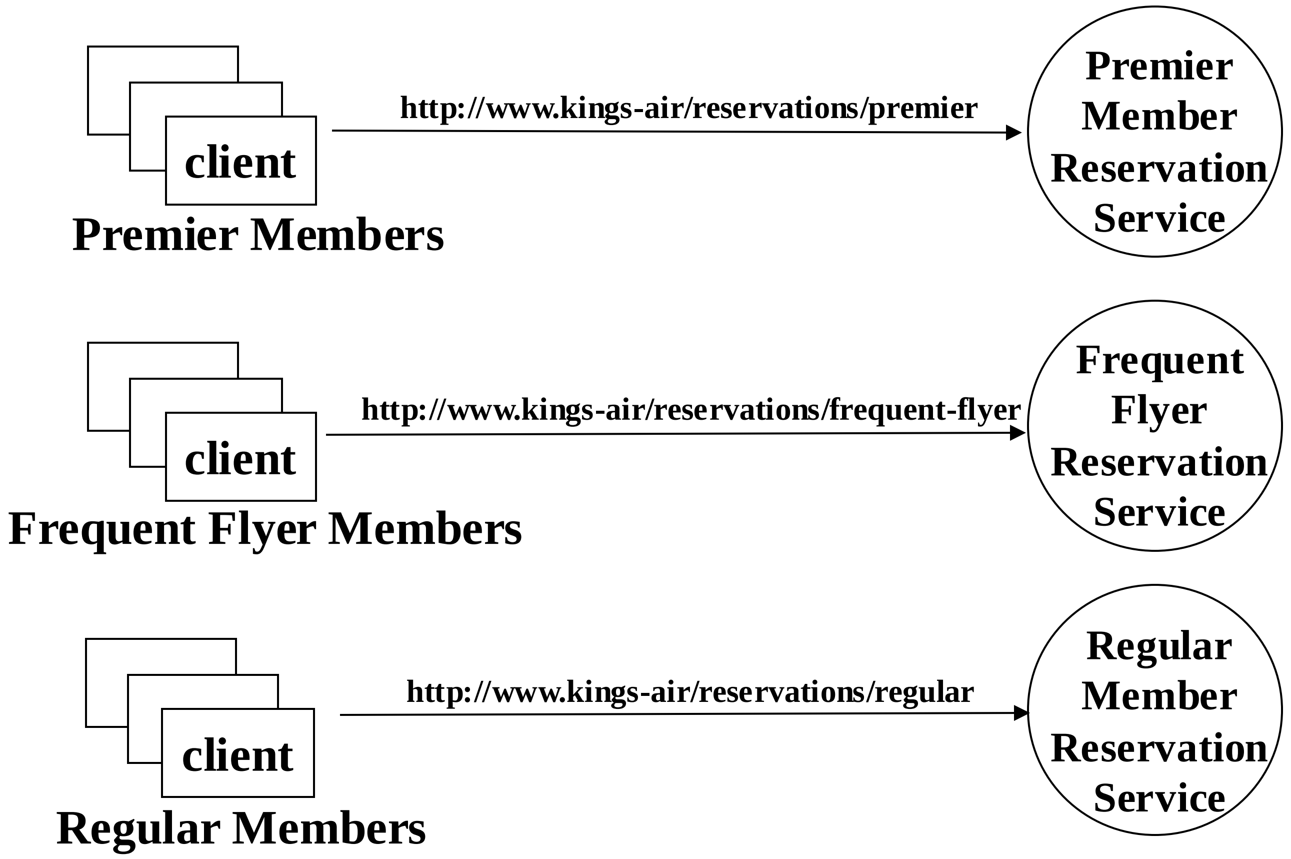


Single URL approach disadvantages

- Clients must learn the rule for expressing priorities, and the Web service application must be written to understand the rule
- Based upon the incorrect assumption that a URL is "expensive" and that their use must be rationed
- The Web service is a central point of failure and a bottleneck
  - Load balancing is a challenge
- It violates Axiom 0 of Tim Berners-Lee Web Design

Multiple URLs approach

- One URL for premier members, a different URL for frequent flyers, and still another for regular customers


client
Premier Members
http://www.kings-air/reservations/premier
Premier Member Reservation Service
client
Frequent Flyer Members
http://www.kings-air/reservations/frequent-flyer
Frequent Flyer Reservation Service
client
Regular Members
http://www.kings-air/reservations/regular
Regular Member Reservation Service

Multiple URLs approach advantages

- It's easy to understand what each service does simply by examining the URL
- There is no need to introduce rules
  - Priorities are elevated to the level of a URL. "What you see is what you get"
- It's easy to implement high priority
  - simply assign a fast machine at the premier member URL.
- There is no bottleneck and no central point of failure
- Consistent with Axiom 0

Software Architectural Transaction Patterns

- A service often encapsulates data or provides access to data that need to be read or updated by clients
- Many services need to provide coordinated update operations
- A **transaction** is a request from a client to a service that consists of two or more operations that perform a single logical function and that must be completed in its entirety or not at all

Transaction properties

- Transactions have the following properties, sometimes referred to as ACID properties:
  - **Atomicity (A).** A transaction is an indivisible unit of work. It is either entirely completed (committed) or aborted (rolled back)
  - **Consistency (C).** After the transaction executes, the system must be in a consistent state
  - **Isolation (I).** A transaction's behavior must not be affected by other transactions
  - **Durability (D).** Changes are permanent after a transaction completes. These changes must survive system failures. This is also referred to as *persistence*

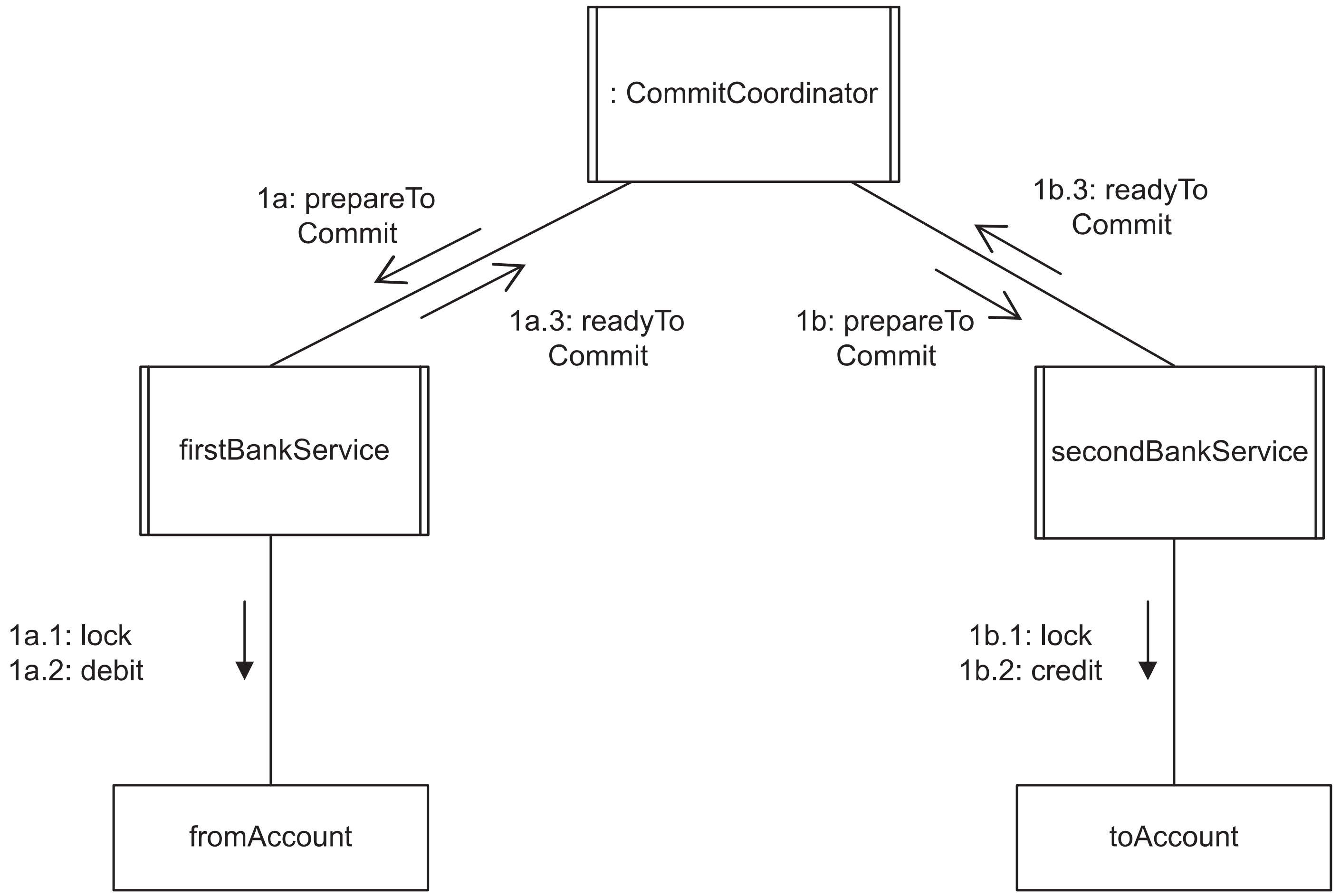
Example Banking Transaction (*Transfer*)

- Consider a transfer transaction between two accounts – for example, from a savings account to a checking account – in which the accounts are maintained at two separate banks (services)
- In this case, it is necessary to debit the savings account and credit the checking account
- Therefore, the transfer transaction consists of two operations that must be atomic – a debit operation and a credit operation – and the transfer transaction must bre either committed or aborted:
  - **Committed.** Both credit and debit operations occur
  - **Aborted.** Neither the credit nor the debit operation occurs

