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

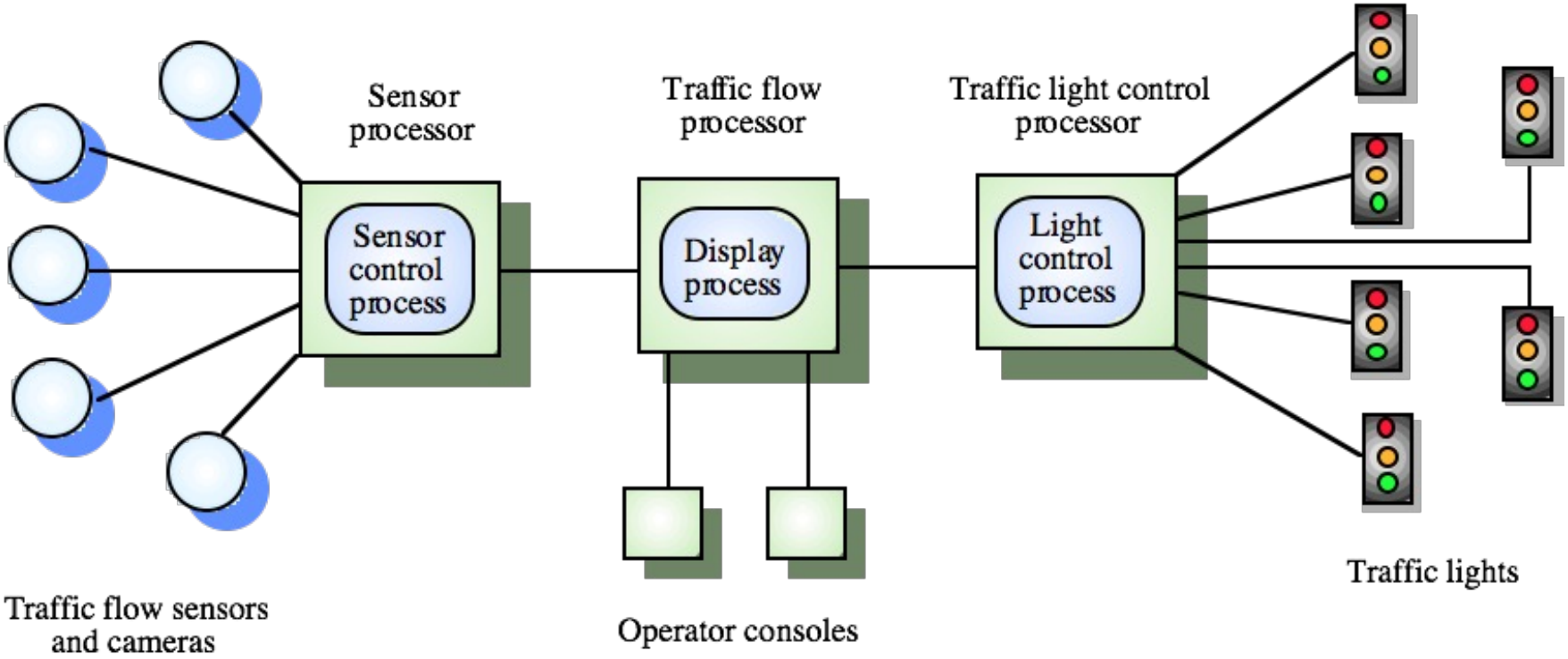
- A **system architecture** defines the structure of the software system components along with the relationships between such components and the principles driving design and system evolution
- Evolution of system architectures:
  1. Mainframe-based architectures
  2. File sharing architectures
  3. 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

