# Architectural Design

- Chapter 6, Software Engineering 9th Edition by Ian Sommerville
- https://msdn.microsoft.com/en-us/library/ee658098.aspx
- https://msdn.microsoft.com/en-us/library/ee658117.aspx
- https://msdn.microsoft.com/en-us/library/ee658124.aspx
- https://msdn.microsoft.com/en-us/library/ee658084.aspx

Software application architecture is **the process of defining a structured solution that meets all of the technical and operational requirements, while optimizing common quality attributes such as performance, security, and manageability**.

It involves a series of decisions based on a wide range of factors, and each of these decisions can have considerable impact on the quality, performance, maintainability, and overall success of the application.

https://msdn.microsoft.com/en-us/library/ee658098.aspx

"Software architecture **encompasses the set of significant decisions about the organization of a software system including the selection of the structural elements and their interfaces by which the system is composed; behavior as specified in collaboration among those elements; composition of these structural and behavioral elements into larger subsystems; and an architectural style that guides this organization**.

Software architecture also involves functionality, usability, resilience, performance, reuse, comprehensibility, economic and technology constraints, tradeoffs and aesthetic concerns."

- *Philippe Kruchten, Grady Booch, Kurt Bittner, and Rich Reitman derived and refined a definition of architecture based on work by Mary Shaw and David Garlan (Shaw and Garlan 1996).*

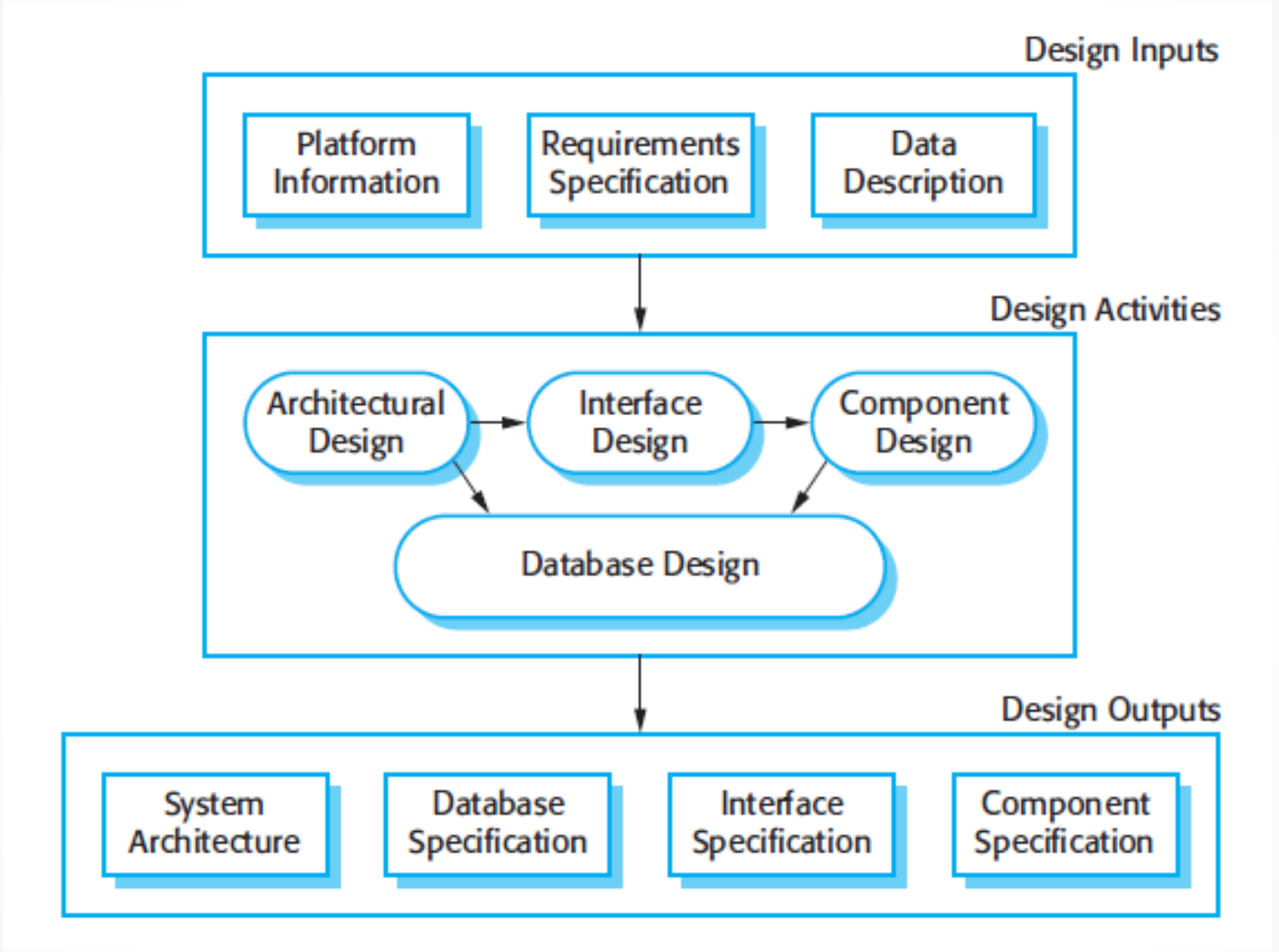
"The software architecture of a program or computing system is **the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.**

Architecture is concerned with the public side of interfaces; private details of elements—details having to do solely with internal implementation—are not architectural."

- *Software Architecture in Practice (2nd edition), Bass, Clements, and Kazman*

- Architectural design is concerned with *understanding how a system should be organized* and *designing the overall structure of that system*
- **Software architecture** is a description of how a software system is organized
- Properties of a system such as performance, security, and availability are influenced by the architecture used.

*- Software Engineering 9th Edition, Ian Sommerville*


Design Inputs
Platform Information
Requirements Specification
Data Description
Design Activities
Architectural Design
Interface Design
Component Design
Database Design
Design Outputs
System Architecture
Database Specification
Interface Specification
Component Specification

# Why is Architecture Important?

- Like any other complex structure, software must be built on a solid foundation

Things that can put your application at risk:

- Failing to consider key scenarios
- Failing to design for common problems
- Failing to appreciate the long term consequences of key decisions

The risks exposed by poor architecture include software that is

- unstable
- unable to support existing or future business requirements
- difficult to deploy or manage in a production environment

# The Architectural Landscape (1)

## key forces that are shaping architectural decisions today

- **User empowerment**
  - A design that supports user empowerment is flexible, configurable, and focused on the user experience
  - Design your application with appropriate levels of user personalization and options in mind
  - Allow the user to define how they interact with your application instead of dictating to them, but do not overload them with unnecessary options and settings that can lead to confusion
  - Understand the key scenarios and make them as simple as possible; make it easy to find information and use the application
- **Market maturity**
  - Take advantage of market maturity by taking advantage of existing platform and technology options
  - Build on higher level application frameworks where it makes sense, so that you can focus on what is uniquely valuable in your application rather than recreating something that already exists and can be reused
  - Use patterns that provide rich sources of proven solutions for common problems