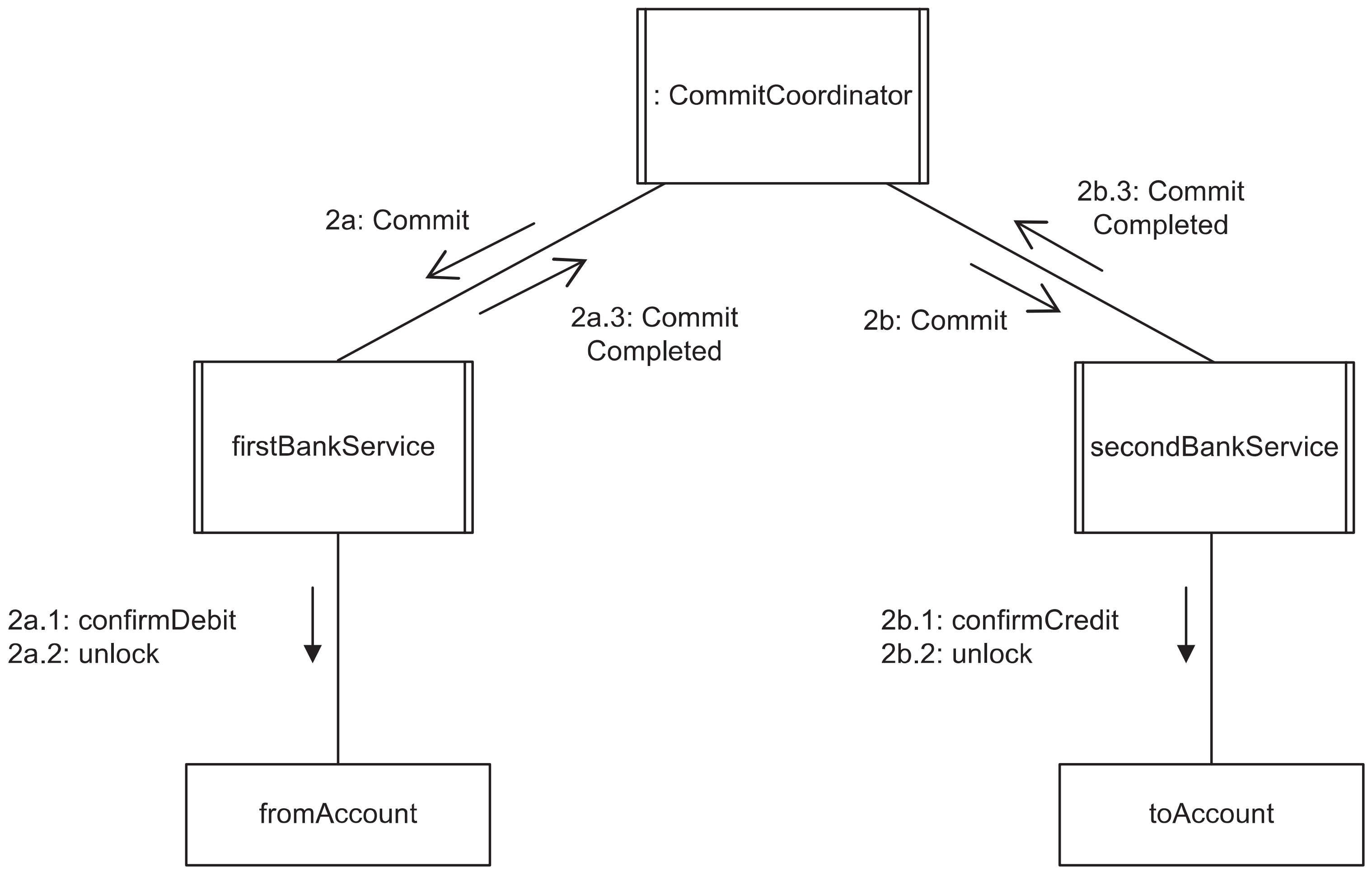
Two-Phase Commit Protocol

- The **Two-Phase Commit Protocol** pattern addresses the problem of managing atomic transactions in distributed systems, by synchronizing updates on different nodes
- Coordination of the transaction is provided by the **`CommitCoordinator`**
- There is one participant service for each node
- There are two participants in the bank transfer transaction:
  - **`firstBankService`**, which maintains the account *from* which money is being transferred (from Account), and
  - **`secondBankService`**, which maintains the account *to* which money is being transferred (to Account)

First Phase


: CommitCoordinator
1a: prepareTo Commit
1a.3: readyTo Commit
1b: prepareTo Commit
1b.3: readyTo Commit
firstBankService
secondBankService
1a.1: lock
1a.2: debit
1b.1: lock
1b.2: credit
fromAccount
toAccount

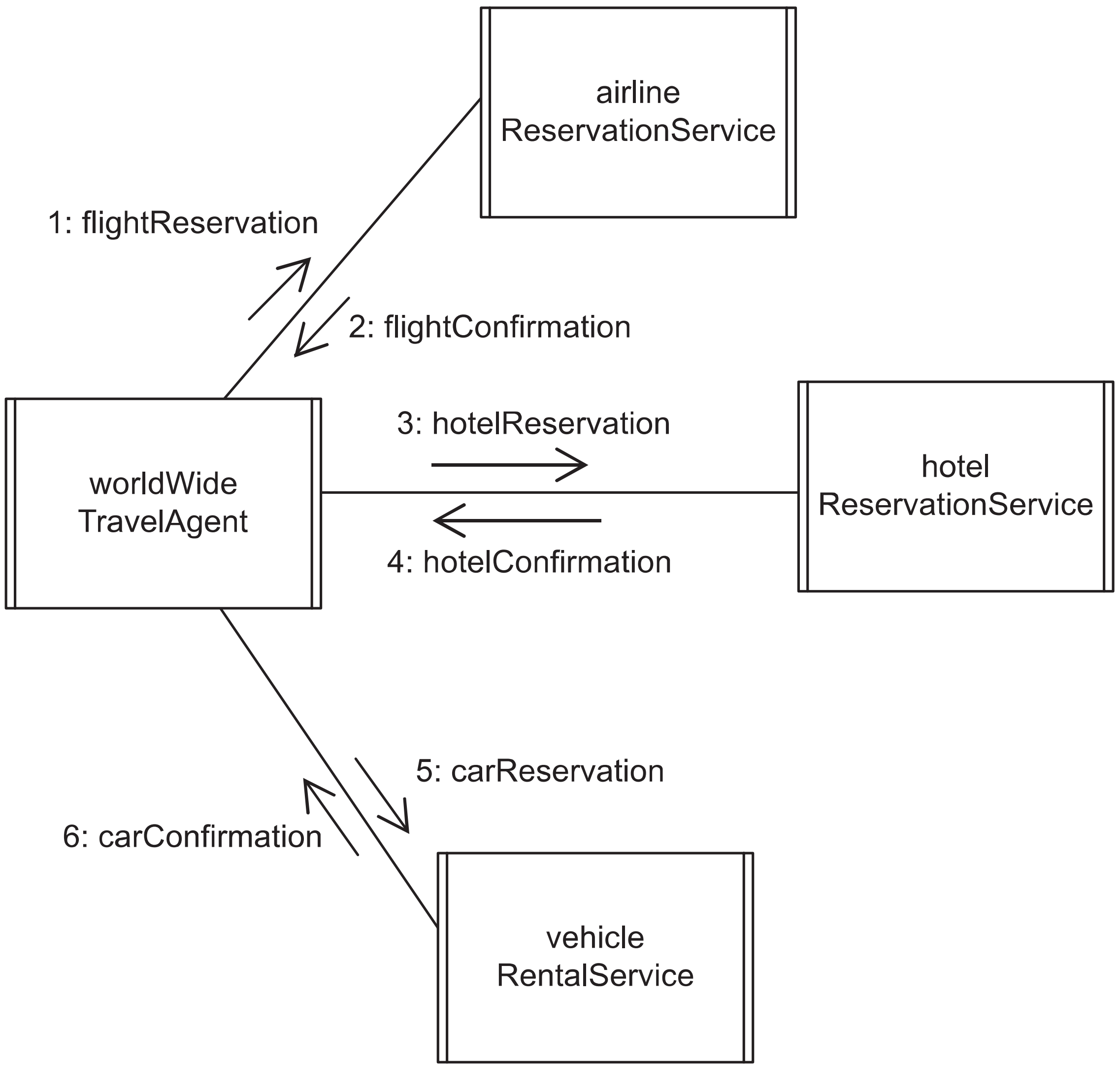
Second Phase



Compound Transaction Pattern

- The previous bank transfer transaction is an example of a flat transaction, which has an “all-or-nothing” characteristic
- A compound transaction, in contrast, might need only a partial rollback
- The **Compound Transaction** pattern can be used when the client’s transaction requirement can be broken down into smaller flat atomic transactions, in which each atomic transaction can be performed separately and rolled back separately
- For example, if a travel agent makes an airplane reservation, followed by a hotel reservation and a rental car reservation, it is more flexible to treat this reservation as consisting of three flat transactions. Treating the transaction as a compound transaction allows part of a reservation to be changed or canceled without the other parts of the reservation being affected.

Example Compound Transaction Pattern



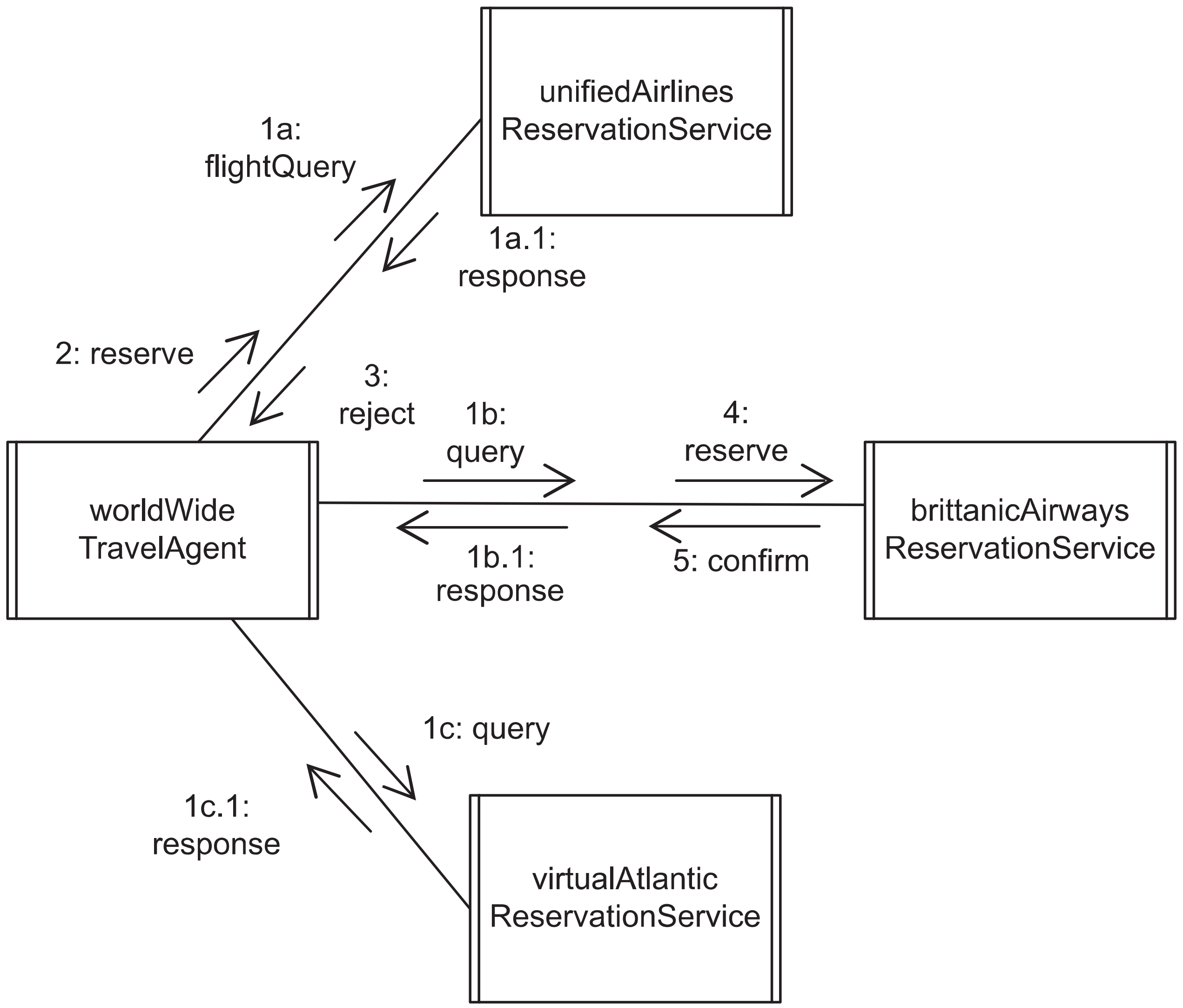
Long-Living Transaction Pattern

- A *long-living transaction* is a transaction that has a human in the loop and that could take a long and possibly indefinite time to execute, because individual human behavior is unpredictable
- The **Long-Living Transaction** pattern splits a long-living transaction into two or more separate transactions (usually two) so that human decision making takes place between the successive pairs (such as first and second) of transactions.

