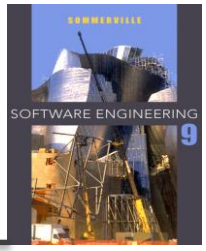
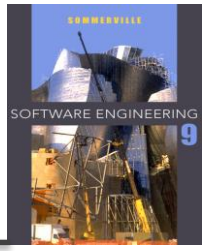


Chapter 6 – Architectural Design

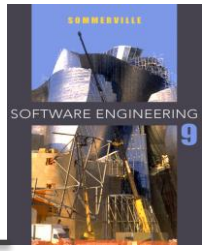
Lecture 1

Software architecture



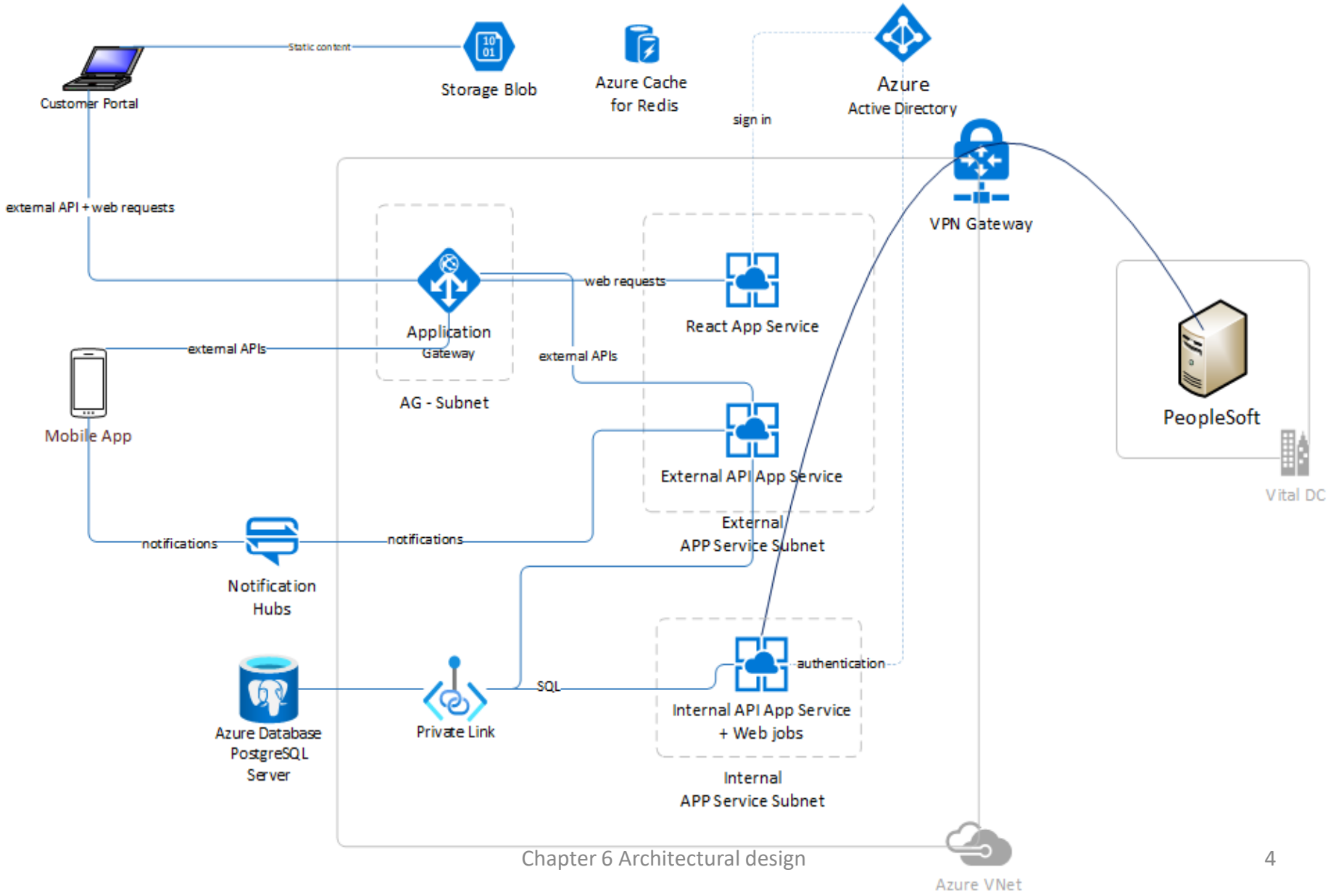
- The design process for identifying the sub-systems making up a system and the framework for sub-system control and communication is architectural design.
- The output of this design process is a description of the software architecture.