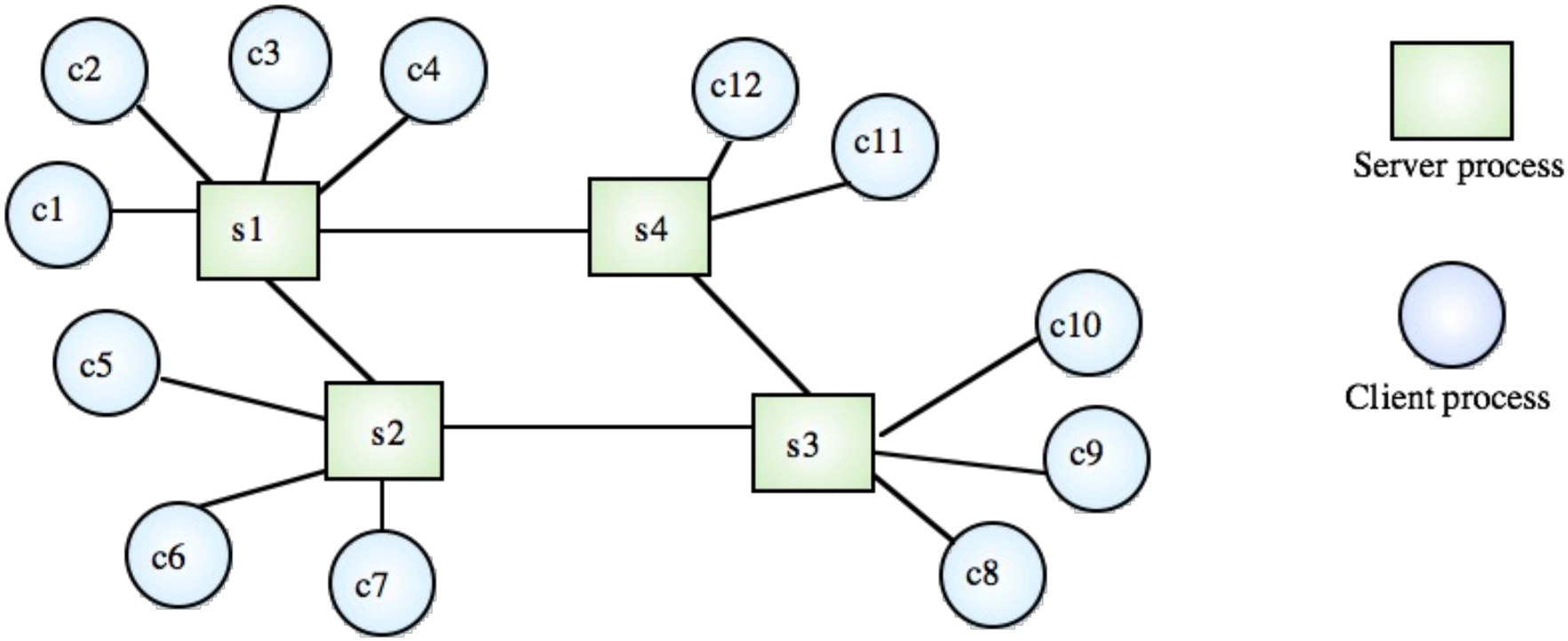
- Data and resource sharing
- Openness (ability to manage heterogeneous resources)
- Concurrency
- Scalability
- Load balancing
- Fault tolerance
- Transparency
- Adaptability to *enterprise computing* scenarios
- Critical factors
  - quality of service (performance, reliability, etc.)
  - interoperability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