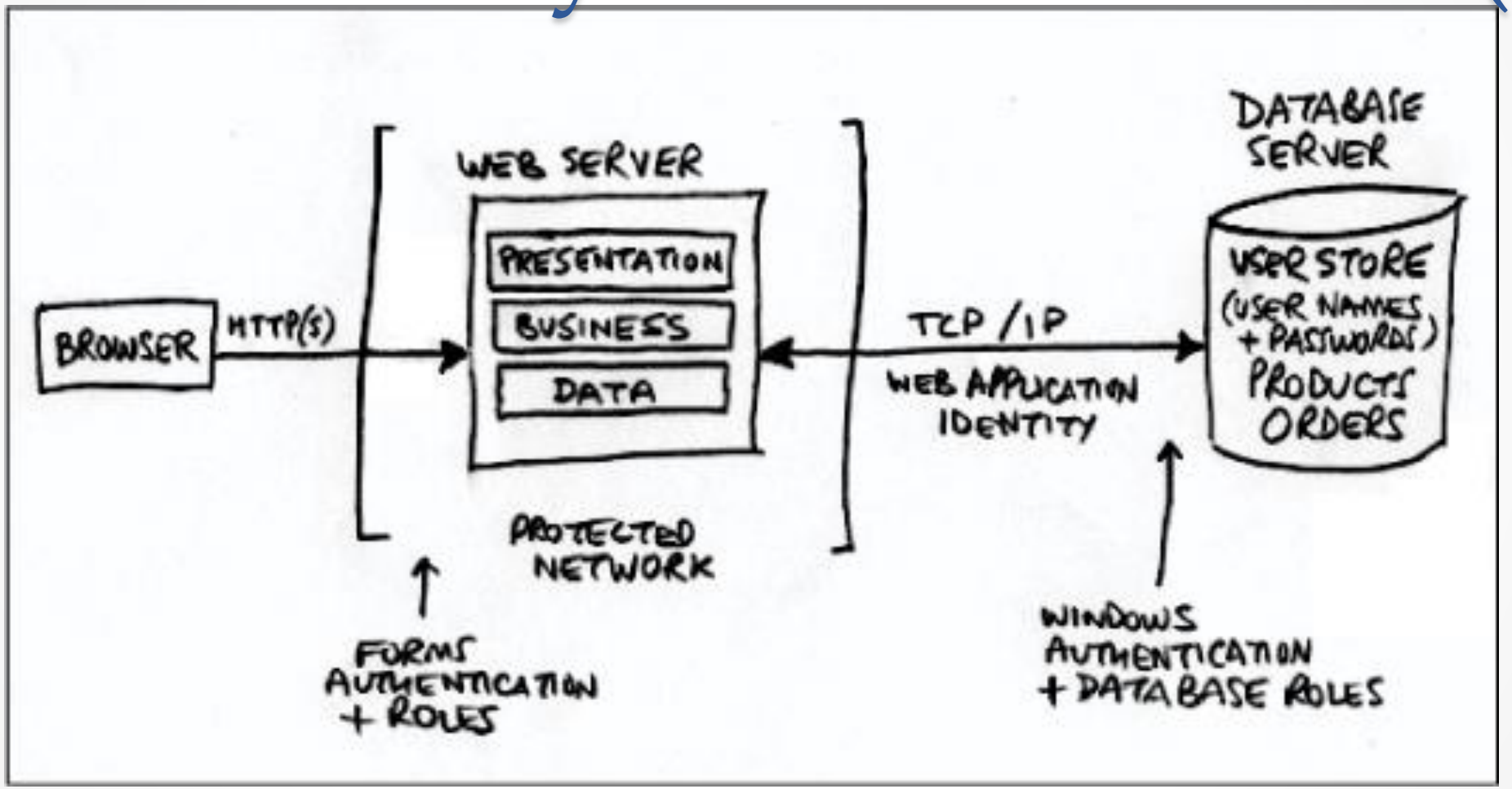
# The Architectural Landscape (2)

- **Flexible design**
  - Increasingly, flexible designs take advantage of loose coupling to allow reuse and to improve maintainability
  - Pluggable designs allow you to provide post-deployment extensibility
  - Take advantage of service orientation techniques such as SOA to provide interoperability with other systems
- **Future trends**
  - When building your architecture, understand the future trends that might affect your design after deployment
  - Consider trends in rich UI and media, composition models such as
    - Mashups
    - increasing network bandwidth and availability
    - increasing use of mobile devices
    - continued improvement in hardware performance
    - interest in community and personal publishing models
    - the rise of cloud-based computing
    - remote operation

# Key Architecture Principles

- **Build to change instead of building to last**
  - Consider how the application may need to change over time to address new requirements and challenges, and build in the flexibility to support this
- **Model to analyze and reduce risk**
  - Use design tools, modeling systems such as Unified Modeling Language (UML), and visualizations where appropriate to help you capture requirements and architectural and design decisions, and to analyze their impact
- **Use models and visualizations as a communication and collaboration tool**
  - Use models, views, and other visualizations of the architecture to communicate and share your design efficiently with all the stakeholders, and to enable rapid communication of changes to the design
- **Identify key engineering decisions**
  - Understand the key engineering decisions and the areas where mistakes are most often made
  - Invest in getting these key decisions right the first time so that the design is more flexible and less likely to be broken by changes