Architectural design

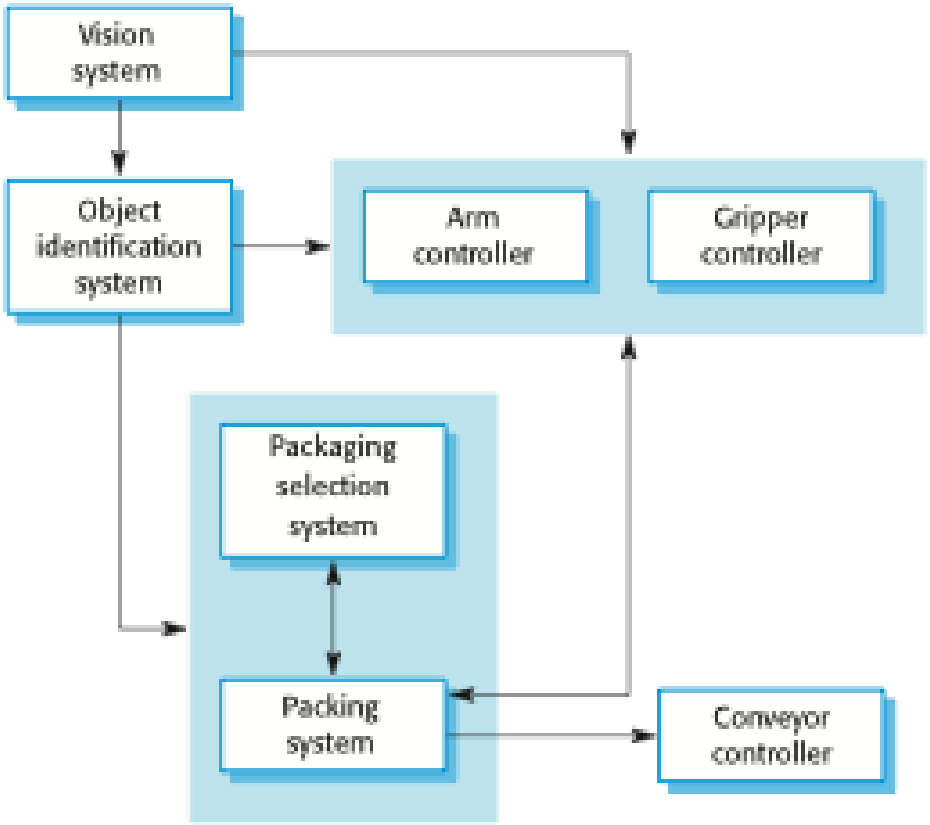


- An early stage of the system design process.
- Represents the link between specification and design processes.
- Often carried out in parallel with some specification activities.
- It involves identifying major system components and their communications.

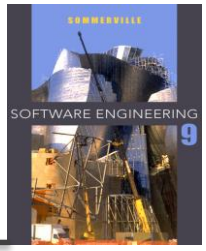
Vital Mobile App



The architecture of a packing robot control system

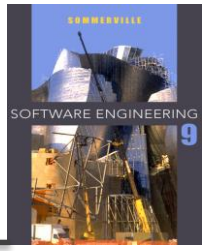


Architectural abstraction



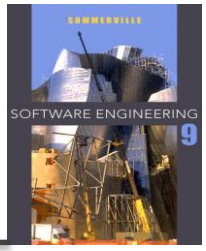
- Architecture in the small is concerned with the architecture of individual programs. At this level, we are concerned with the way that an individual program is decomposed into components.
- Architecture in the large is concerned with the architecture of complex enterprise systems that include other systems, programs, and program components. These enterprise systems are distributed over different computers, which may be owned and managed by different companies.

Advantages of explicit architecture



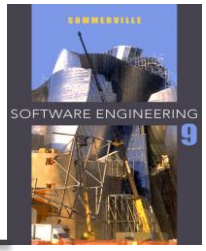
- ✧ Stakeholder communication
  - Architecture may be used as a focus of discussion by system stakeholders.
- ✧ System analysis
  - Means that analysis of whether the system can meet its non-functional requirements is possible.
- ✧ Large-scale reuse
  - The architecture may be reusable across a range of systems
  - Product-line architectures may be developed.

Architectural representations



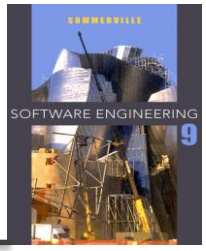
- Simple, informal block diagrams showing entities and relationships are the most frequently used method for documenting software architectures.
- But these have been criticised because they lack semantics, do not show the types of relationships between entities nor the visible properties of entities in the architecture.
- Depends on the use of architectural models.The requirements for model semantics depends on how the models are used.

Box and line diagrams



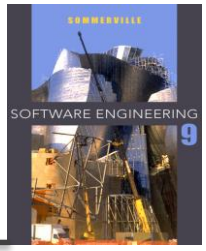
- Very abstract - they do not show the nature of component relationships nor the externally visible properties of the sub-systems.
- However, useful for communication with stakeholders and for project planning.

Use of architectural models



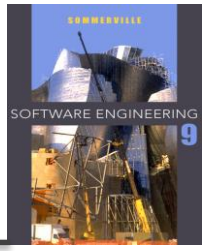
- As a way of facilitating discussion about the system design
  - A high-level architectural view of a system is useful for communication with system stakeholders and project planning because it is not cluttered with detail. Stakeholders can relate to it and understand an abstract view of the system. They can then discuss the system as a whole without being confused by detail.
- As a way of documenting an architecture that has been designed
  - The aim here is to produce a complete system model that shows the different components in a system, their interfaces and their connections.

Architectural design decisions



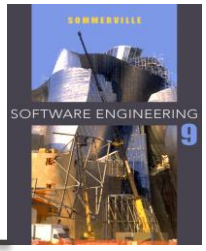
- Architectural design is a creative process so the process differs depending on the type of system being developed.
- However, a number of common decisions span all design processes and these decisions affect the non-functional characteristics of the system.