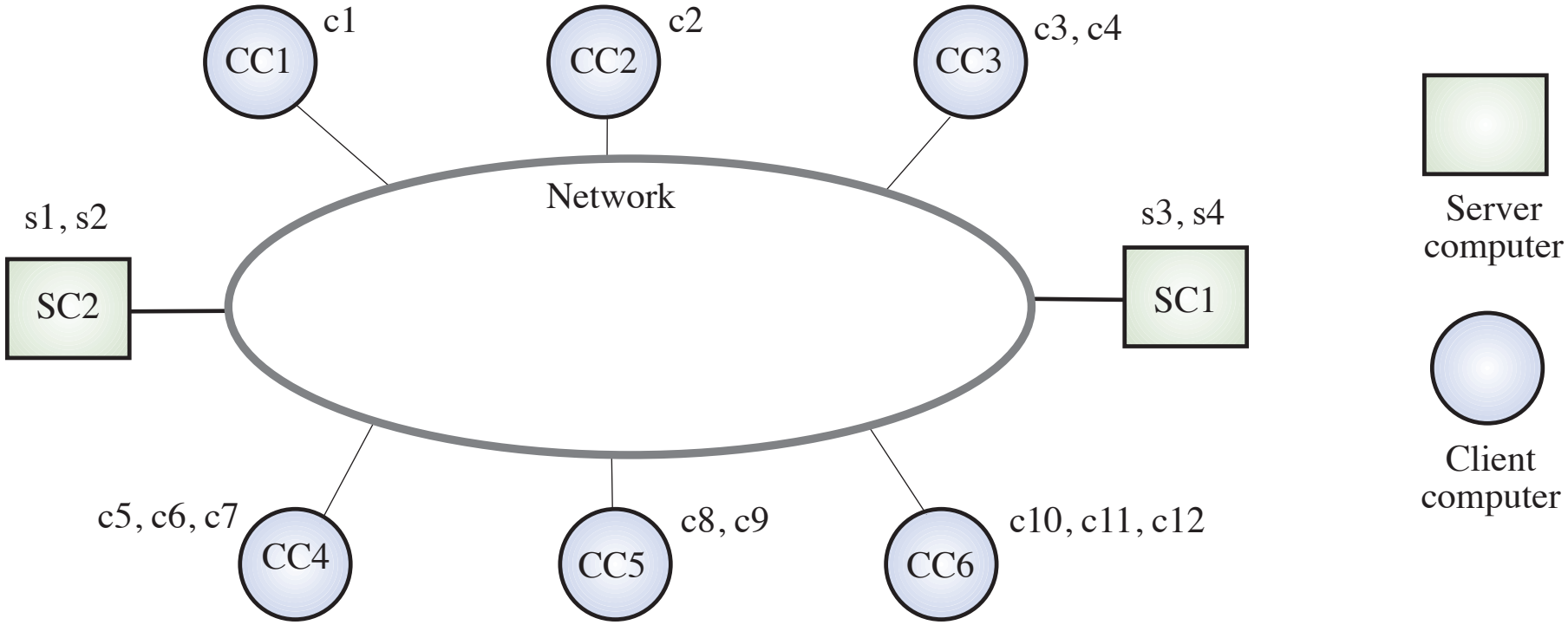
Client/server (C/S) architectures (2)

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



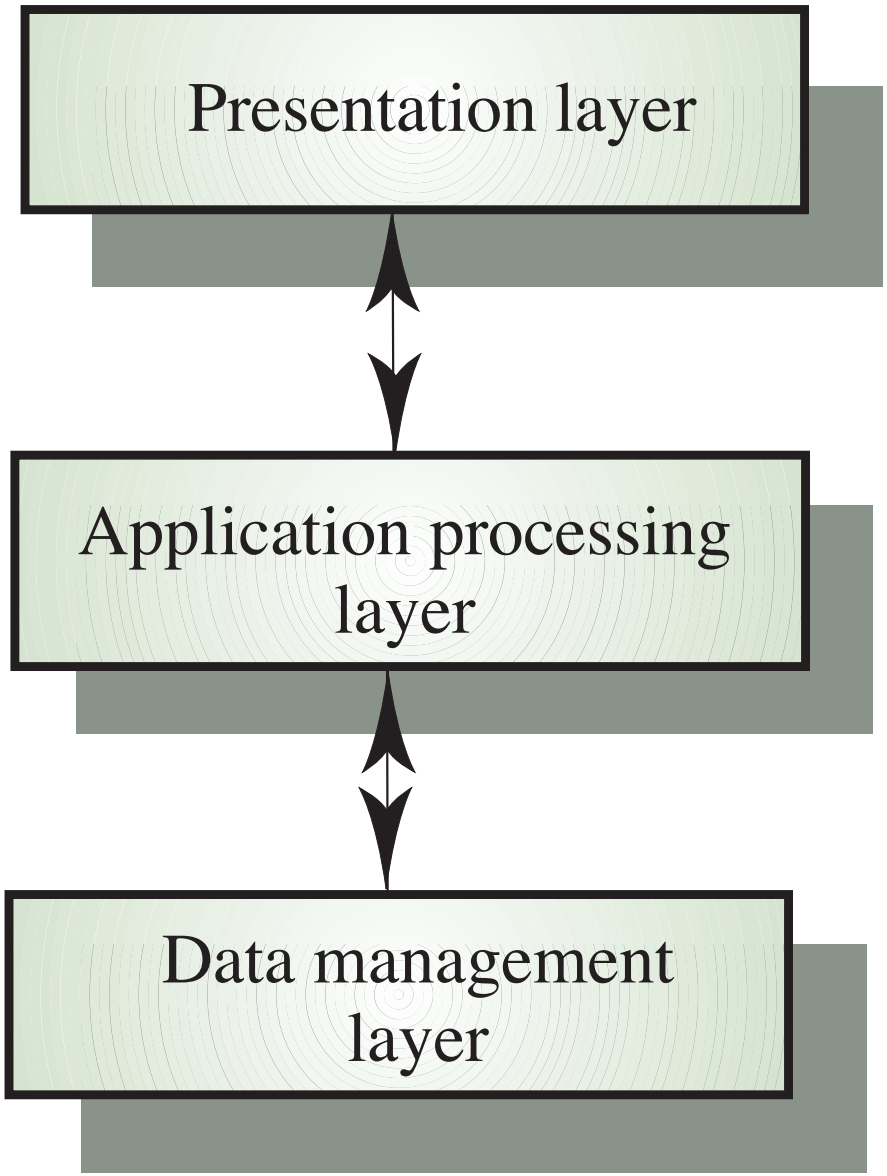
Client/server (C/S) architectures (3)