# Whiteboard your architecture (1)

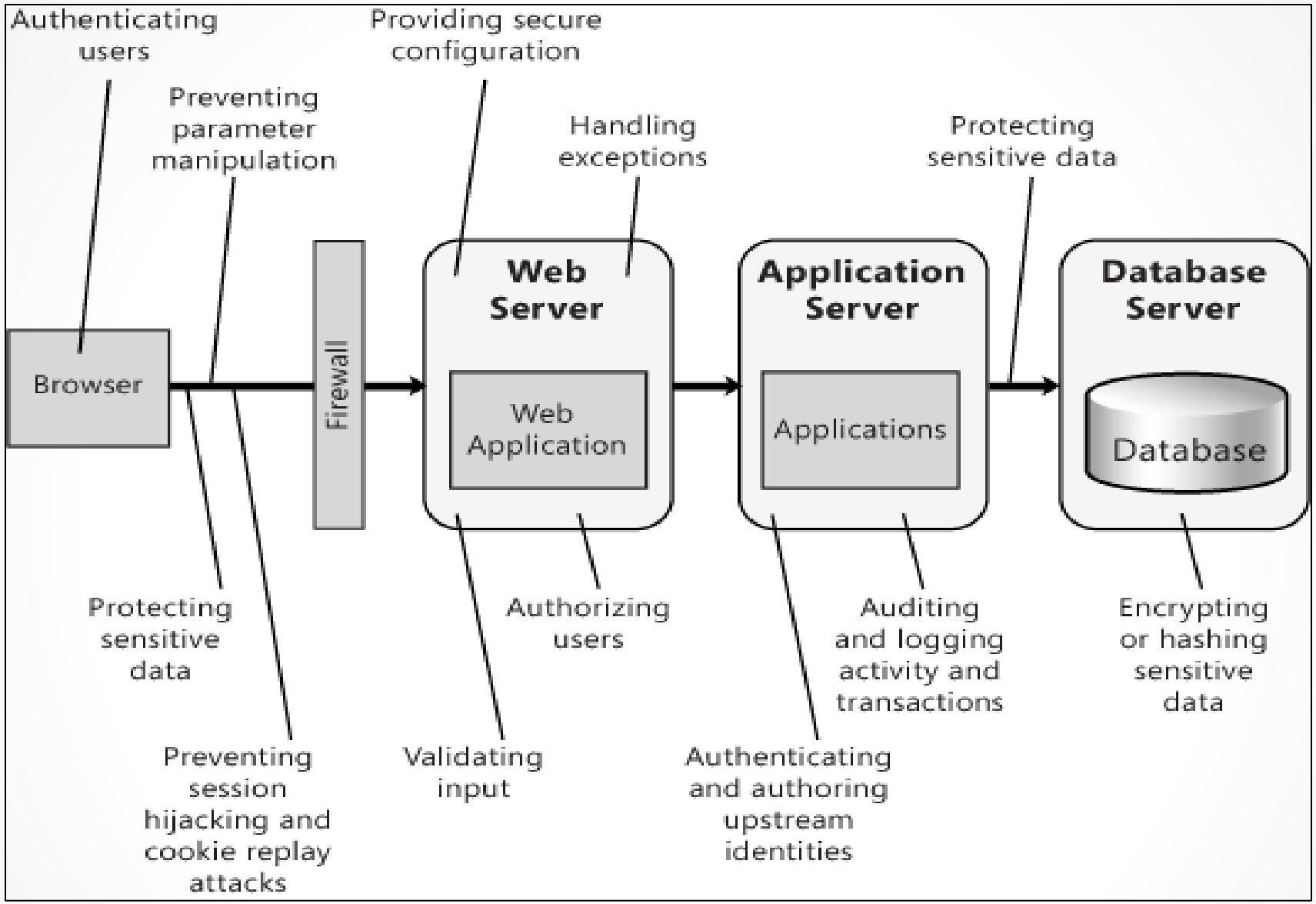



https://msdn.microsoft.com/en-us/library/ee658084.aspx

# Whiteboard your architecture (2)

- Whether you share your whiteboard on paper, slides, or through another format, *the key is to show the major constraints and decisions in order to frame and start conversations*.
- The value is actually twofold:

1) If you cannot whiteboard the architecture then it suggests that *it is not well understood*
2) If you can provide a clear and concise whiteboard diagram, *others will understand it* and you can communicate details to them more easily.




Security issues identified in a typical Web application architecture
https://msdn.microsoft.com/en-us/library/ee658084.aspx

# Architectural Views

# How many views should we make? (1)

- Architectures may be documented from several different perspectives or views
- It is impossible to represent all relevant information about a system's architecture in a single architectural model, as each model only shows one view or perspective of the system
  - It might show how a system is decomposed into modules, how the run-time processes interact, or the different ways in which system components are distributed across a network
- For both design and documentation, you usually need to **present multiple views** of the software architecture

# How many views should we make? (2)

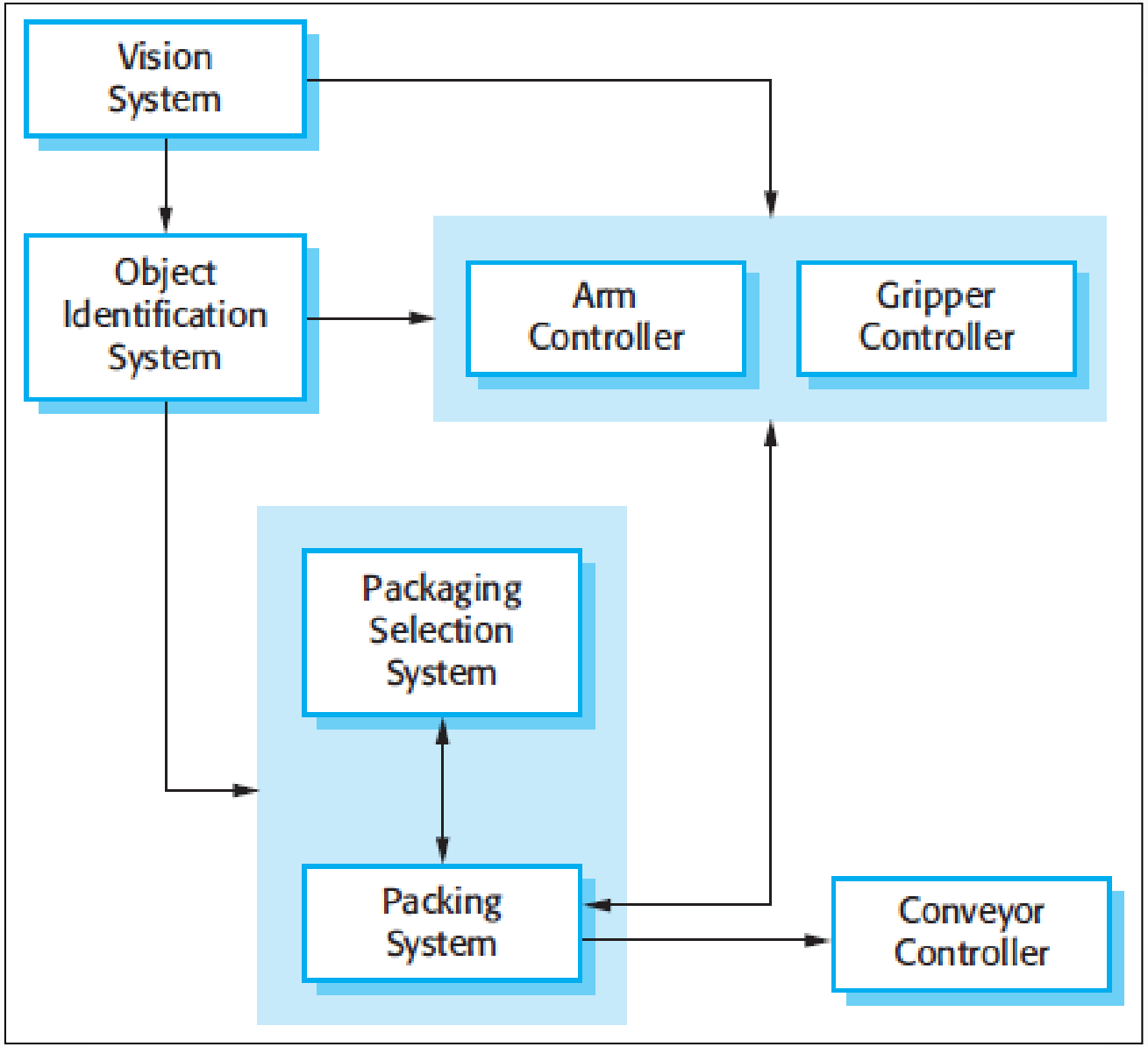
Krutchen (1995), in his well-known 4+1 view model of software architecture, suggests that there should be **four (4) fundamental architectural views**, which are related using use cases or scenarios.

- **A logical view**
  - shows the key abstractions in the system as objects or object classes
  - It should be possible to relate the system requirements to entities in this logical view
- **A process view**
  - shows how, at run-time, the system is composed of interacting processes
  - This view is useful for making judgments about nonfunctional system characteristics such as performance and availability

# How many views should we make? (3)

- **A development view**
  - shows how the software is decomposed for development
  - it shows the breakdown of the software into components that are implemented by a single developer or development team
  - This view is useful for software managers and programmers
- **A physical view**
  - shows the system hardware and how software components are distributed across the processors in the system
  - This view is useful for systems engineers planning a system deployment

Hofmeister et al. (2000) suggest the use of similar views but add to this the notion of a **conceptual view**.




A conceptual view – the architecture of a packing robot control system.