Example Long-Living Transaction

- Consider an airline reservation with human involvement in the transaction
- First a query transaction displays the available seats
- The query transaction is followed by a reserve transaction
- With this approach, it is necessary to recheck seat availability before the reservation is made
- A seat available at query time might no longer be available at reservation time because several agents might be querying the same flight at the same time
- If only one seat is available, the first agent will get the seat but not the others

Example Long-Living Transaction Pattern



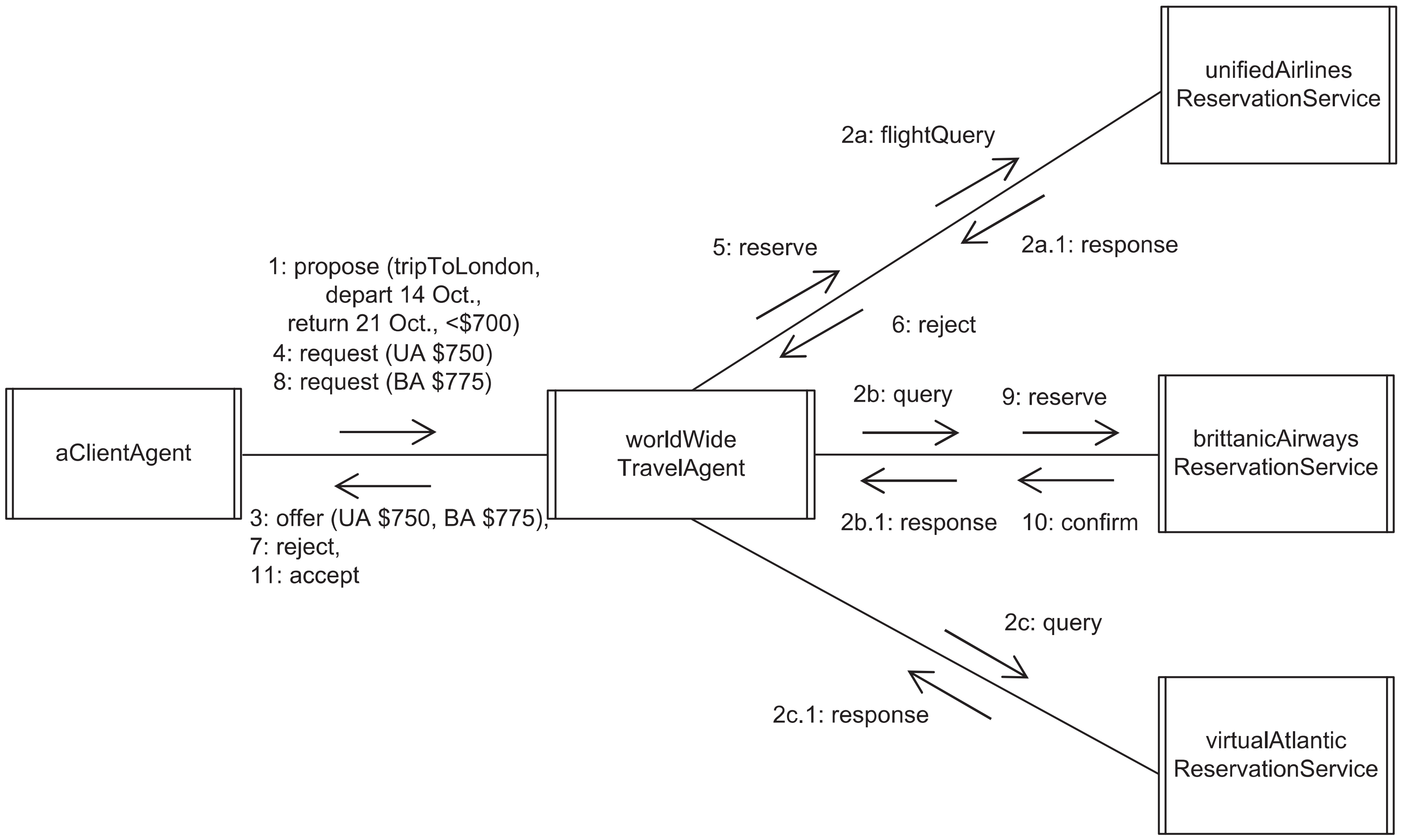
Negotiation Pattern

- In some SOAs, the coordination between services involves negotiations between software agents so that they can cooperatively make decisions
- In the **Negotiation** pattern (also known as the *Agent-Based Negotiation* or *Multi-Agent Negotiation* pattern), a client agent acts on behalf of the user and makes a proposal to a service agent
- The service agent attempts to satisfy the client's proposal, which might involve communication with other services
- Having determined the available options, the service agent then offers the client agent one or more options that come closest to matching the original client agent proposal
- The client agent may then request one of the options, propose further options, or reject the offer
- If the service agent can satisfy the client agent request, it accepts the request; otherwise, it rejects the request

Negotiation Services

- The client agent, who acts on behalf of the client, may do any of the following:
  - **Propose a service.** The client agent proposes a service to the service agent. This proposed service is *negotiable,* meaning that the client agent is willing to consider counteroffers
  - **Request a service.** The client agent requests a service from the service agent. This requested service is *nonnegotiable,* meaning that the client agent is not willing to consider counteroffers
  - **Reject a service offer.** The client agent rejects an offer made by the service agent
- The service agent, who acts on behalf of the service, may do any of the following:
  - **Offer a service.** In response to a client proposal, a service agent offers a counter- proposal
  - **Reject a client request/proposal.** The service agent rejects the client agent’s proposed or requested service
  - **Accept a client request/proposal.** The service agent accepts the client agent’s proposed or requested service

Example Negotiation Pattern



Service Interface Design in SOA

- New services are initially designed by using class structuring criteria
- During dynamic interaction modeling, the interaction between client objects and service objects is determined
- The approach taken for designing service operations is similar to that used in class interface design
- The messages arriving at a service form the basis for designing the service operations. The messages are analyzed to determine the name of the operation, as well as to determine the input and output parameters

Service Coordination in SOA

- In SOA applications that involve multiple services, coordination of these services is usually required
- To ensure loose coupling among the services, it is often better to separate the details of the coordination from the functionality of the individual services
- In SOA, different types of coordination are provided, including ***orchestration*** and ***choreography***

Orchestration and Choreography

- **Orchestration** consists of centrally controlled workflow coordination logic for coordinating multiple participant services
  - This allows the reuse of existing services by incorpo-rating them into new service applications
- **Choreography** provides distributed coordination among services, and it can be used when coordination is needed between different business organizations
  - Thus, choreography can be used for collaboration between services from different service providers provided by different business organizations
  - Whereas orchestration is centrally controlled, choreography involves distributed control

Coordination

- Because the terms *orchestration* and *choreography* are often used interchangeably, the more general term **coordination** is used to describe the control and sequencing of different services as needed by a SOA application, whether they are centrally controlled or involve distributed control.
- Transaction patterns can also be used for service coordination

Detailed OOD

- *Preliminary OOD* provides the HW/SW execution platform to which the detailed design sub-phase must conform
- The main part of the detailed OOD sub-phase is devoted to refine what produced in the OOA phase
- The objective is to transform the OOA models, which have been defined in the problem domain, into *models defined in the solution domain*, which in turn will be used in the coding phase
- The detailed OOD sub-phase provides the detailed design of the architectural unites identified in the preliminary OOD sub-phase, through the *addition of technical details* (or by *producing additional models* at a decreased level of abstraction)
- The detailed OOD sub-phase defines the **collaboration between objects**, which is at the basis of each object-oriented program
- Such a collaboration is defined through the **realization of use cases and operations**
- Collaborations are to OOD what use cases are to OOA
  - if use cases drive OOA then collaborations drive OOD

Realization of use cases

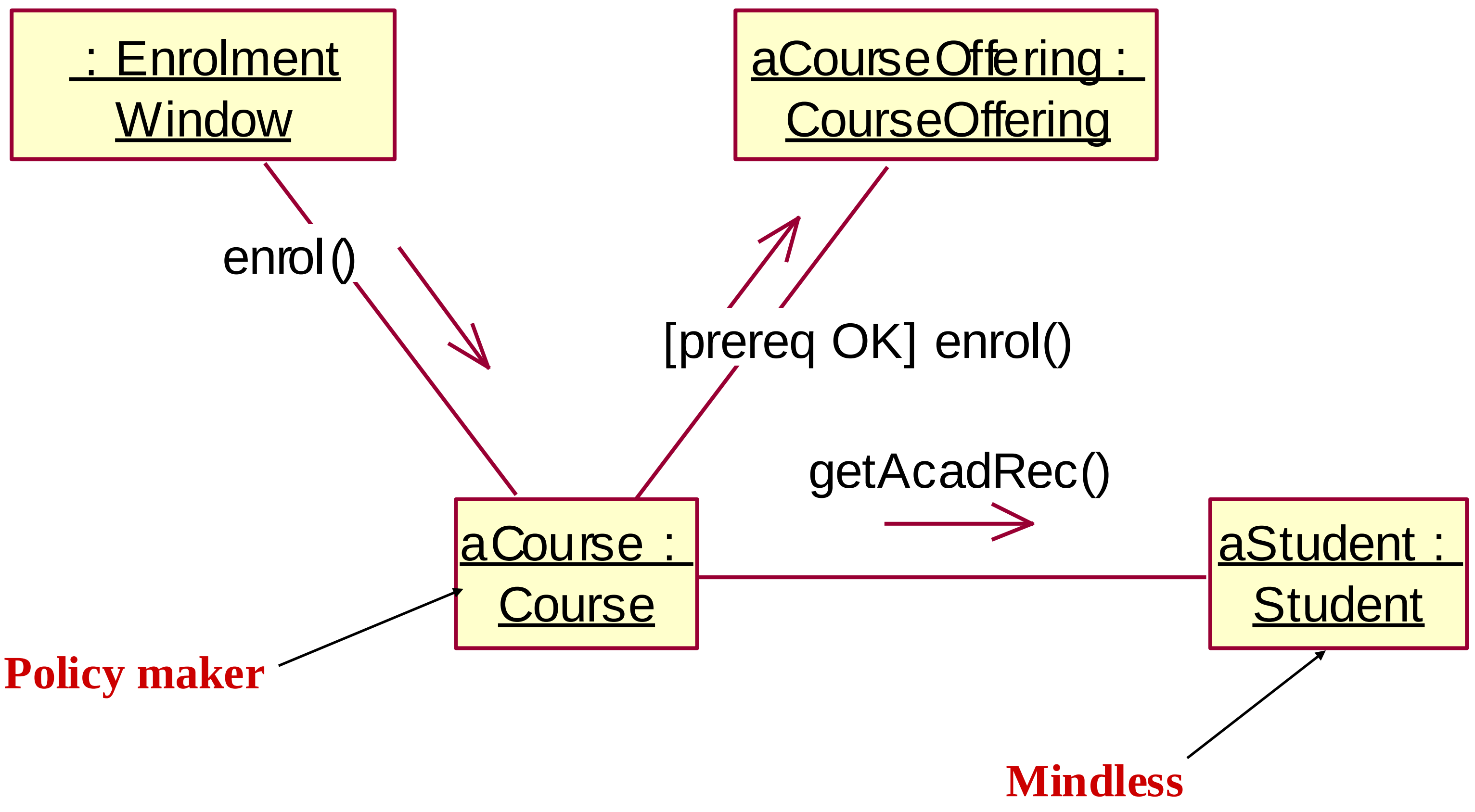
- *Use cases* introduced at OOA time are realized through *collaborations* at OOD time
- A single use case is typically realized by a set of collaborations, due to the different level of abstraction
- A collaboration has:
  - a **behavioral part**
    - represents the *dynamics* that shows how the static elements collaborate
    - defined by use of *communication diagrams*
  - a **structural part**
    - represents *static aspects of collaboration*
    - defined by elaborating the *OOA class diagram* with the implementation details, leading to *composite structure diagrams*