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



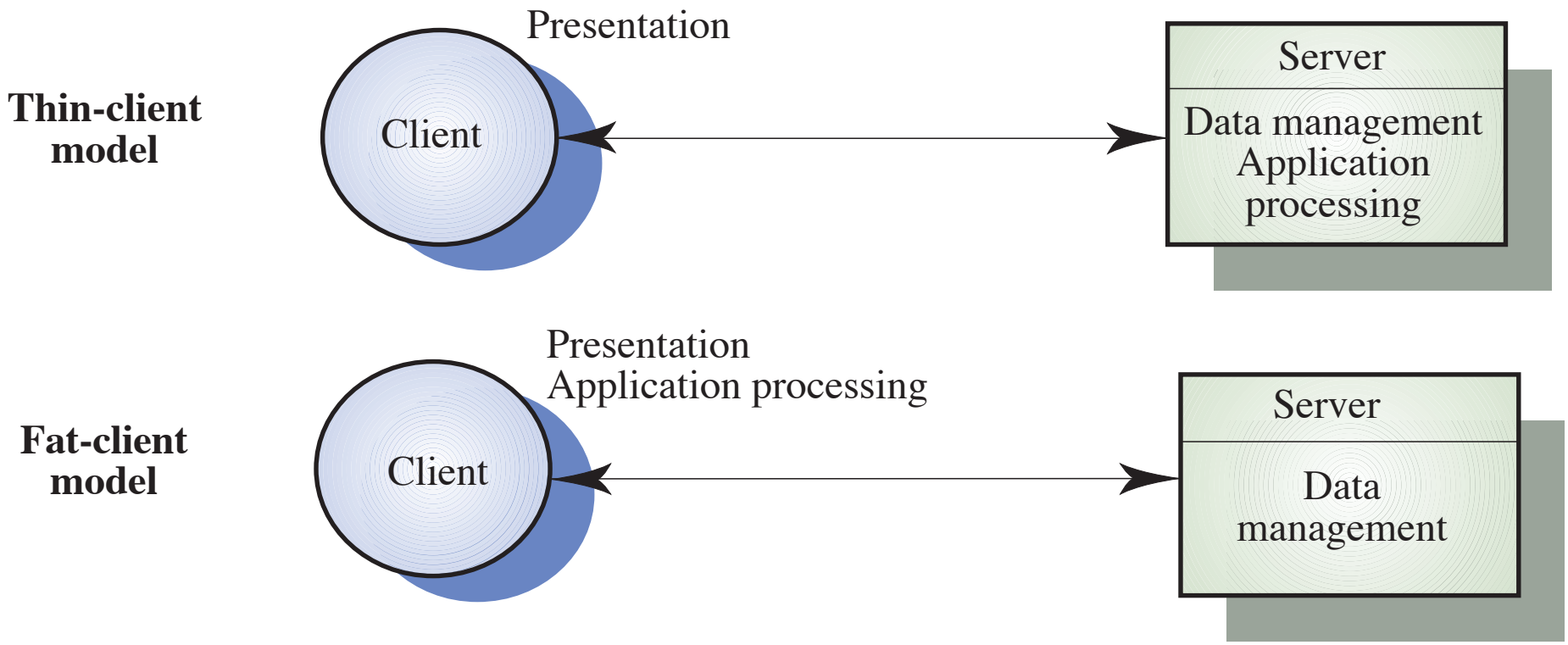
Application layers

- *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



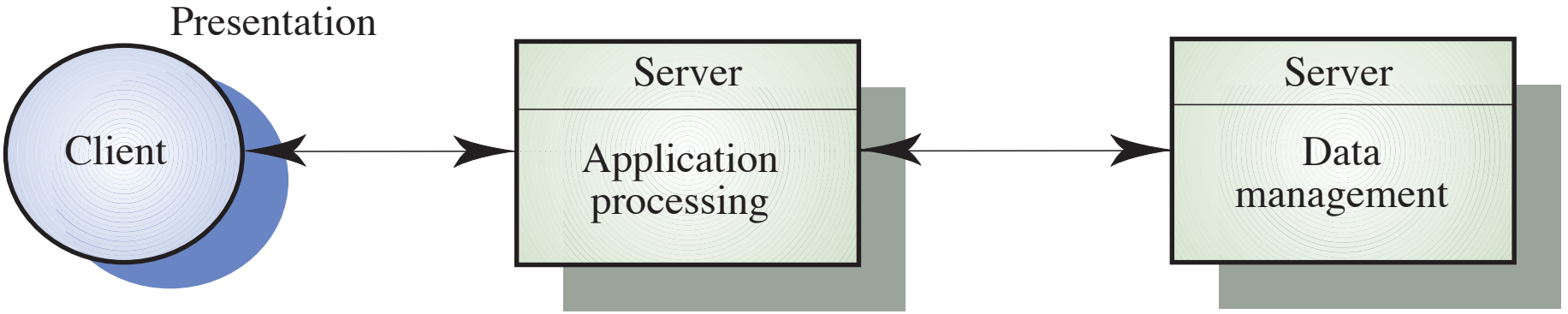
Two-tier C/S architectures

- **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.