# Architectural Patterns

# Architectural Patterns

- a means of reusing knowledge about generic system architectures
- They describe the architecture, explain when it may be used, and discuss its advantages and disadvantages

Commonly used architectural patterns include:

- Model-View-Controller
- Layered Architecture
- Repository
- Client–server
- Pipe and Filter

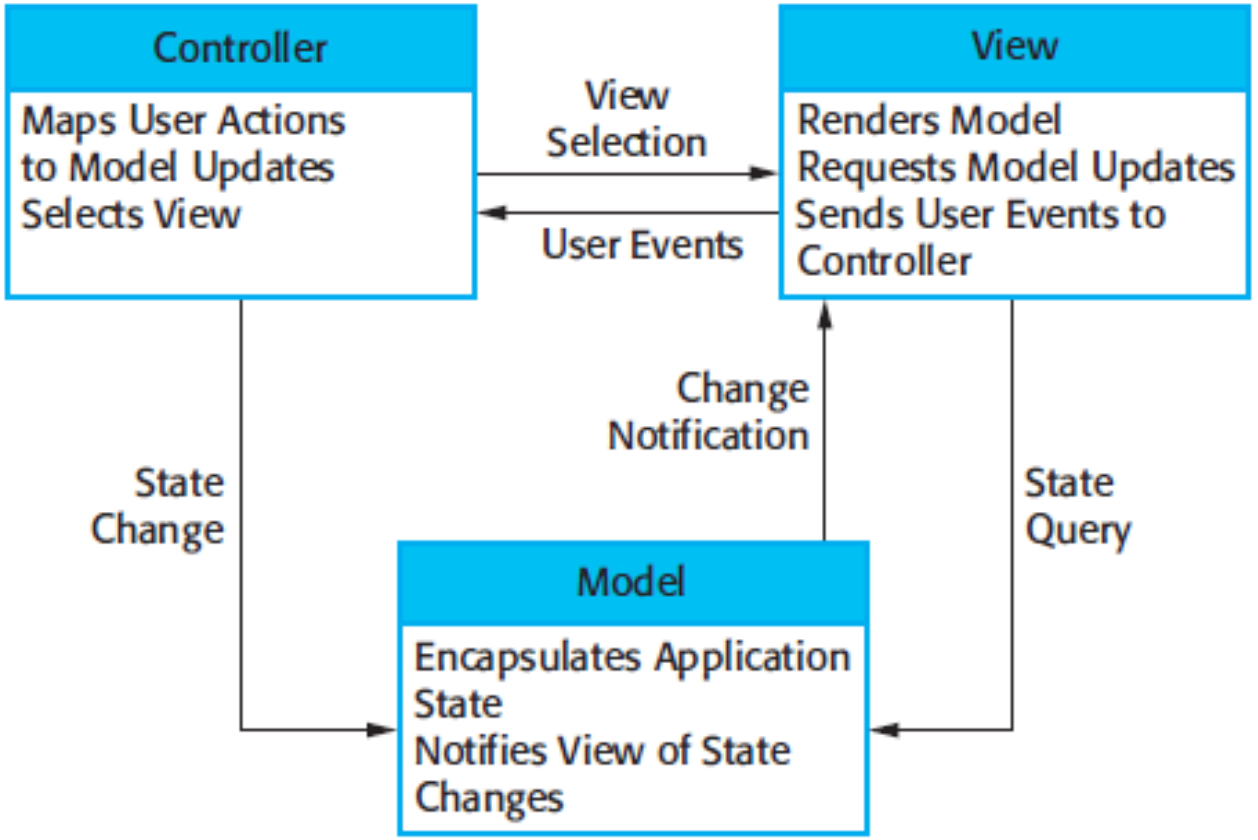
# Model-View-Controller (MVC) pattern (1)

The MVC pattern separates presentation and interaction from the system data.

The system is structured into three logical components that interact with each other:

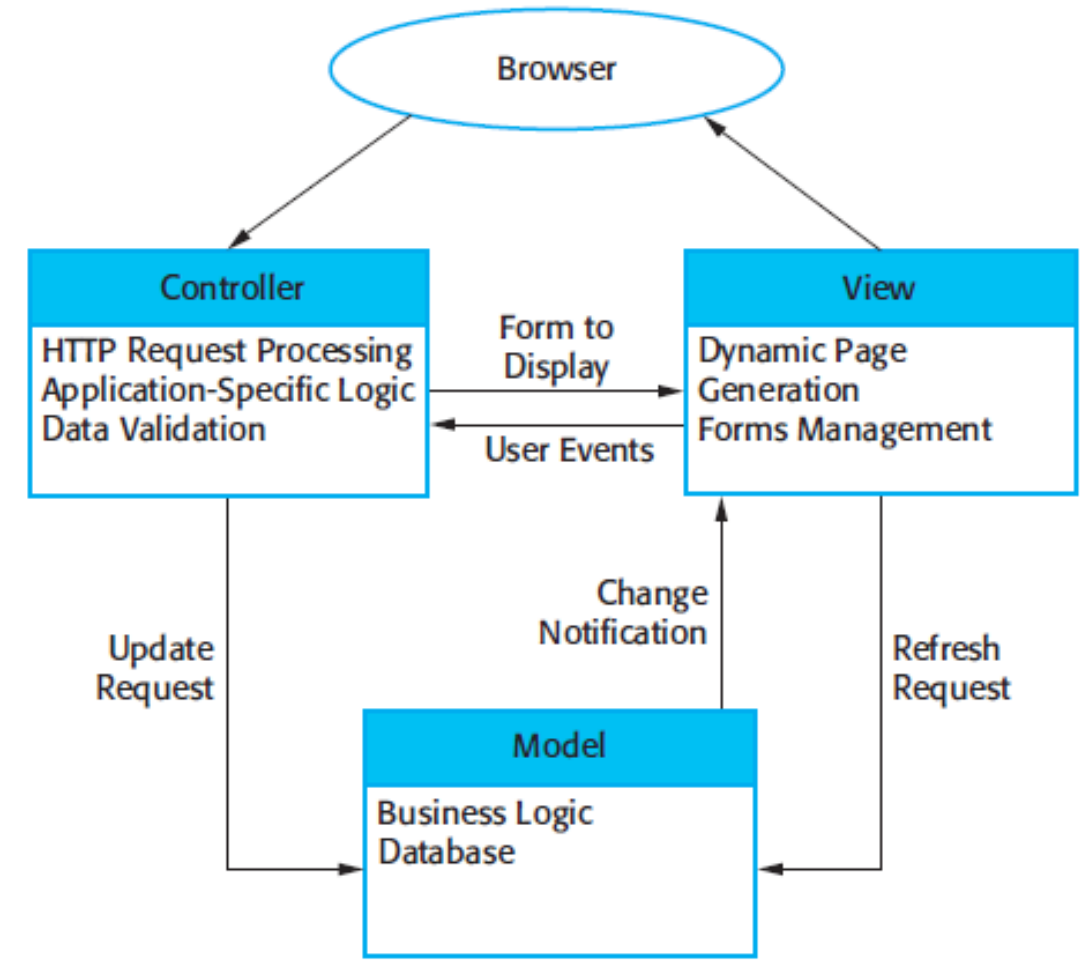
- The **Model component** manages the system data and associated operations on that data
- The **View component** defines and manages how the data is presented to the user
- The **Controller component** manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model

# Model-View-Controller (MVC) pattern (2)




The organization of the MVC

# Model-View-Controller (MVC) pattern (3)




Web application architecture using the MVC pattern

# Model-View-Controller (MVC) pattern (4)

- MVC is used when there are multiple ways to view and interact with data
- It is 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

*Disadvantage:*

- Can involve additional code and code complexity when the data model and interactions are simple

# Layered Architecture pattern (1)

- 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

The Layered Architecture is used

- when building new facilities on top of existing systems;
- when the development is spread across several teams with each team responsible for a layer of functionality
- when there is a requirement for multi-level security