Collaboration – *control management*

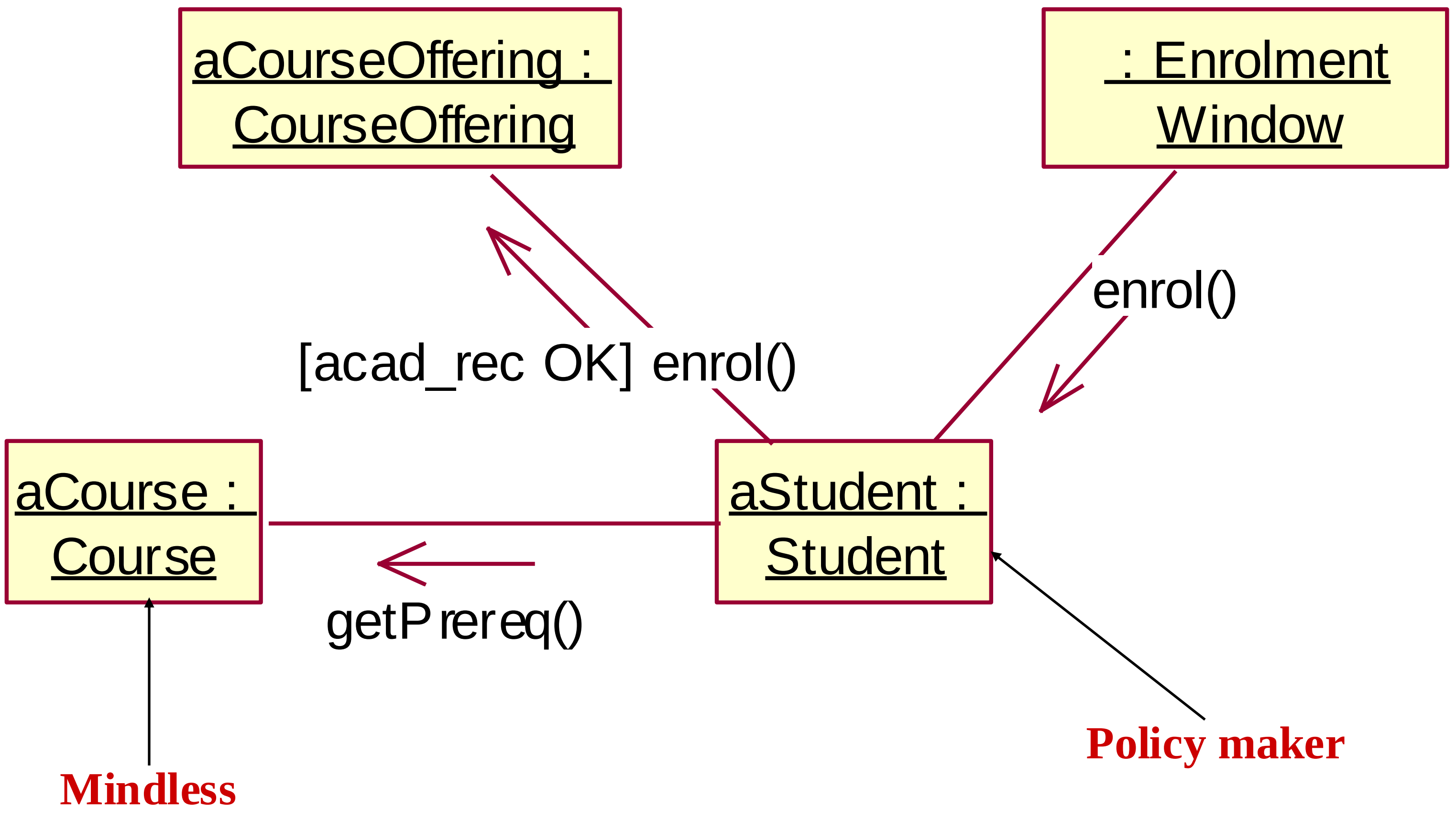
University Enrolment system

- Example
  - add a student (`Student`) to a course offering (`CourseOffering`)
- Actions to be executed
  1. identify the prerequisite courses for the course offering
  2. check if the student satisfies the prerequisites
- Consider that:
  - the `enrol()` message is sent by the boundary object `:EnrolmentWindow`
  - three entity classes are involved – `CourseOffering`, `Course` and `Student`
- At least four solutions possible (with different *class coupling* characteristics)