Architecture and system characteristics



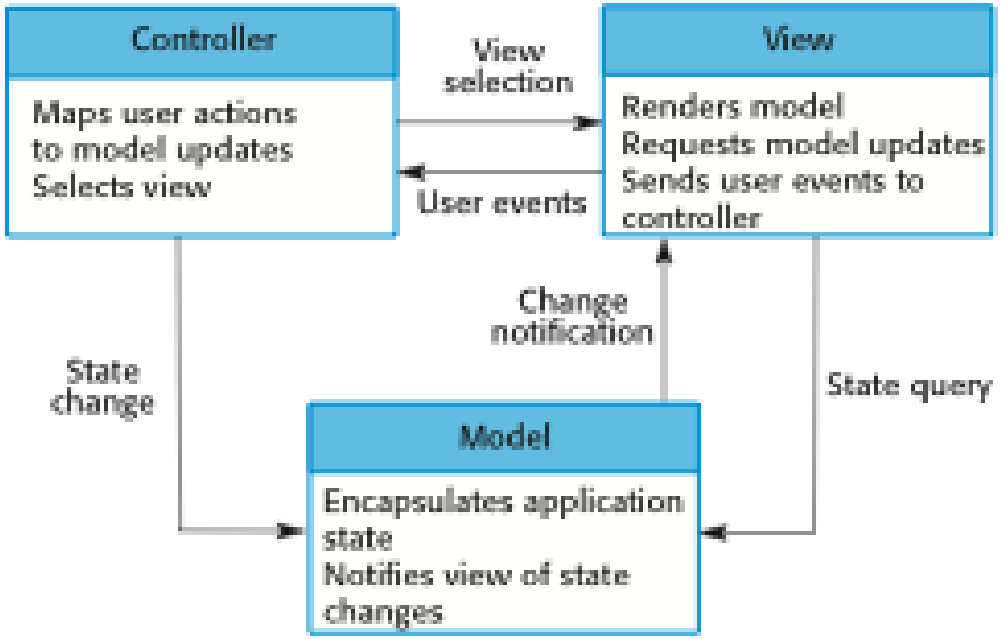
- Performance
  - Localise critical operations and minimise communications. Use large rather than fine-grain components.
- Security
  - Use a layered architecture with critical assets in the inner layers.
- Safety
  - Localise safety-critical features in a small number of sub-systems.
- Availability
  - Include redundant components and mechanisms for fault tolerance.
- Maintainability
  - Use fine-grain, replaceable components.

The Model-View-Controller (MVC) pattern

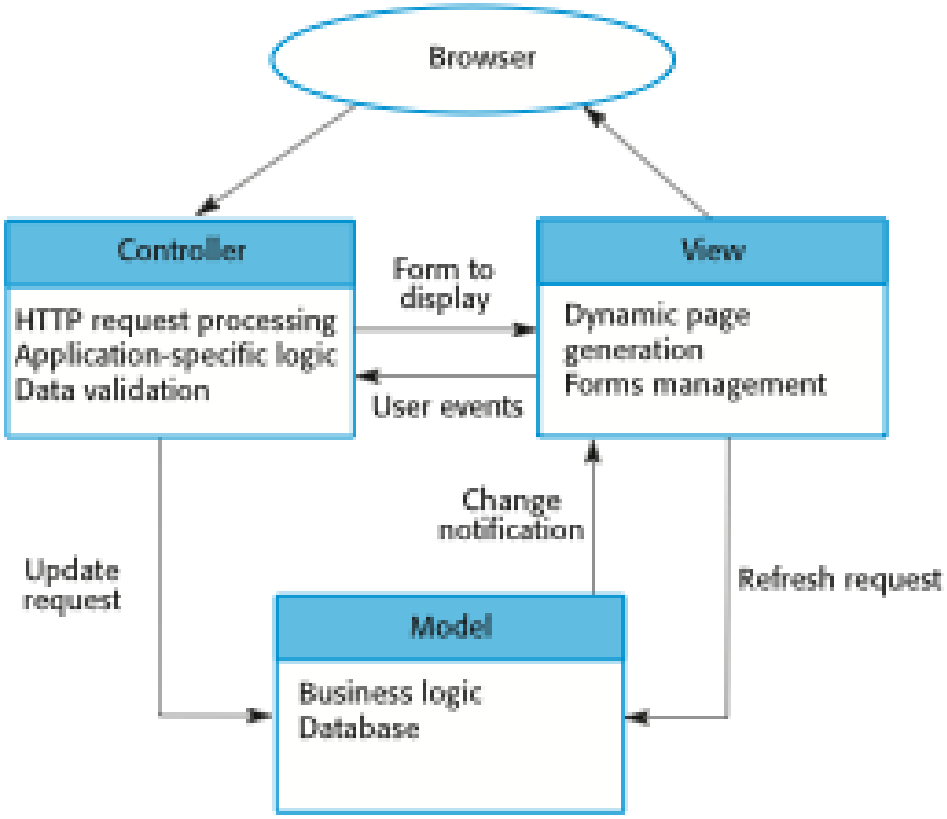


Name	MVC (Model-View-Controller)
Description	Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines and manages how the data is presented to the user. The Controller component manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model. See Figure 6.3.
Example	Figure 6.4 shows the architecture of a web-based application system organized using the MVC pattern.
When used	Used when there are multiple ways to view and interact with data. Also used when the future requirements for interaction and presentation of data are unknown.
Advantages	Allows the data to change independently of its representation and vice versa. Supports presentation of the same data in different ways with changes made in one representation shown in all of them.
Disadvantages	Can involve additional code and code complexity when the data model and interactions are simple.