# Layered Architecture pattern (2)

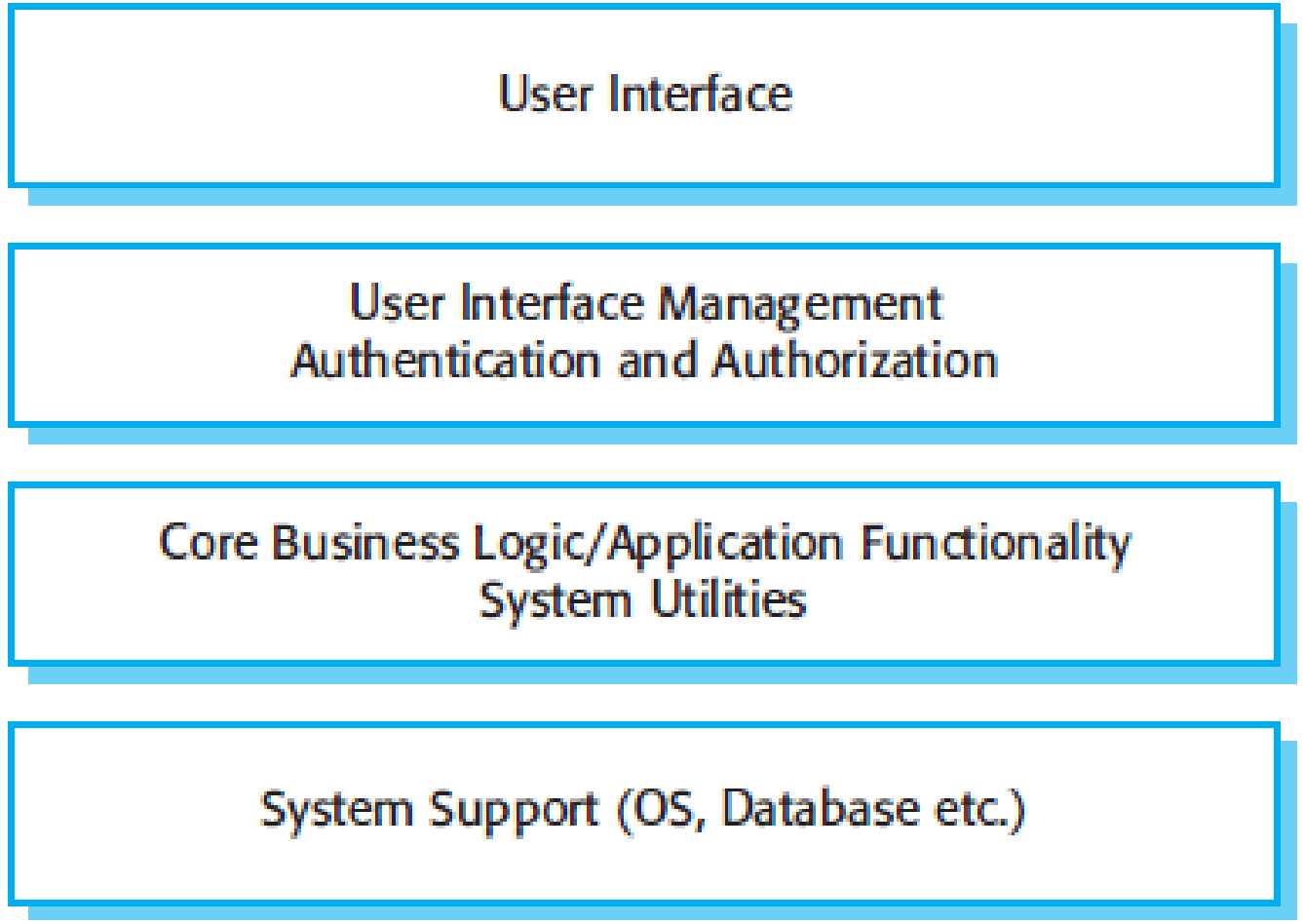
*Advantages:*

- 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:*

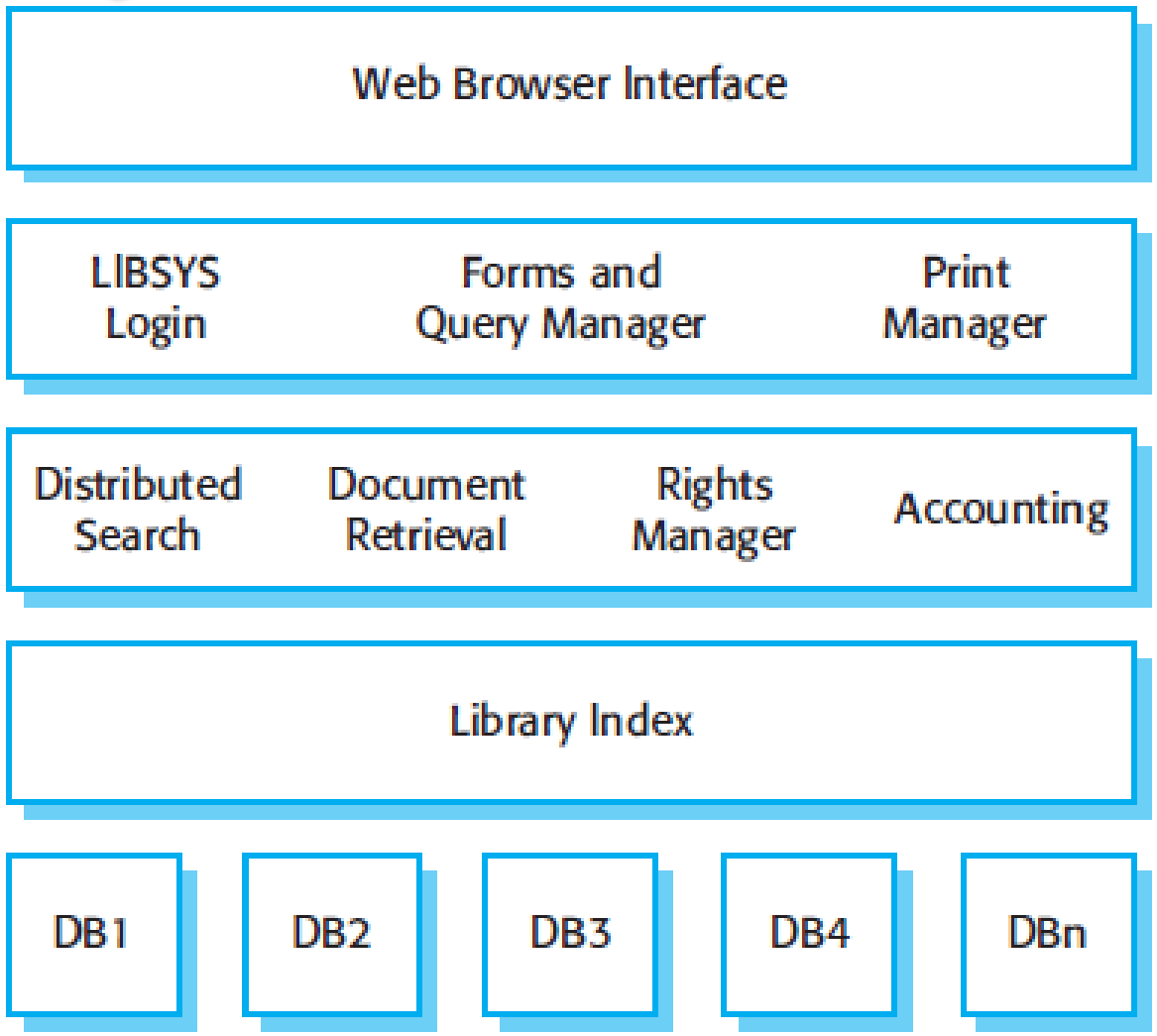
- providing a clean separation between layers is often difficult and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it
- Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer