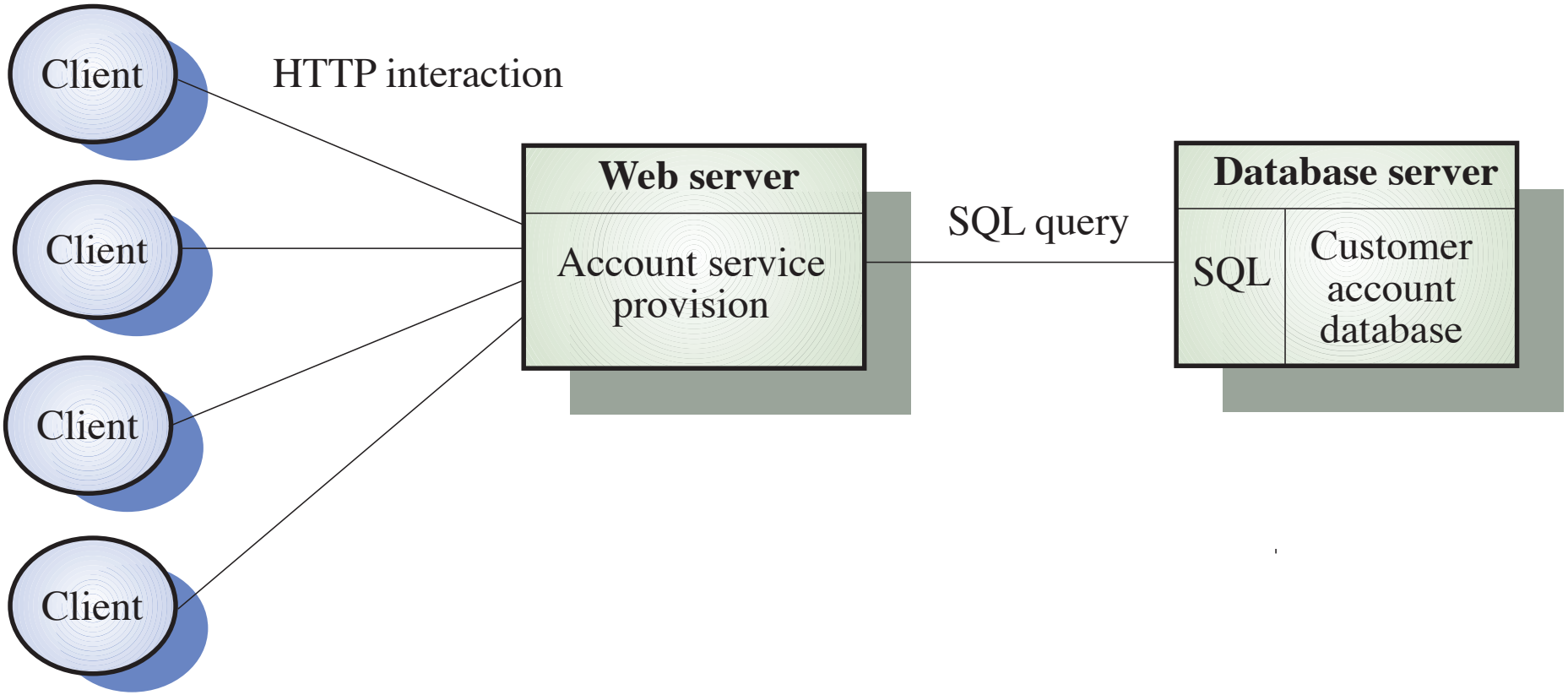
Three-tier C/S architectures

- 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

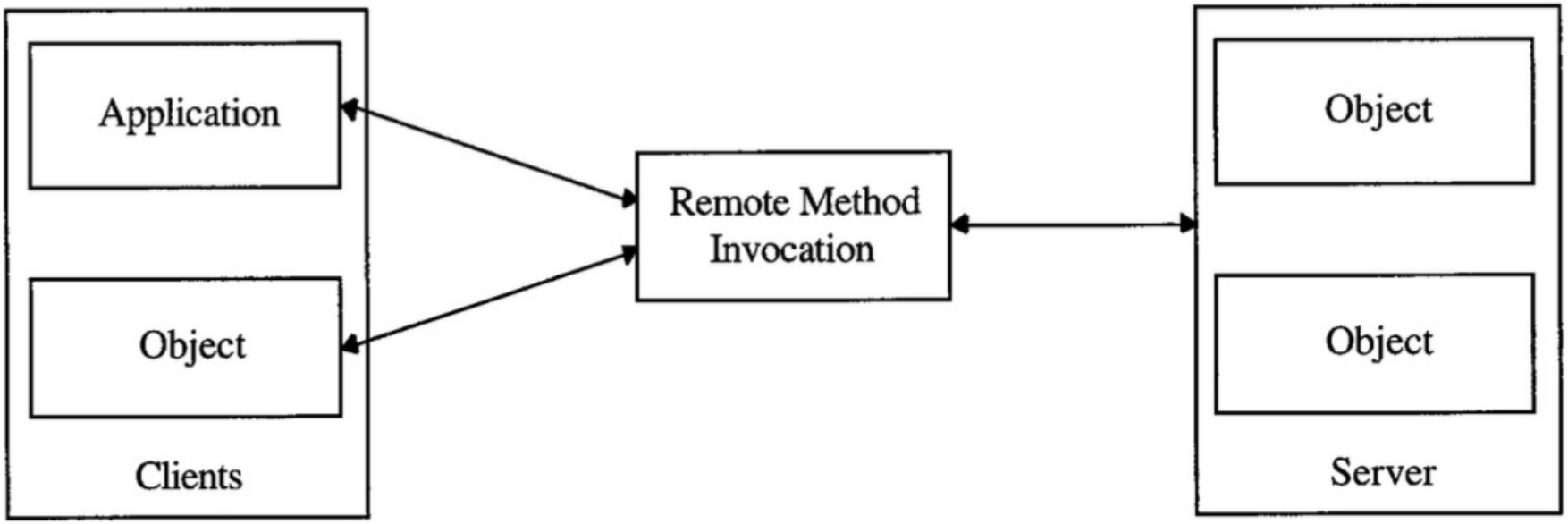
Internet Banking System



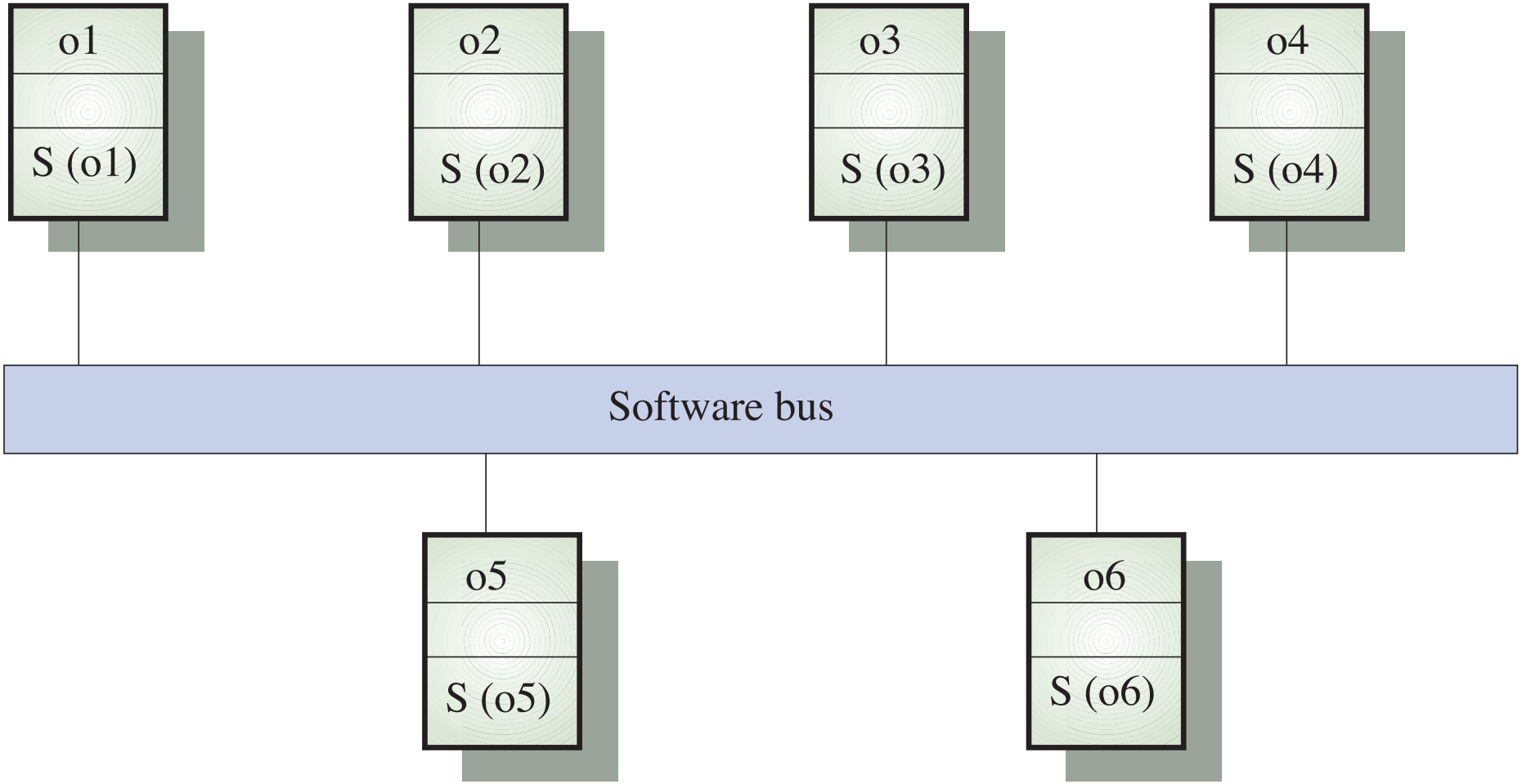
Distributed object architectures