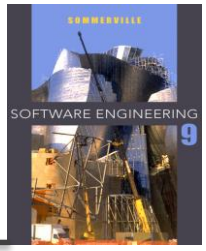
The organization of the Model-View-Controller



Web application architecture using the MVC pattern

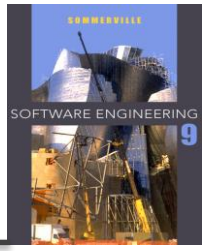


Layered architecture



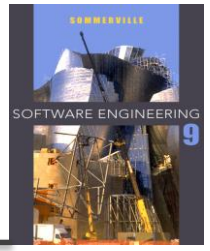
- Used to model the interfacing of sub-systems.
- Organises the system into a set of layers (or abstract machines) each of which provide a set of services.
- Supports the incremental development of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected.
- However, often artificial to structure systems in this way.

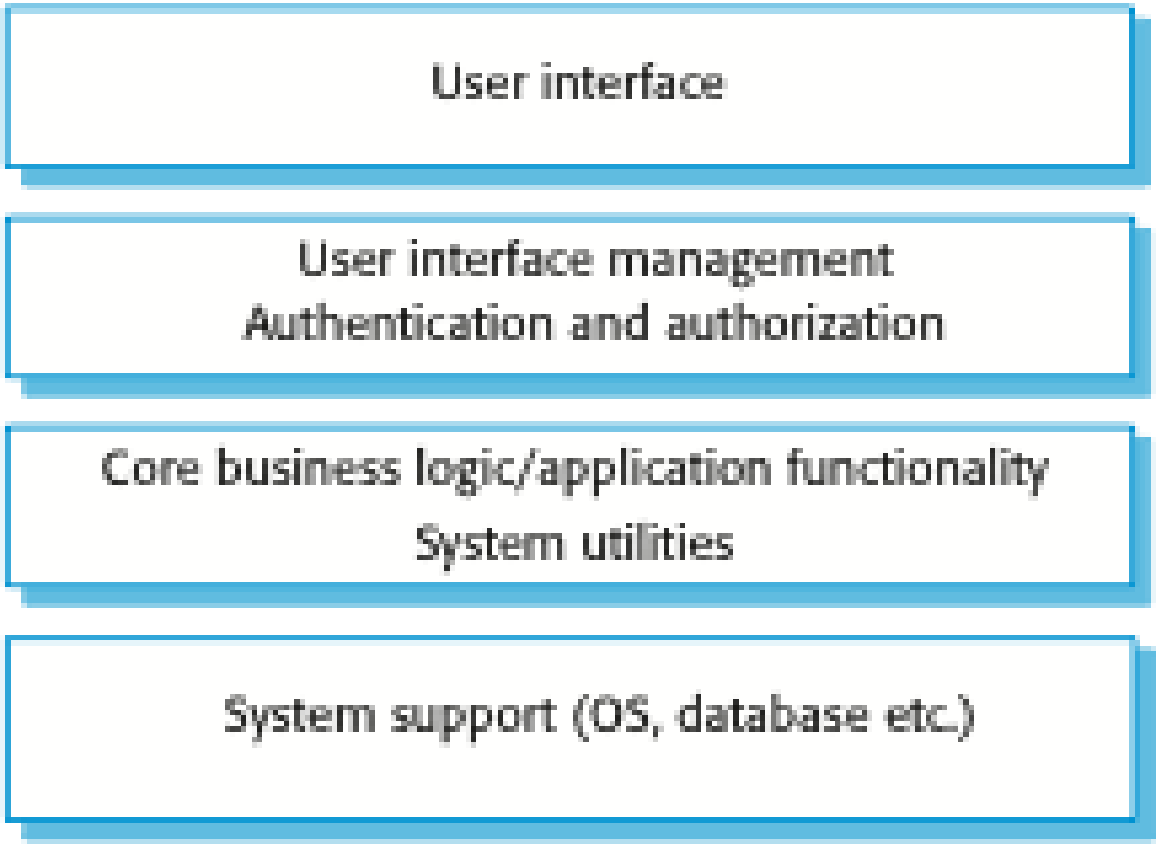
The Layered architecture pattern



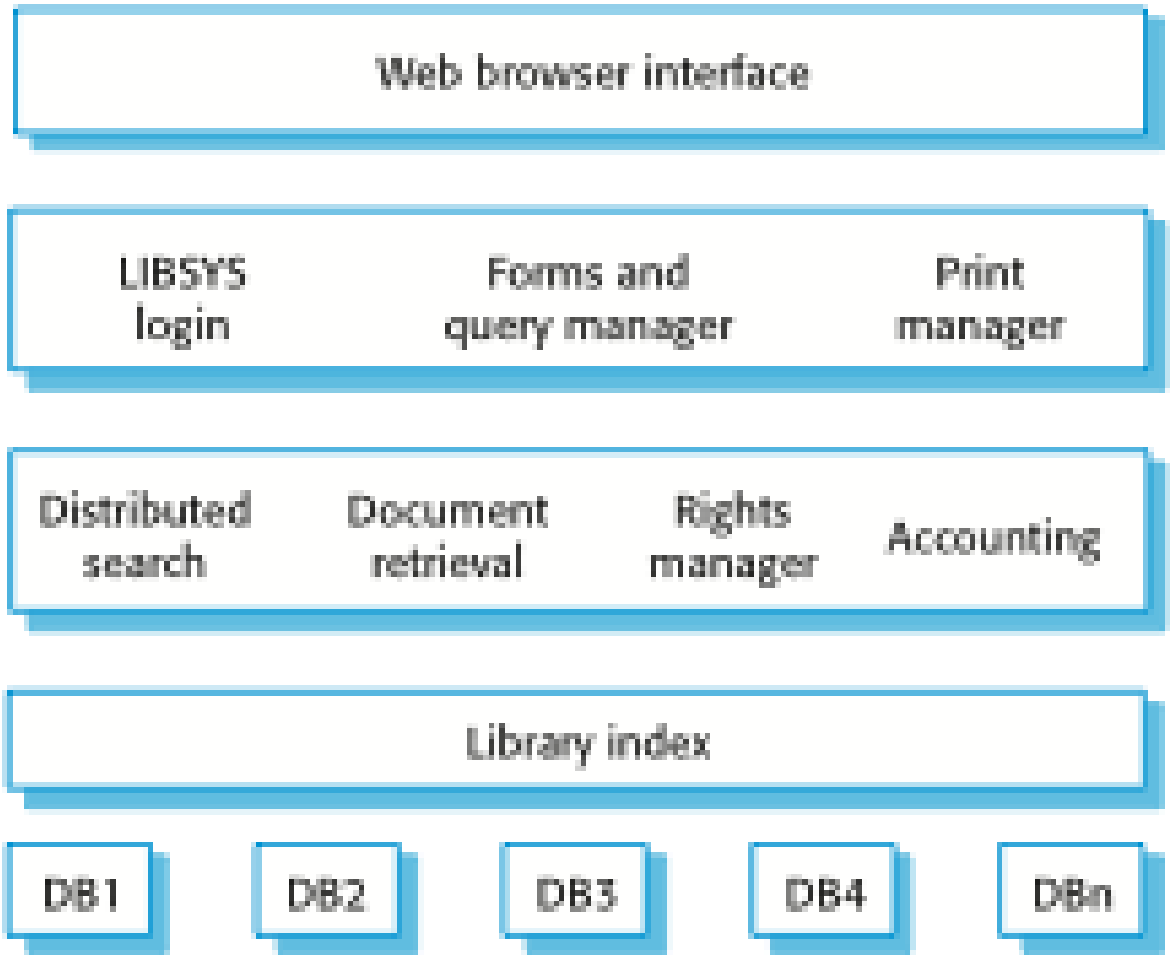
Name	Layered architecture
Description	Organizes the system into layers with related functionality associated with each layer. A layer provides services to the layer above it so the lowest-level layers represent core services that are likely to be used throughout the system. See Figure 6.6.
Example	A layered model of a system for sharing copyright documents held in different libraries, as shown in Figure 6.7.
When used	Used when building new facilities on top of existing systems; when the development is spread across several teams with each team responsibility for a layer of functionality; when there is a requirement for multi-level security.
Advantages	Allows replacement of entire layers so long as the interface is maintained. Redundant facilities (e.g., authentication) can be provided in each layer to increase the dependability of the system.
Disadvantages	In practice, providing a clean separation between layers is often difficult and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it. Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.

A generic layered architecture

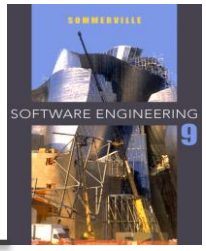




The architecture of the LIBSYS system

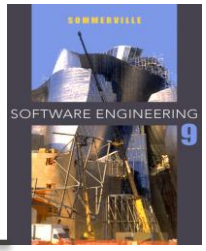


Client-server architecture



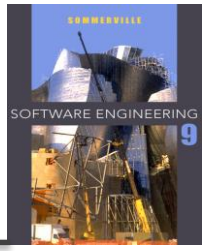
- Distributed system model which shows how data and processing is distributed across a range of components.
  - Can be implemented on a single computer.
- Set of stand-alone servers which provide specific services such as printing, data management, etc.
- Set of clients which call on these services.
- Network which allows clients to access servers.