# Layered Architecture pattern (3)




Generic Layered Architecture

# Layered Architecture pattern (4)

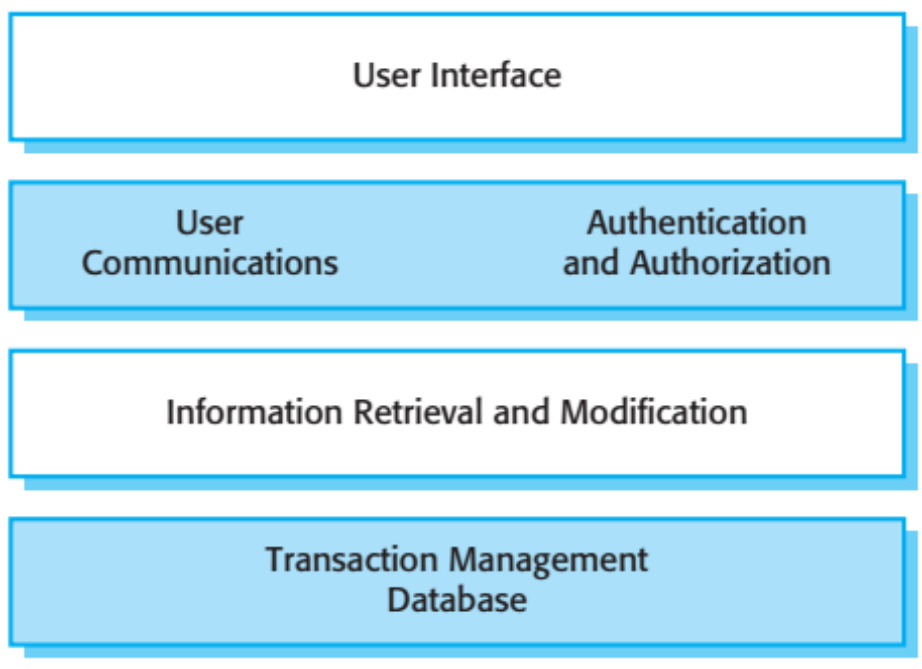



Architecture of the LIBSYS system

- allows controlled electronic access to copyright material from a group of university libraries

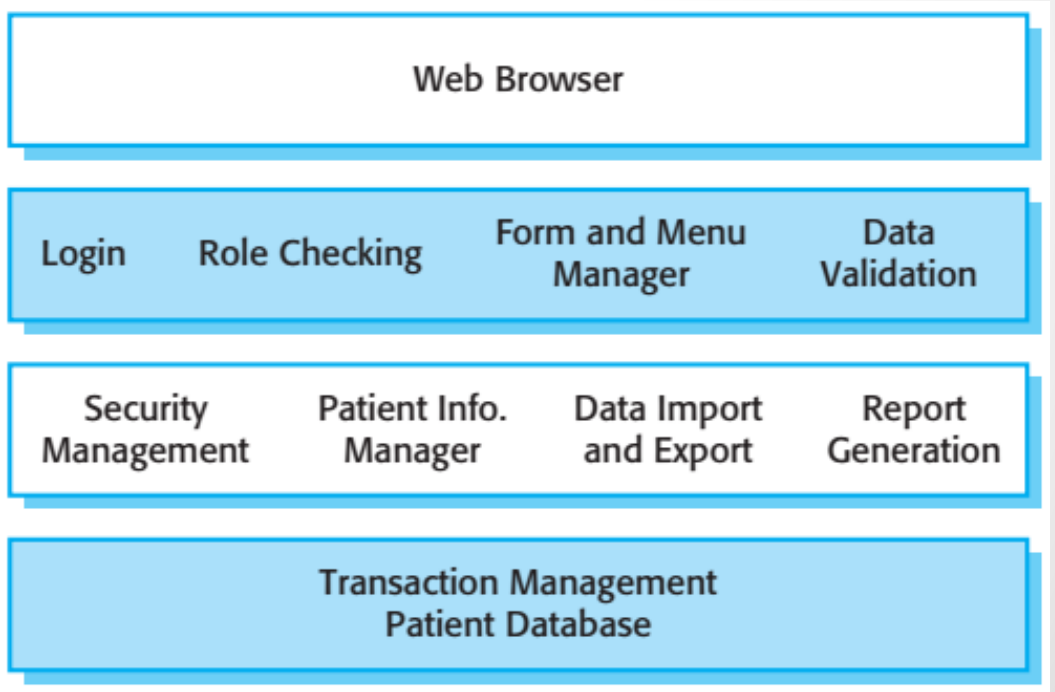
- This has a five-layer architecture, with the bottom layer being the individual databases in each library

# Layered Architecture pattern (5)




Layered information system architecture

The architecture of the MHC-PMS



# Repository Architecture pattern (1)

- describes how a set of interacting components can share data
- All data in a system is managed in a central repository that is accessible to all system components
- Components do not interact directly, only through the repository
- The majority of systems that use large amounts of data are organized around a shared database or repository

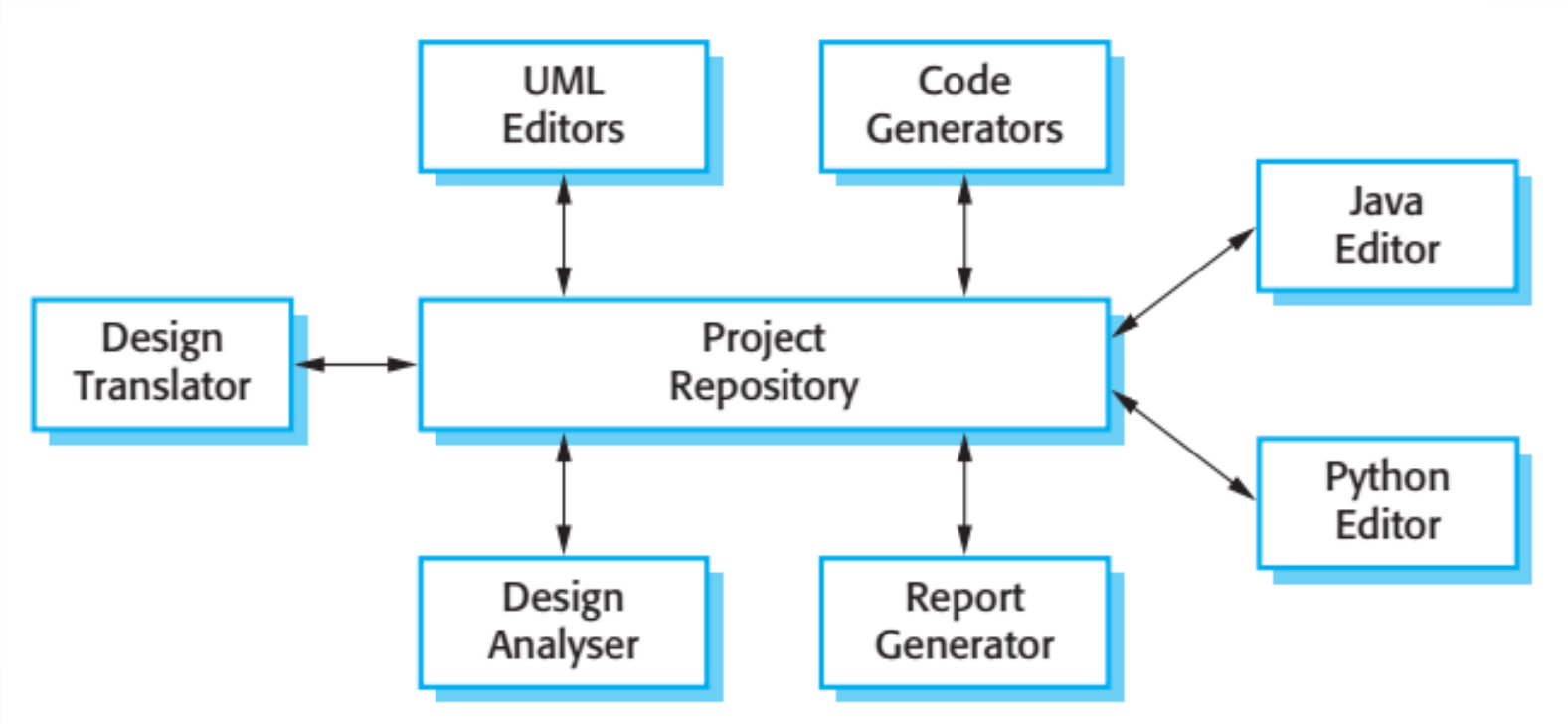
# Repository Architecture pattern (2)