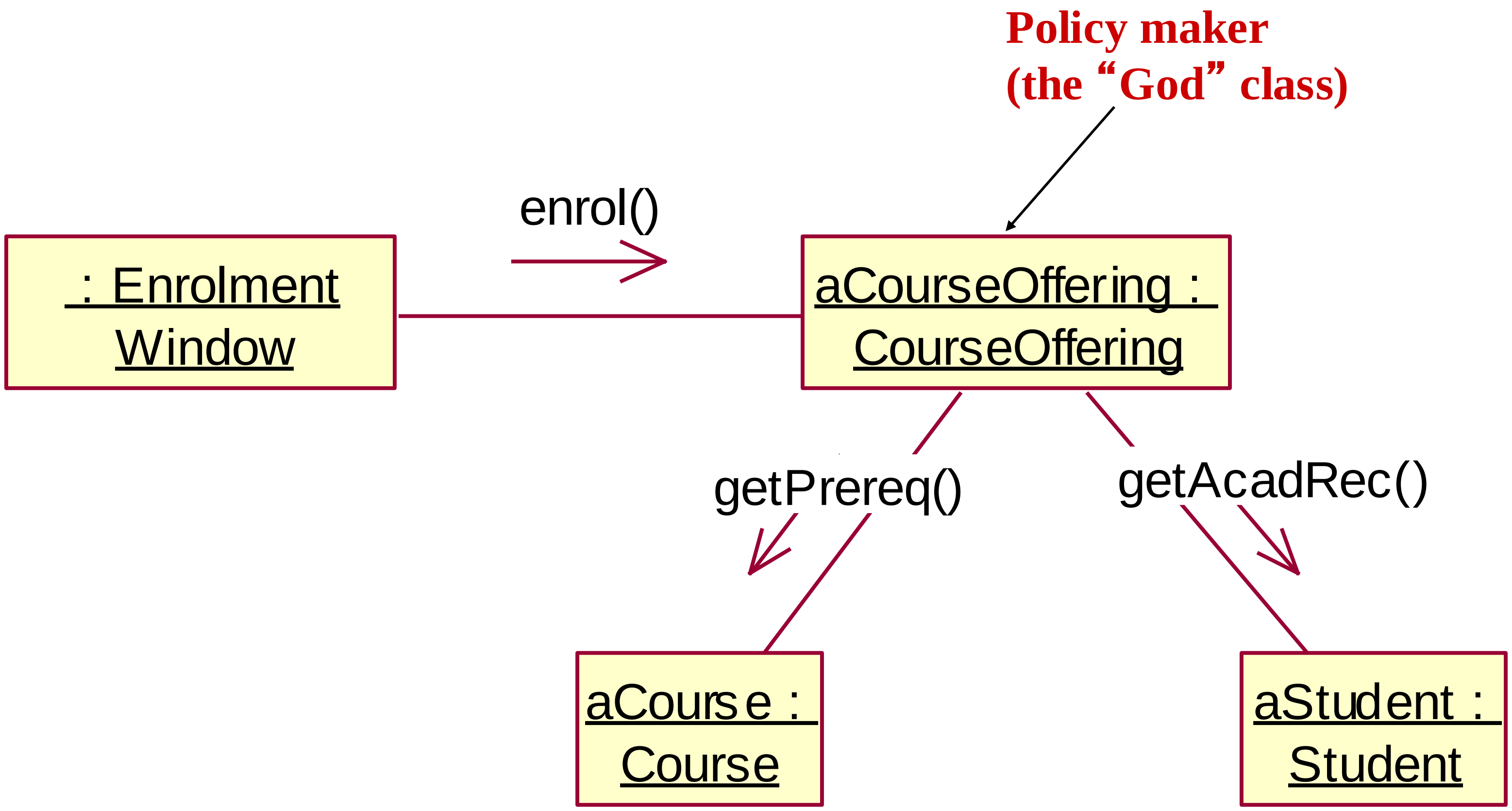
Control management - *solution 1*



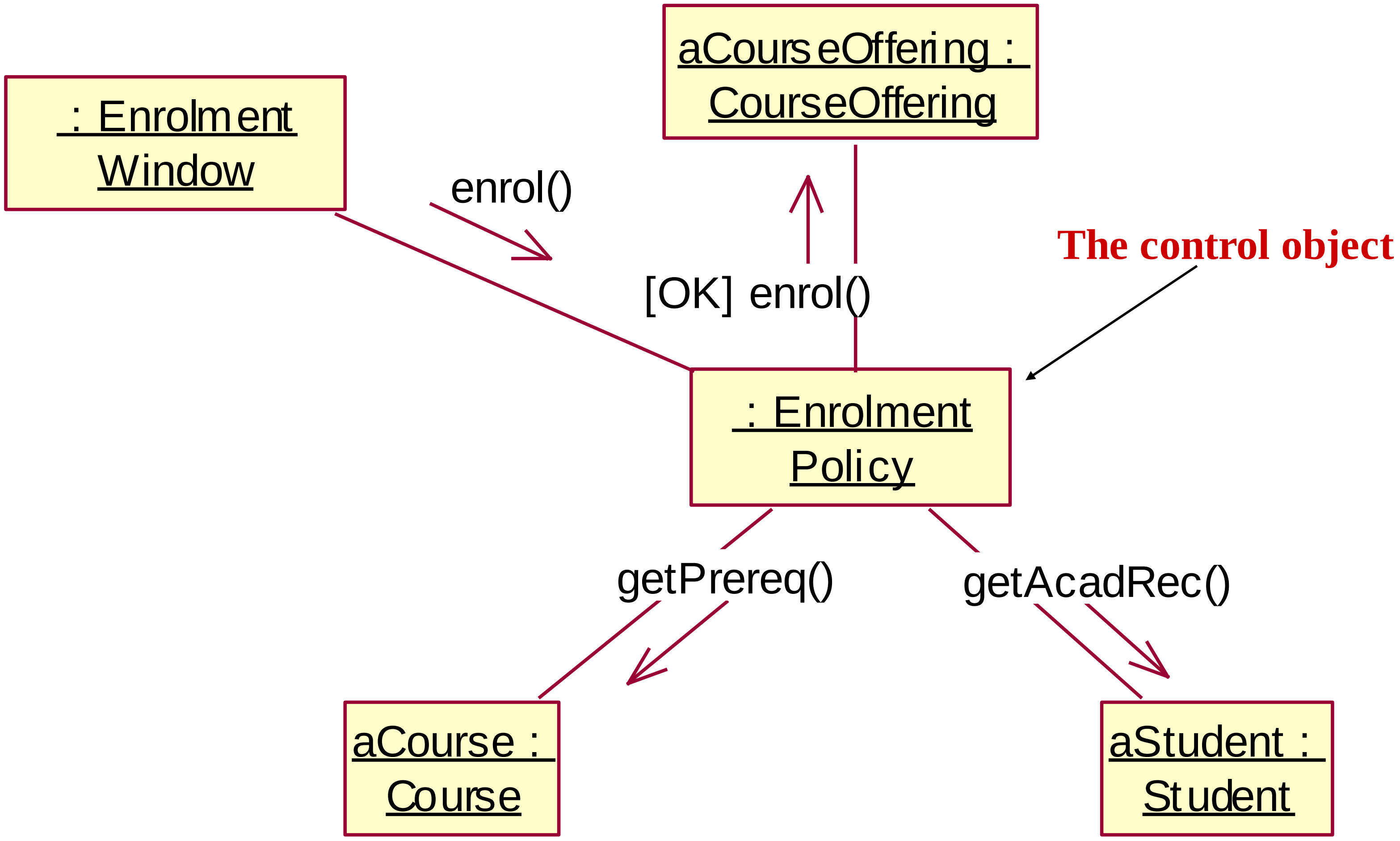
Control management – *solution 2*



Control management – *solution 3*



Control management – *solution 4*



Boundary-Control-Entity (BCE) Approach!

Coupling issues

- The BCE approach specifies three layers of classes
- Objects communicate within a layer and between the adjacent layers
- *Intra-layer coupling*
  - desirable
  - localizes software maintenance and evolution to individual layers
- *Inter-layer coupling*
  - to be minimized
  - communication interfaces to be carefully defined
- The *Law of Demeter* can be used to reduce the inter-layer class coupling

Law of Demeter

- The target of a message (in class methods) can only be one of the following objects:
  1. The method’s object itself (i.e. **`this`** in C++ and Java, **`self`** and **`super`** in Smalltalk)
  2. An object that is an argument in the method’s signature
  3. An object referred to by the object’s attribute (*strong law* ➔ inherited attributes cannot be used to identify the target object)
  4. An object created by the method
  5. An object referred to by a global variable
- Also known as the “*don’t talk to strangers*”

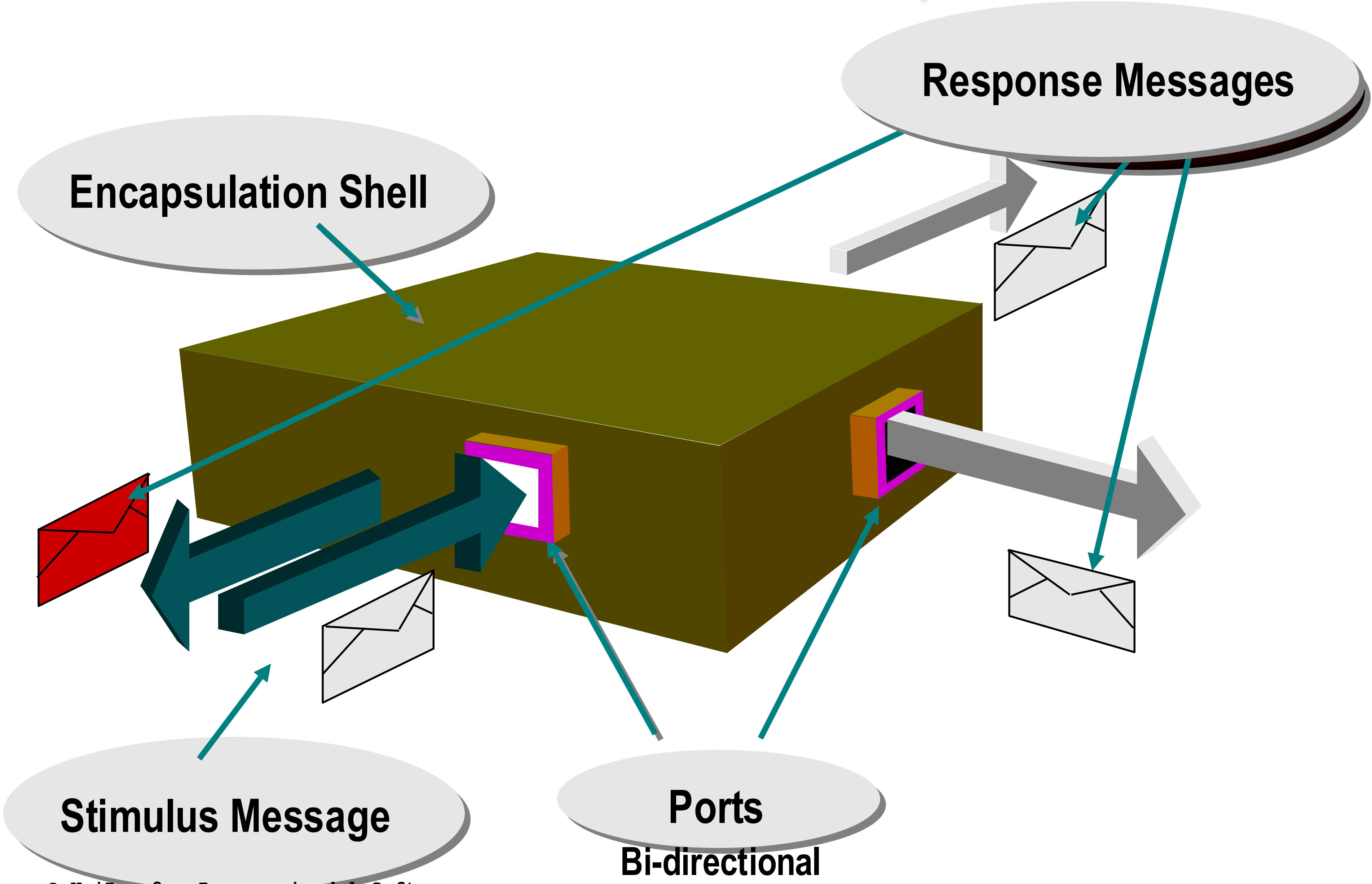
UML Structured Class

- A structured class contains *roles* or *parts* that form its structure and realize its behavior
  - Describes the internal implementation structure
- The *parts* themselves may also be structured classes
  - Allows hierarchical structure to permit a clear expression of multilevel models
- A *connector* is used to represent an association in a particular context
  - Represents communications paths among parts

Structured Class Usage

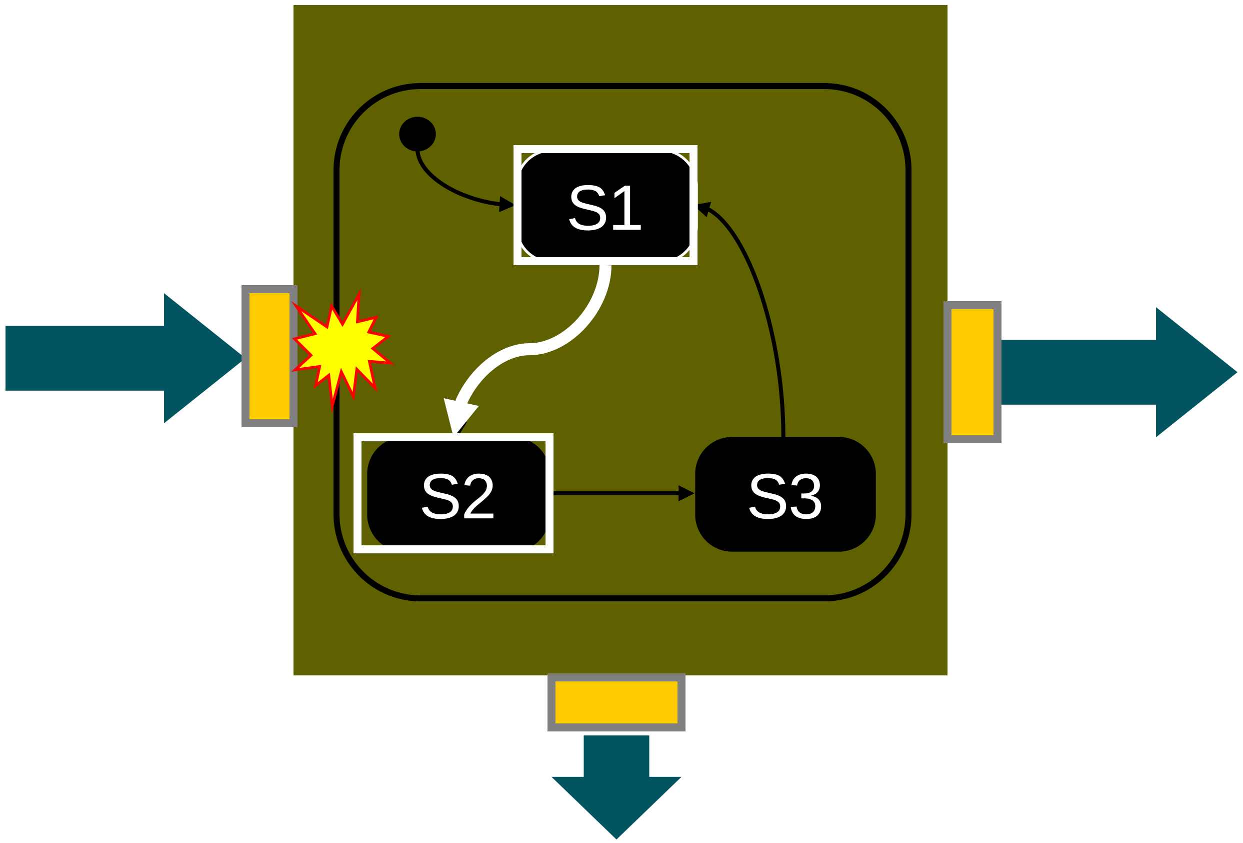
- Can be used as the primary building blocks of an application
  - Provides graphical representation of design elements
  - Can *hide implementation details*
    - Powerful abstraction tool - same construct applies to multiple semantic levels
    - Clear communication and understanding of system architecture
  - *Strict encapsulation of behavior*
    - Interactions restricted to message-based communications passed through external interfaces (ports)