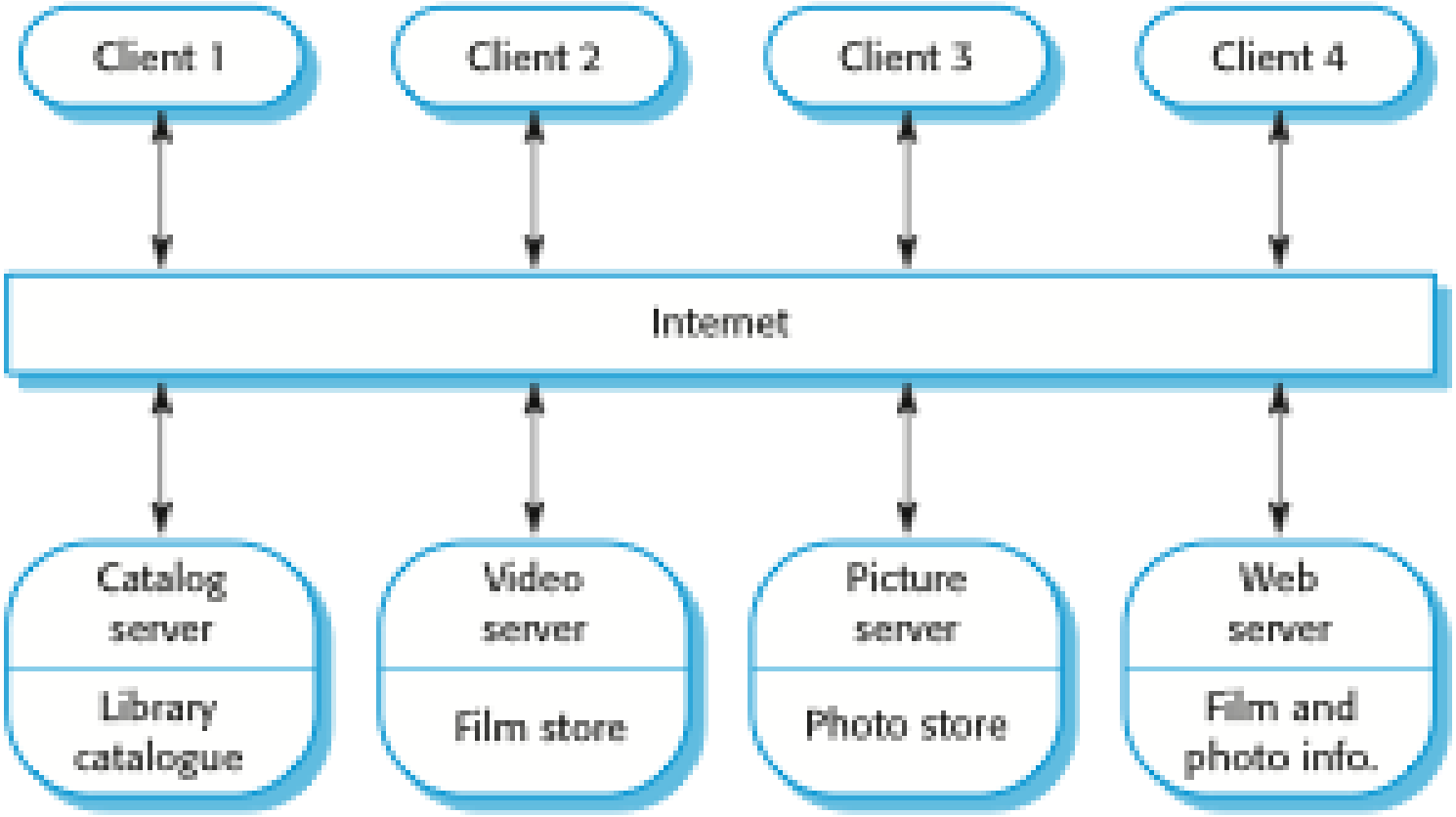
The Client–server pattern

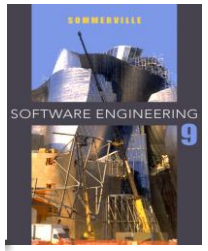


Name	Client-server
Description	In a client–server architecture, the functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.
Example	Figure 6.11 is an example of a film and video/DVD library organized as a client–server system.
When used	Used when data in a shared database has to be accessed from a range of locations. Because servers can be replicated, may also be used when the load on a system is variable.
Advantages	The principal advantage of this model is that servers can be distributed across a network. General functionality (e.g., a printing service) can be available to all clients and does not need to be implemented by all services.
Disadvantages	Each service is a single point of failure so susceptible to denial of service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system. May be management problems if servers are owned by different organizations.

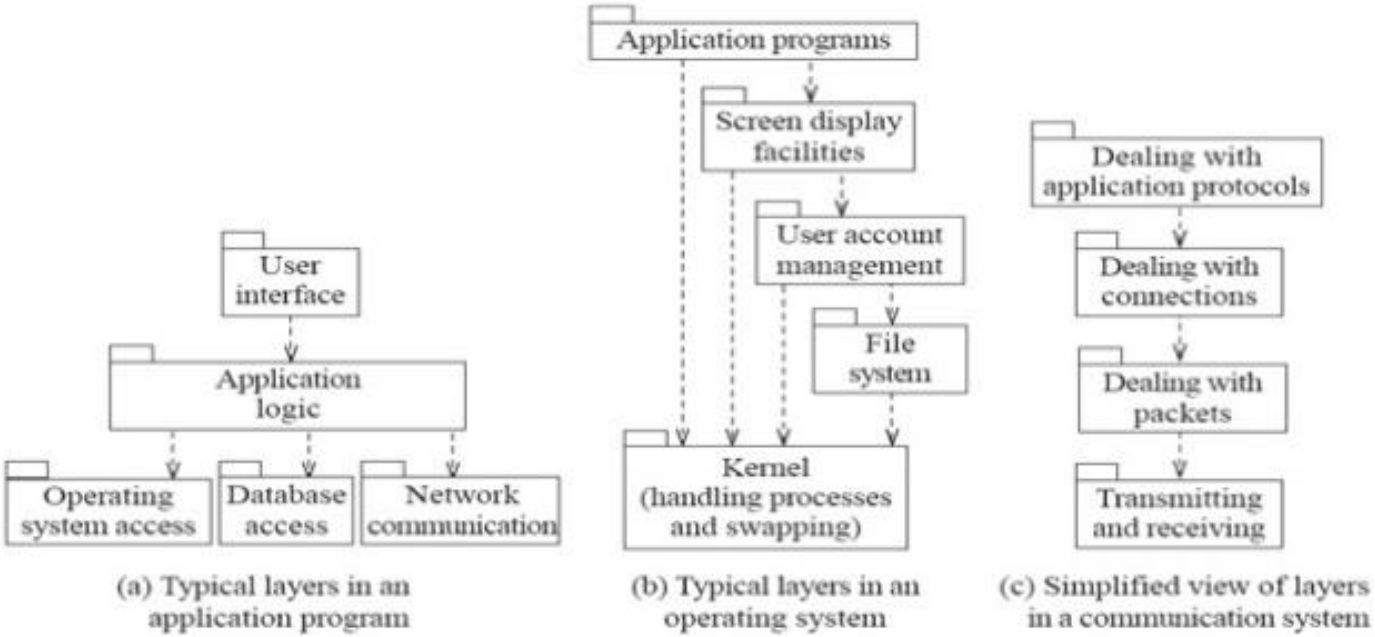
A client–server architecture for a film library