- 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 (⇒ *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

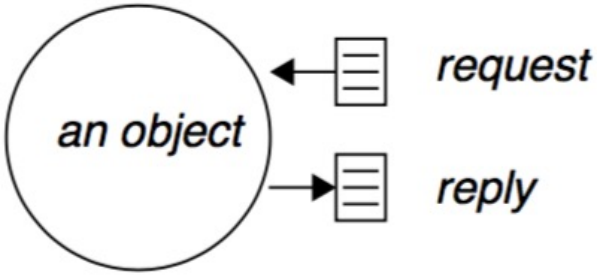
- ***Component-based architectures*** define software products assembled from a set of *software components*, which are designed to work together as part of a *component framework*
- A *component framework* makes use of generic software architectures to formalize given classes of applications
- *Component-based 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

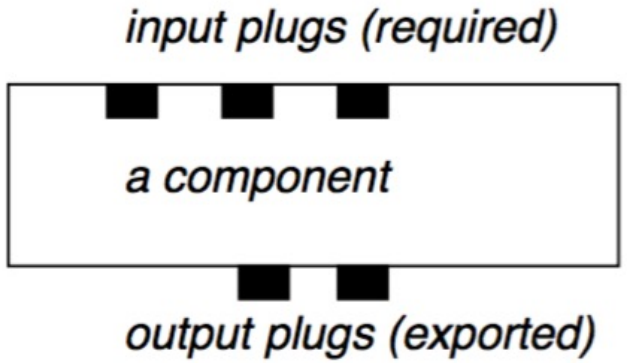
Component-based architectures (2)