Structured Class: Conceptual View



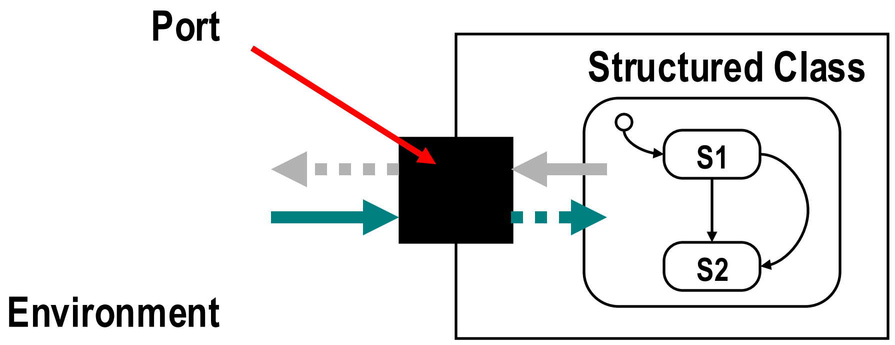
Structured Class: Behavior

- Optional hierarchical state machine
  - State based signal handler

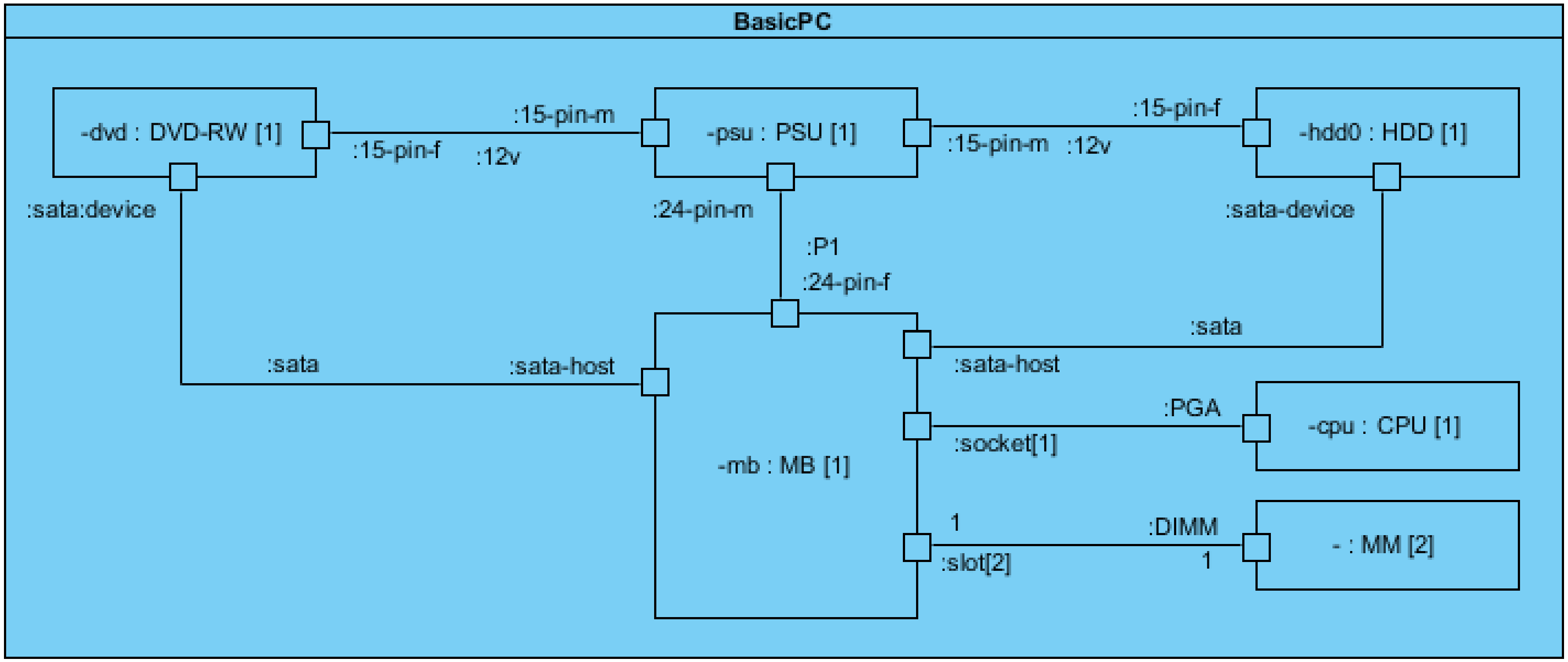


Structured Class: Autonomous Design Unit

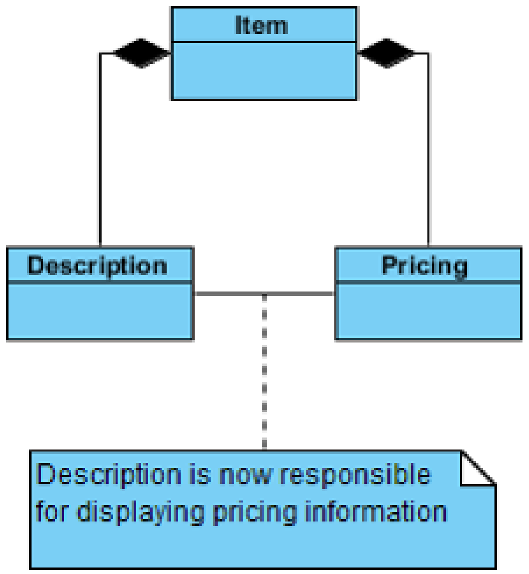
- Strict encapsulation ensures that the implementation is independent from the environment
  - Ports can play a bi-directional mediation role
    - Environment only sees the port of the structured class
    - The internal behavior is built to the “specification” provided by its interface
  - Structured classes can be independently designed, and unit tested



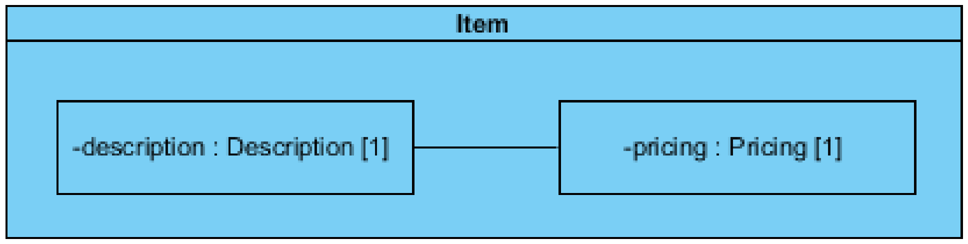
Example:
UML Composite Structure Diagram



Class Diagram vs. Composite Structure Diagram


Item
Description
Pricing
Description is now responsible for displaying pricing information

≠

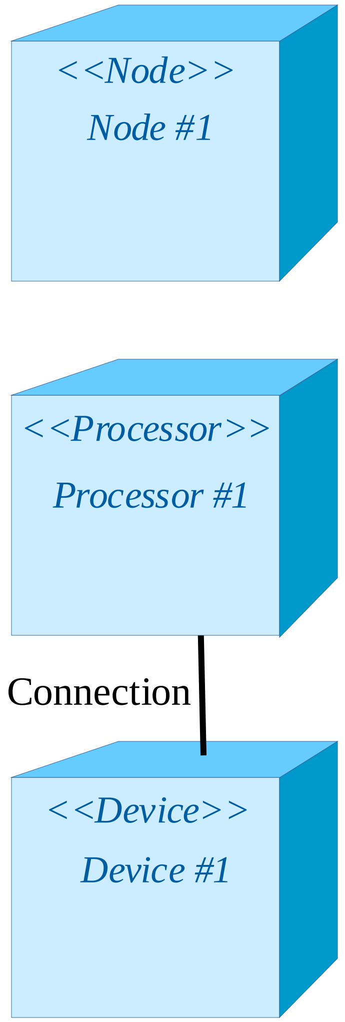


Platform Configuration

- A platform configuration describes the hardware/software solution that defines how the functionality of the system can be distributed across physical nodes
  - Explain the relationship between model elements and their implementation, as well as their deployment
- It is obtained by:
  - defining the platform configuration by use of a *deployment diagram*
  - allocating system elements (artifacts) to nodes of the deployment diagrams

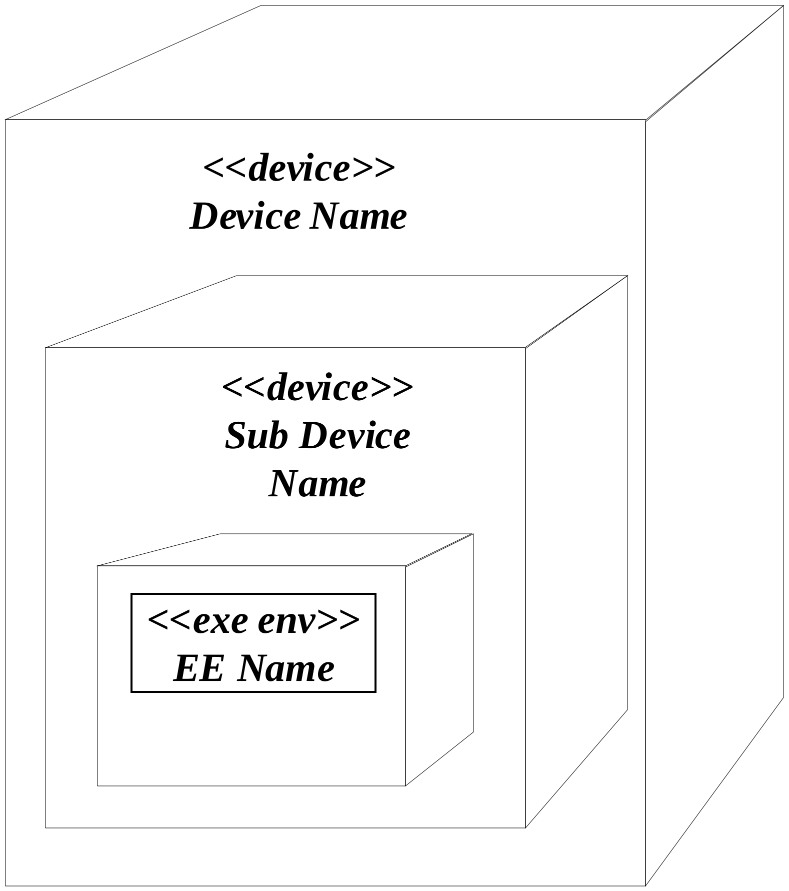
Deployment Model Modeling Elements

- Node
  - Physical run-time computational resource
  - Processor node - Executes system software
  - Device node
    - Support device
    - Typically controlled by a processor
- Connection
  - Communication mechanism
  - Physical medium
  - Software protocol