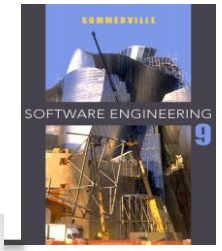
Example of multi-layer systems



(a) Typical layers in an application program

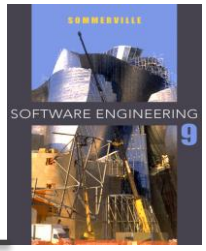
(b) Typical layers in an operating system

(c) Simplified view of layers in a communication system



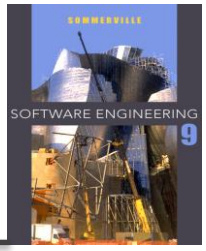
The multi-layer architecture and design principles

1. *Divide and conquer*: The layers can be independently designed.
2. *Increase cohesion*: Well-designed layers have layer cohesion.
3. *Reduce coupling*: Well-designed lower layers do not know about the higher layers and the only connection between layers is through the API.
4. *Increase abstraction*: you do not need to know the details of how the lower layers are implemented.
5. *Increase reusability*: The lower layers can often be designed generically.



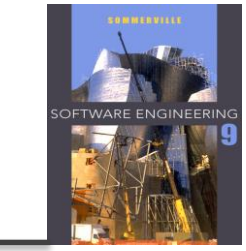
The multi-layer architecture and design principles

6. *Increase reuse*: You can often reuse layers built by others that provide the services you need.
7. Increase flexibility: you can add new facilities built on lower-level services, or replace higher-level layers.
8. *Anticipate obsolescence*: By isolating components in separate layers, the system becomes more resistant to obsolescence.
9. *Design for portability*: All the dependent facilities can be isolated in one of the lower layers.
10. *Design for testability*: Layers can be tested independently.
11. *Design defensively*: The APIs of layers are natural places to build in rigorous assertion-checking.

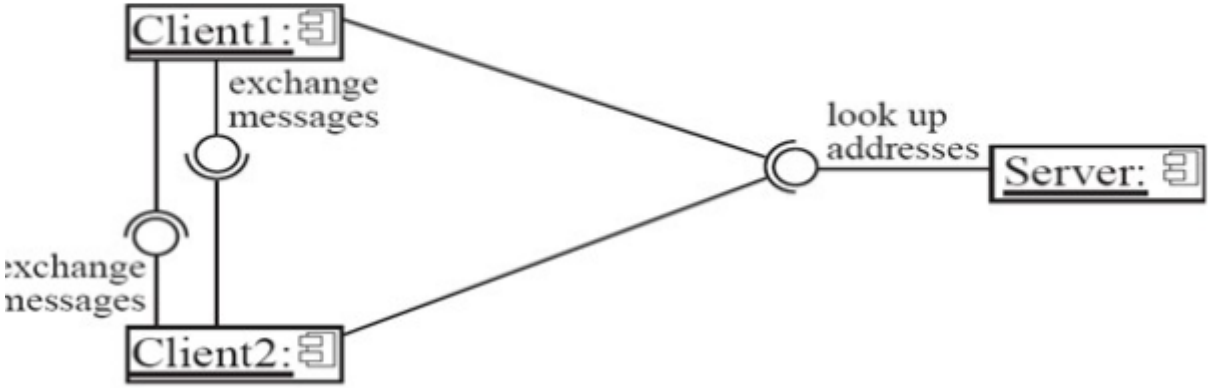


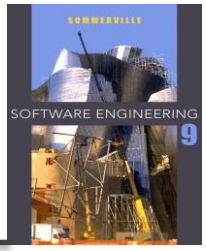
The Client-Server and other distributed architectural patterns

- There is at least one component that has the role of *server*, waiting for and then handling connections.
- There is at least one component that has the role of *client*, initiating connections in order to obtain some service.
- A further extension is the Peer-to-Peer pattern.
  - A system composed of various software components that are distributed over several hosts.



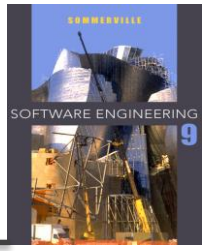
An example of a distributed system





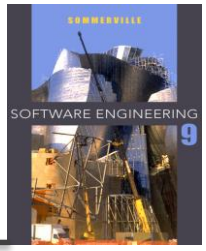
The distributed architecture and design principles

1. *Divide and conquer*: Dividing the system into client and server processes is a strong way to divide the system.
   - Each can be separately developed.
2. *Increase cohesion*: The server can provide a cohesive service to clients.
3. *Reduce coupling*: There is usually only one communication channel exchanging simple messages.
4. *Increase abstraction*: Separate distributed components are often good abstractions.
6. *Increase reuse*: It is often possible to find suitable frameworks on which to build good distributed systems
   - However, client-server systems are often very application specific.