- This model is suited to applications in which data is generated by one component and used by another
- The Repository pattern is used when you have a system in which large volumes of information are generated that has to be stored for a long time
- You may also use it in data-driven systems where the inclusion of data in the repository triggers an action or tool

Examples of this type of system include

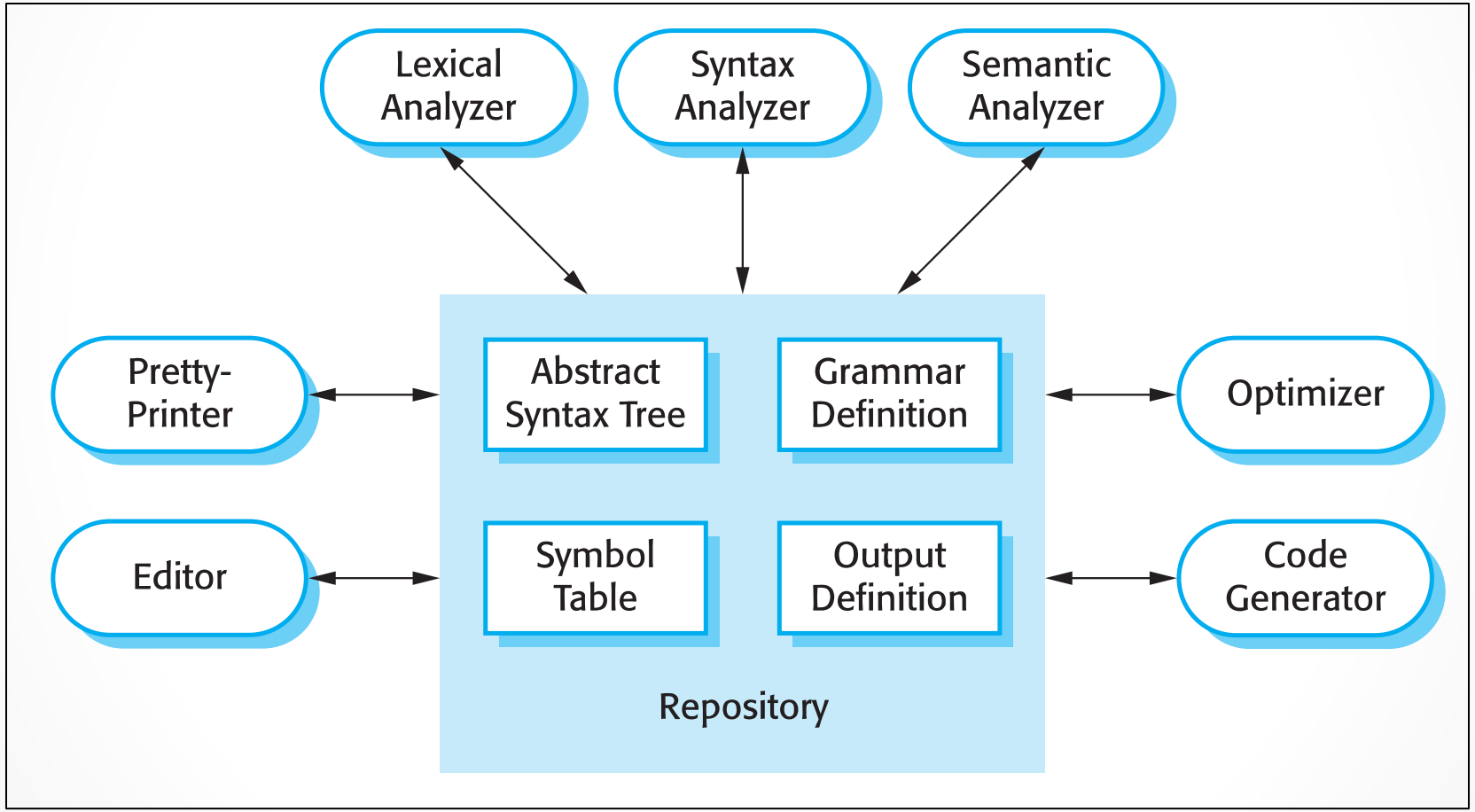
- command and control systems
- management information systems
- CAD systems
- interactive development environments for software

# Repository Architecture pattern (3)




- A repository architecture for an IDE where the components use a repository of system design information
- Each software tool generates information which is then available for use by other tools

# Repository Architecture pattern (4)




A repository architecture for a language processing system

# Repository Architecture pattern (5)

*Advantages:*

- Components can be independent—they do not need to know of the existence of other components
- Changes made by one component can be propagated to all components
- All data can be managed consistently (e.g., backups done at the same time) as it is all in one place

*Disadvantages:*

- The repository is a single point of failure so problems in the repository affect the whole system
- There may be inefficiencies in organizing all communication through the repository
- Distributing the repository across several computers may be difficult