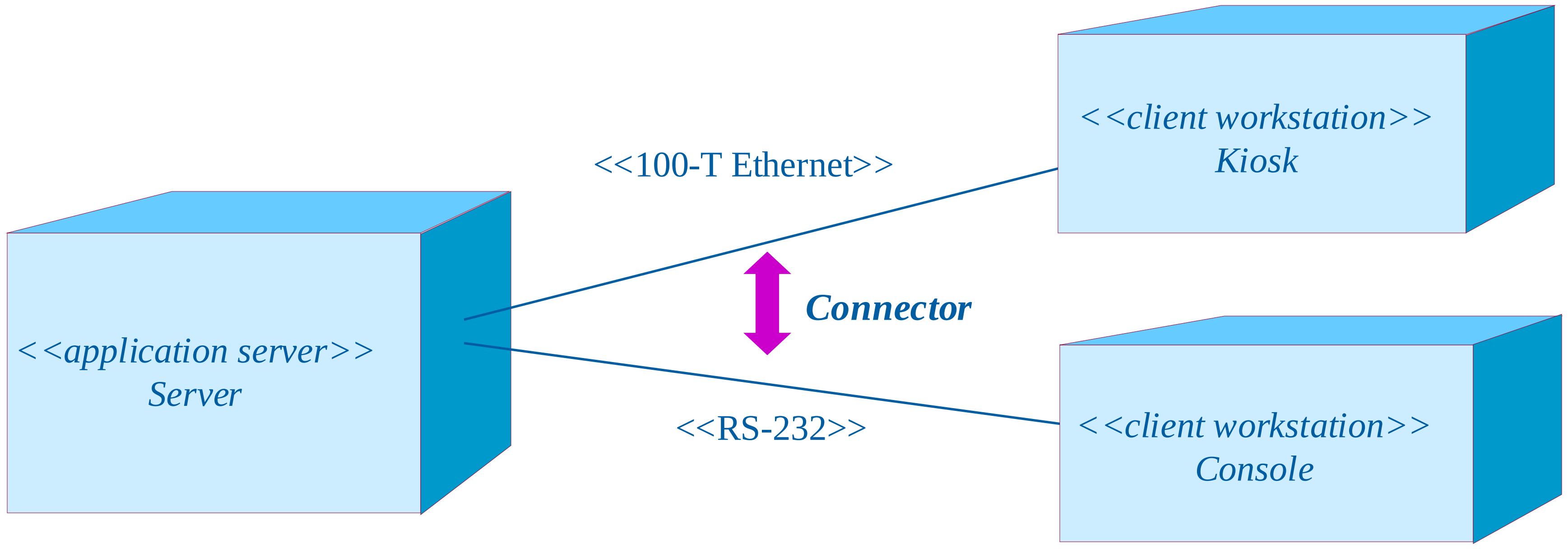
What Is a Node?

- Represents a run-time computational resource, and generally has at least memory and often processing capability.
- Types:
  - *Device* - Physical computational resource with processing capability. Devices may be nested
  - *Execution Environment* - Represents particular execution platforms

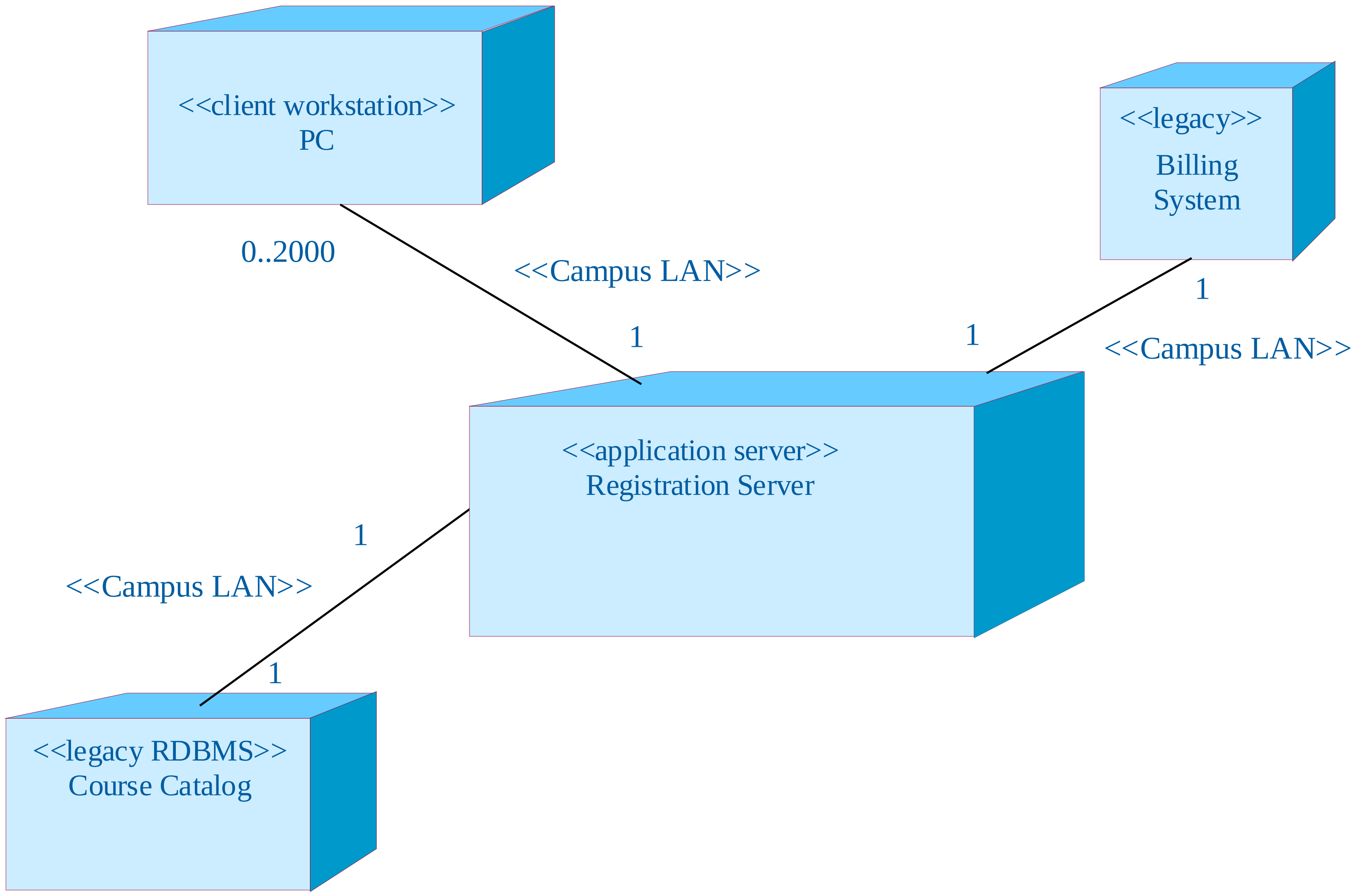


What Is a Connector?

- A connector represents a communication mechanism described by:
  - Physical medium
  - Software protocol



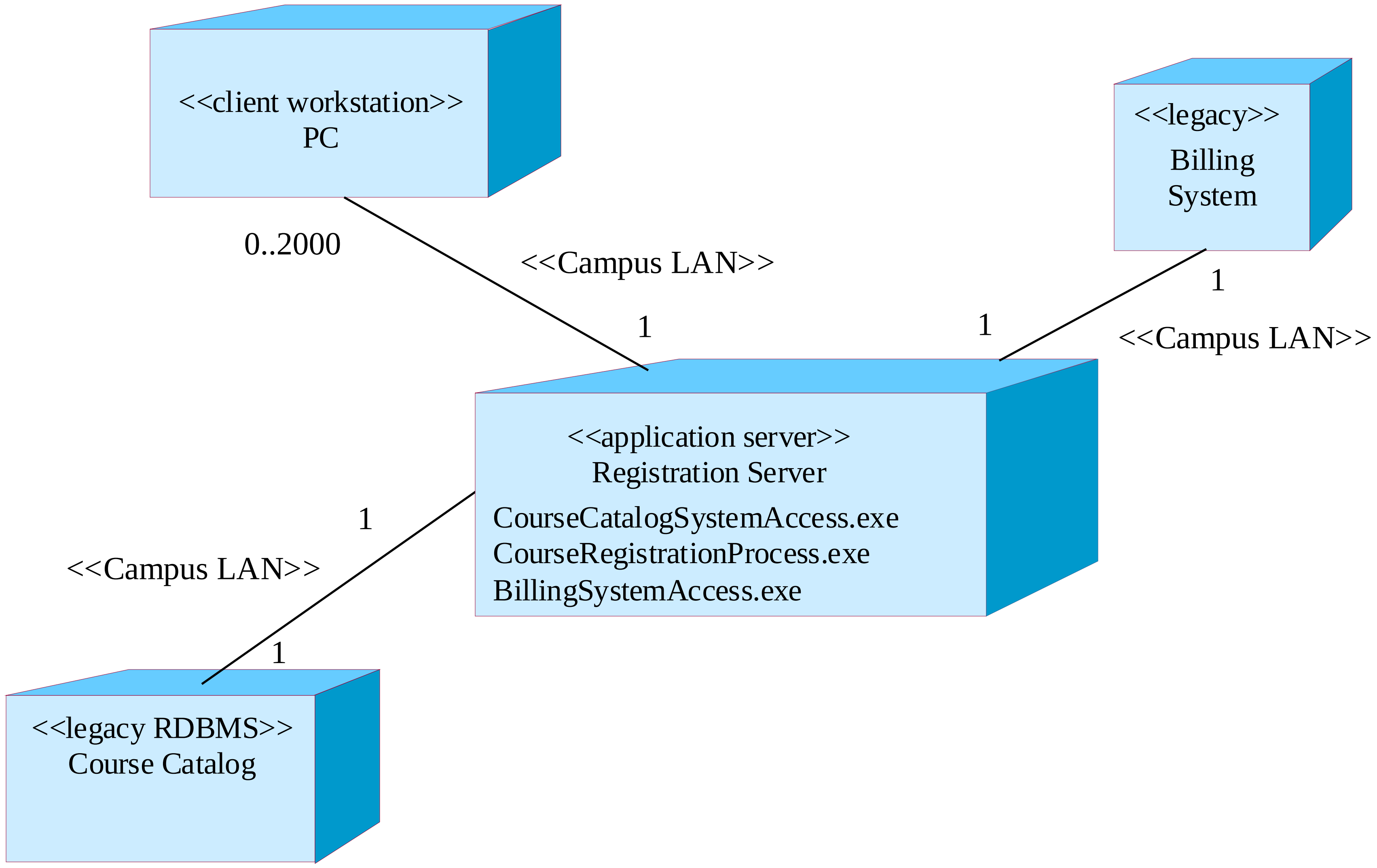
Example: Deployment Diagram



Process-to-Node Allocation Considerations

- Distribution patterns
- Response time and system throughput
- Minimization of cross-network traffic
- Node capacity
- Communication medium bandwidth
- Availability of hardware and communication links
- Rerouting requirements