The distributed architecture and design principles

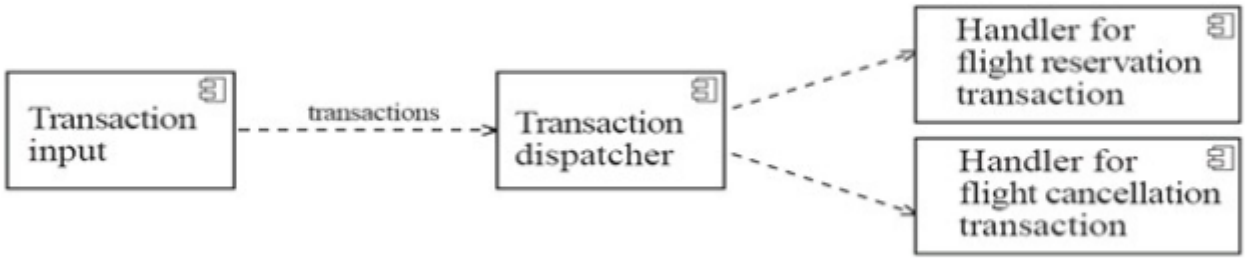
7. *Design for flexibility*: Distributed systems can often be easily reconfigured by adding extra servers or clients.
9. *Design for portability*: You can write clients for new platforms without having to port the server.
10 *Design for testability*: You can test clients and servers independently.
11. *Design defensively*: You can put rigorous checks in the message handling code.

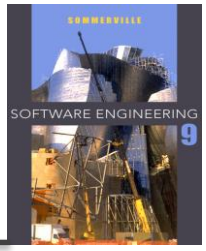


The Transaction-Processing architectural pattern

- A process reads a series of inputs one by one.
  - Each input describes a *transaction* – a command that typically some change to the data stored by the system
  - There is a transaction *dispatcher* component that decides what to do with each transaction
  - This dispatches a procedure call or message to one of a series of component that will *handle* the transaction

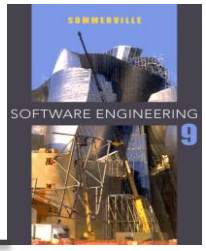
Example of a transaction-processing system





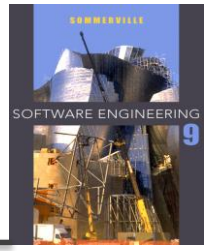
The transaction-processing architecture and design principles

1. *Divide and conquer*: The transaction handlers are suitable system divisions that you can give to separate software engineers.
2. *Increase cohesion*: Transaction handlers are naturally cohesive units.
3. *Reduce coupling*: Separating the dispatcher from the handlers tends to reduce coupling.
7. *Design for flexibility*: You can readily add new transaction handlers.
11. *Design defensively*: You can add assertion checking in each transaction handler and/or in the dispatcher.

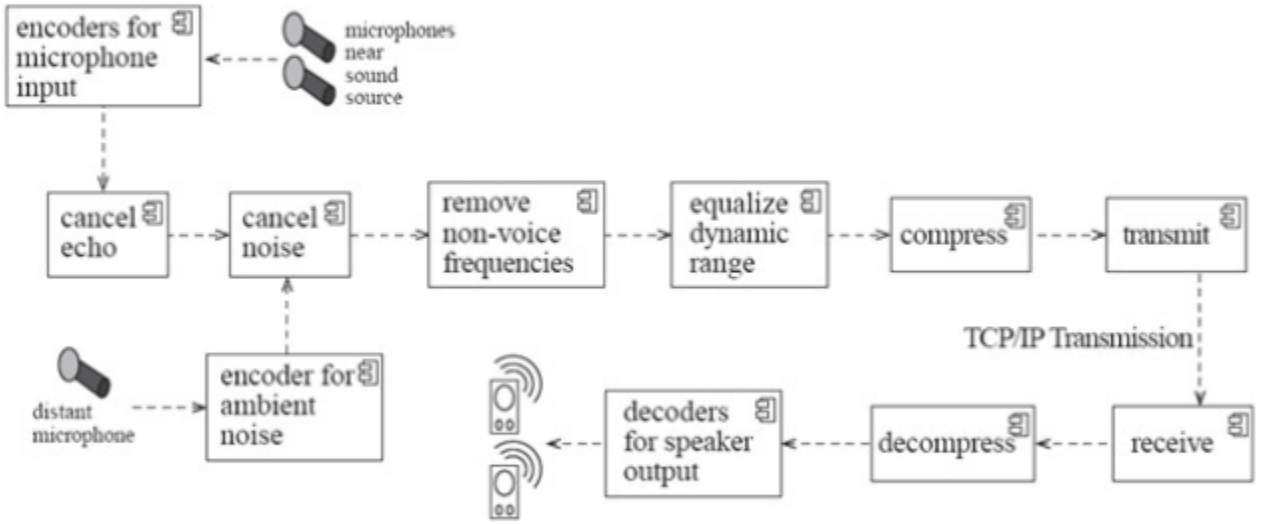


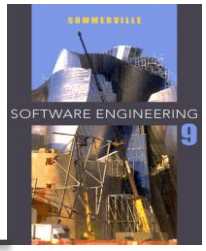
The Pipe-and-Filter architectural pattern

- A stream of data, in a relatively simple format, is passed through a series of processes
  - Each of which transforms it in some way.
  - Data is constantly fed into the pipeline.
  - The processes work concurrently.
  - The architecture is very flexible.
    - Almost all the components could be removed.
    - Components could be replaced.
    - New components could be inserted.
    - Certain components could be reordered.



Example of a pipe-and-filter system





The Service-oriented architectural pattern

- This architecture organizes an application as a collection of services that communicates using well-defined interfaces
  - In the context of the Internet, the services are called *Web services*
  - A web service is an application, accessible through the Internet, that can be integrated with other services to form a complete system
  - The different components generally communicate with each other using open standards such as XML.