- 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*)

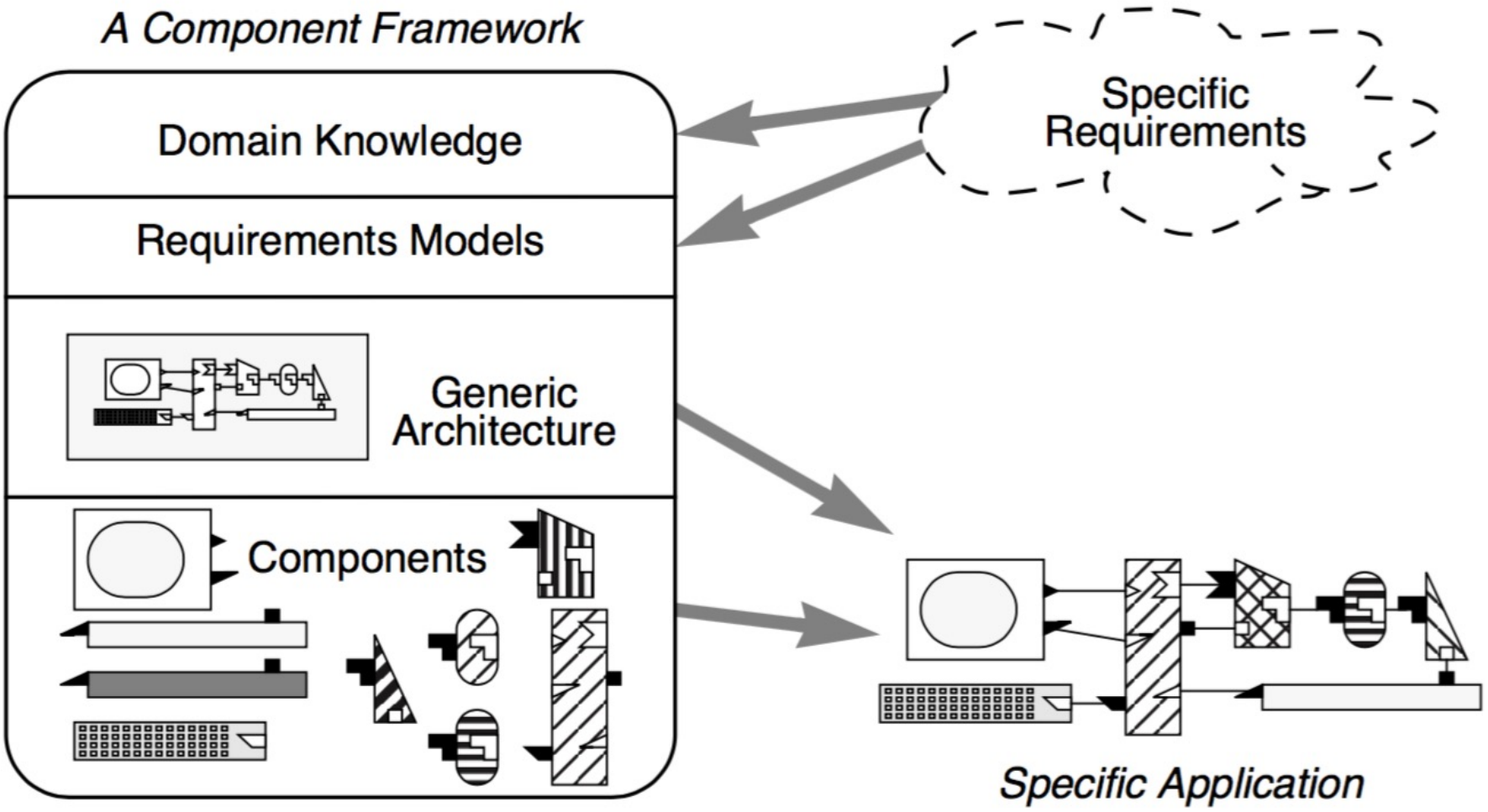
Component-based architectures (3)

- *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