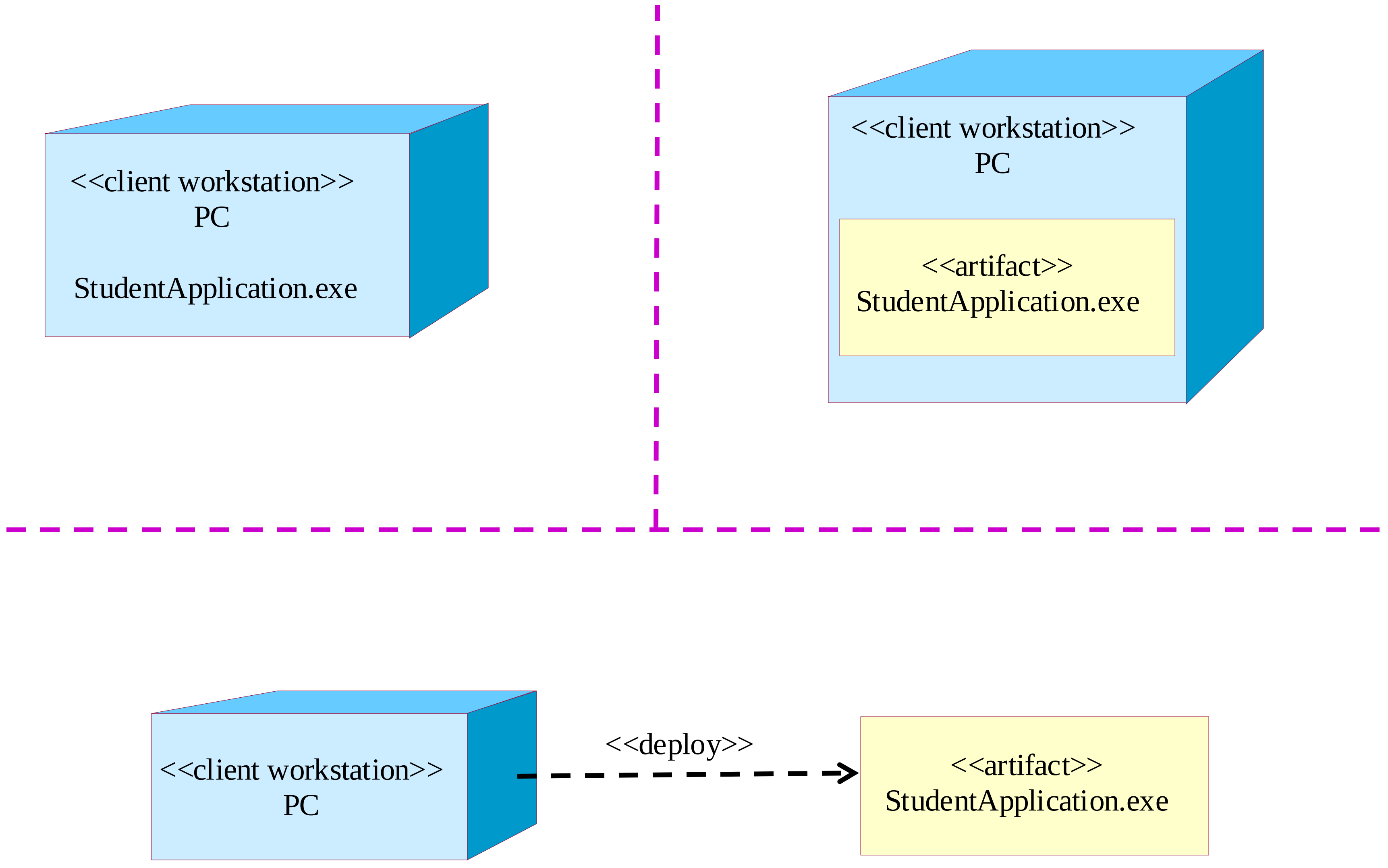
Example: Deployment Diagram with Processes



What is Deployment?

- Deployment is the assignment, or mapping, of software artifacts to physical nodes during execution
  - Artifacts are the entities that are deployed onto physical nodes
    - Processes are assigned to computers
- Artifacts model physical entities
  - Files, executables, database tables, web pages, and so on.
- Nodes model computational resources
  - Computers, storage units

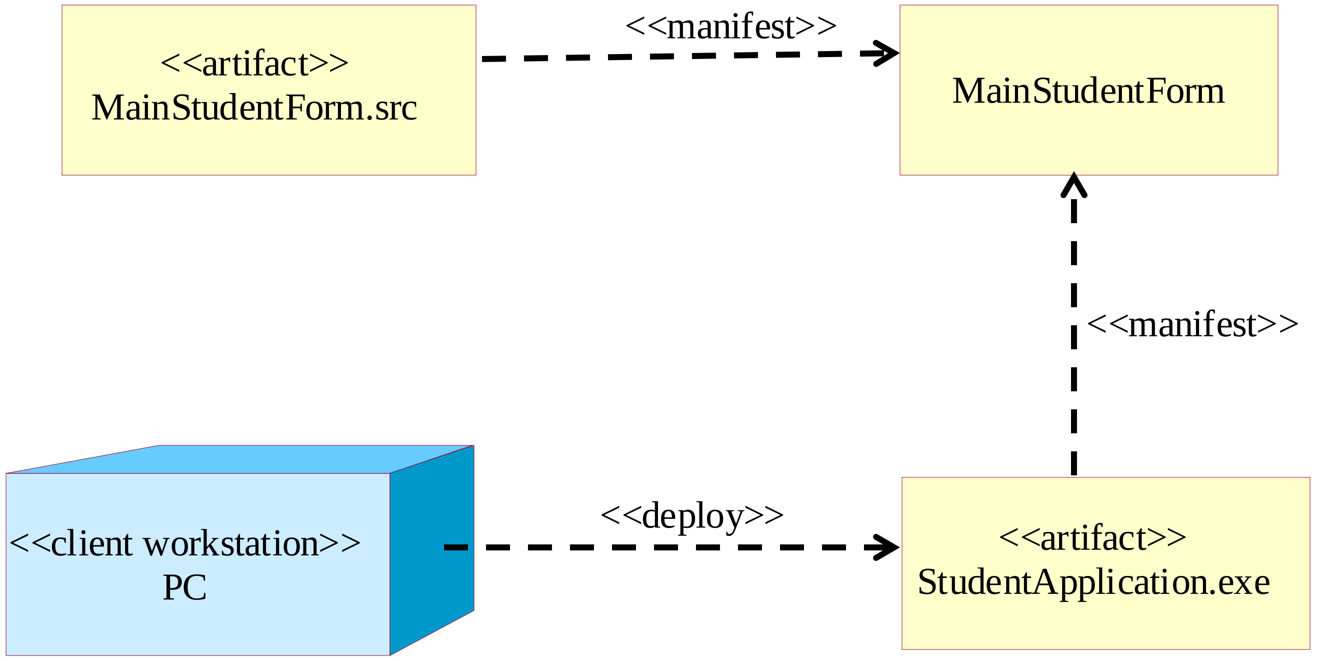
Example: Deploying Artifacts to Nodes



What is Manifestation?

- The physical implementation of a model element as an artifact.
  - A relationship between the model element and the artifact that implements it
  - Model elements are typically implemented as a set of artifacts.
  - Examples of Model elements are source files, executable files, documentation file

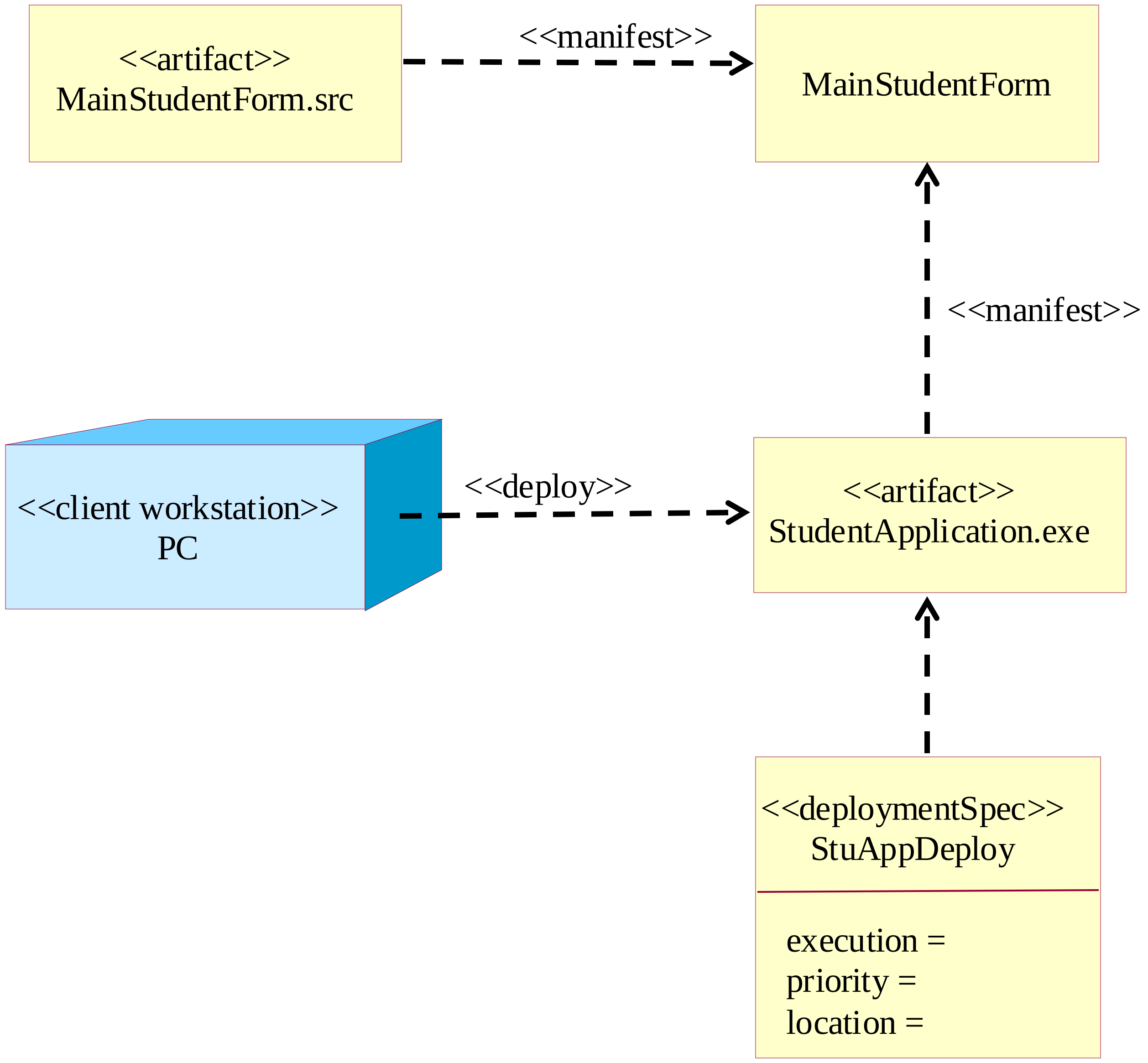
Example: Manifestation



What is a Deployment Specification?

- A detailed specification of the parameters of the deployment of an artifact to a node
  - May define values that parameterize the execution

Example: Deployment Specification


<<artifact>>
MainStudentForm.src
<<manifest>>
MainStudentForm
<<manifest>>
<<client workstation>>
PC
<<deploy>>
<<artifact>>
StudentApplication.exe
<<deploymentSpec>>
StuAppDeploy
execution =
priority =
location =