# Client-Server Architecture pattern (1)

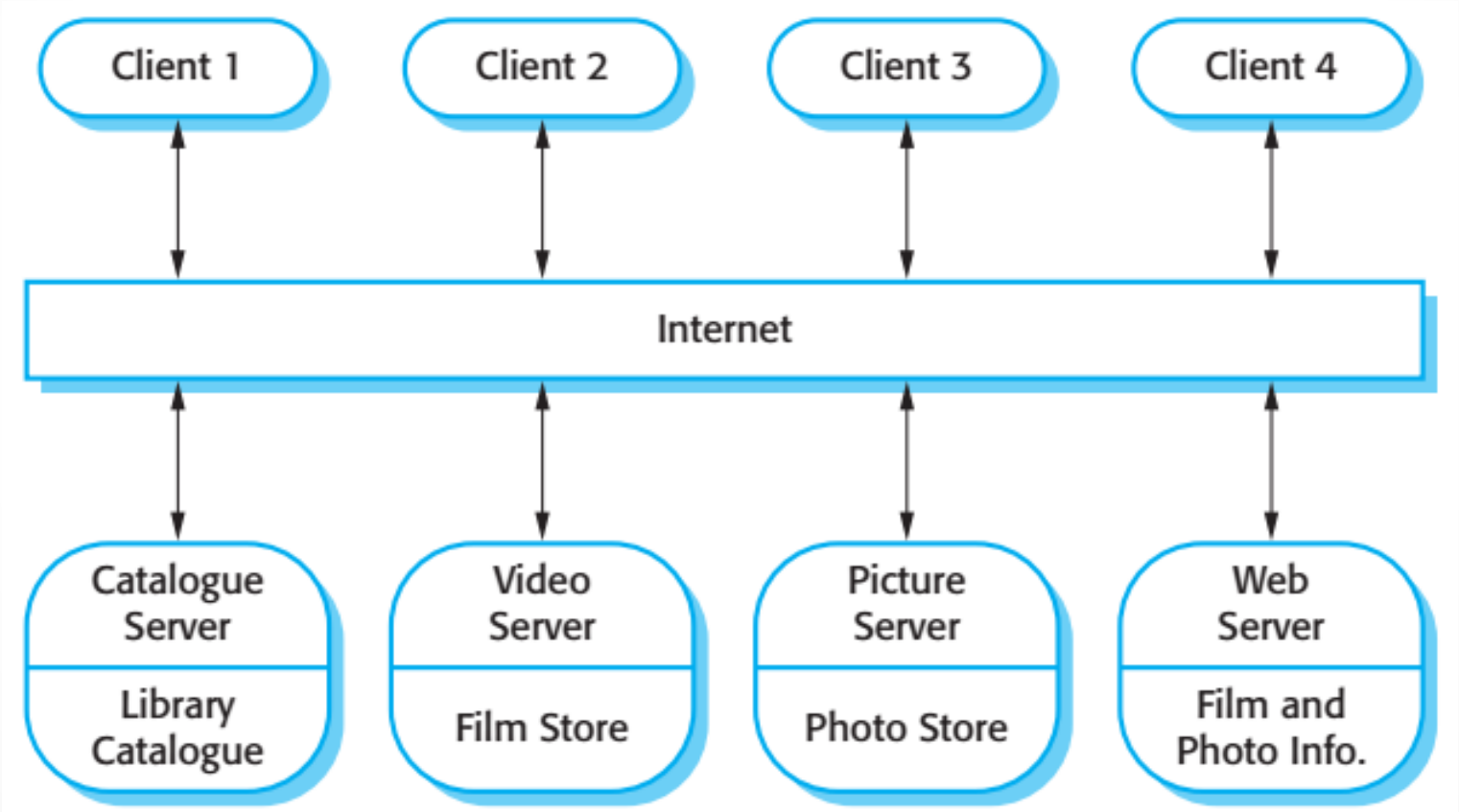
- In a client–server architecture, the functionality of the system is organized into services, with each service delivered from a separate server
- Clients are users of these services and access servers to make use of them
- A client makes a request to a server and waits until it receives a reply
- Used when data in a shared database has to be accessed from a range of locations

# Client-Server Architecture pattern (2)

Major components:

- A set of servers that offer services to other components
  - Examples of servers include print servers that offer printing services, file servers that offer file management services, and a compile server, which offers programming language compilation services
- A set of clients that call on the services offered by servers
  - There will normally be several instances of a client program executing concurrently on different computers
- A network that allows the clients to access these services
  - Most client–server systems are implemented as distributed systems, connected using Internet protocols

# Client-Server Architecture pattern (3)




A client—server architecture for a film library

# Client-Server Architecture pattern (4)

*Advantages:*

- 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

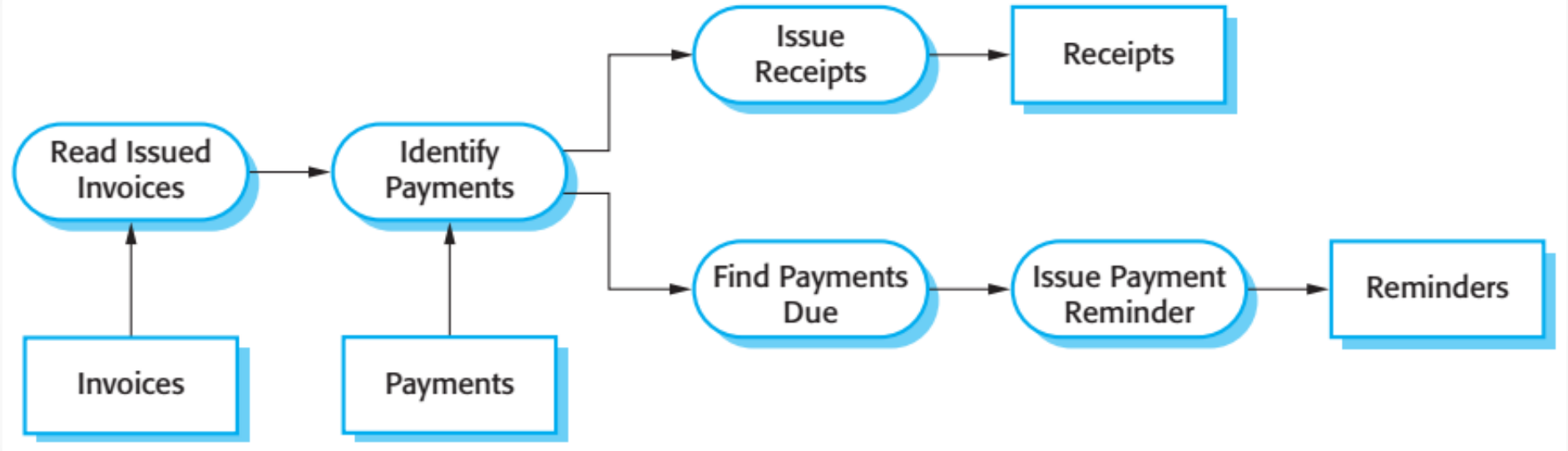
*Disadvantages:*

- Each service is a single point of failure so it susceptible to denial of service attacks or server failure
- Performance may be unpredictable because it depends on the network as well as the system
- There may be management problems if servers are owned by different organizations

# Pipe and Filter Architecture pattern (1)

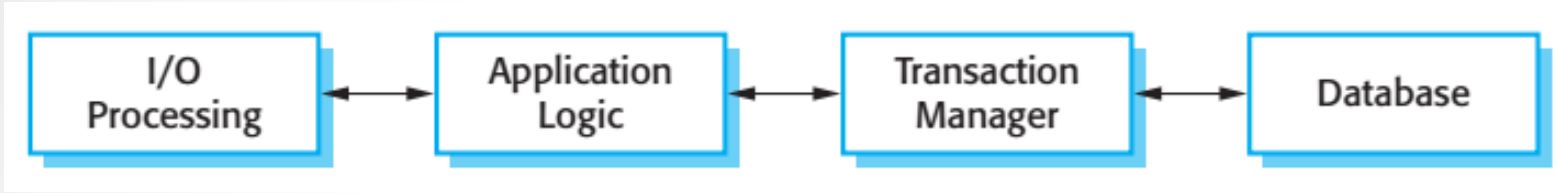
- The processing of the data in a system is organized so that each processing component (filter) is discrete and carries out one type of data transformation
- The data flows (as in a pipe) from one component to another for processing
- Commonly used in data processing applications (both batch- and transaction-based) where inputs are processed in separate stages to generate related outputs

# Pipe and Filter Architecture pattern (2)

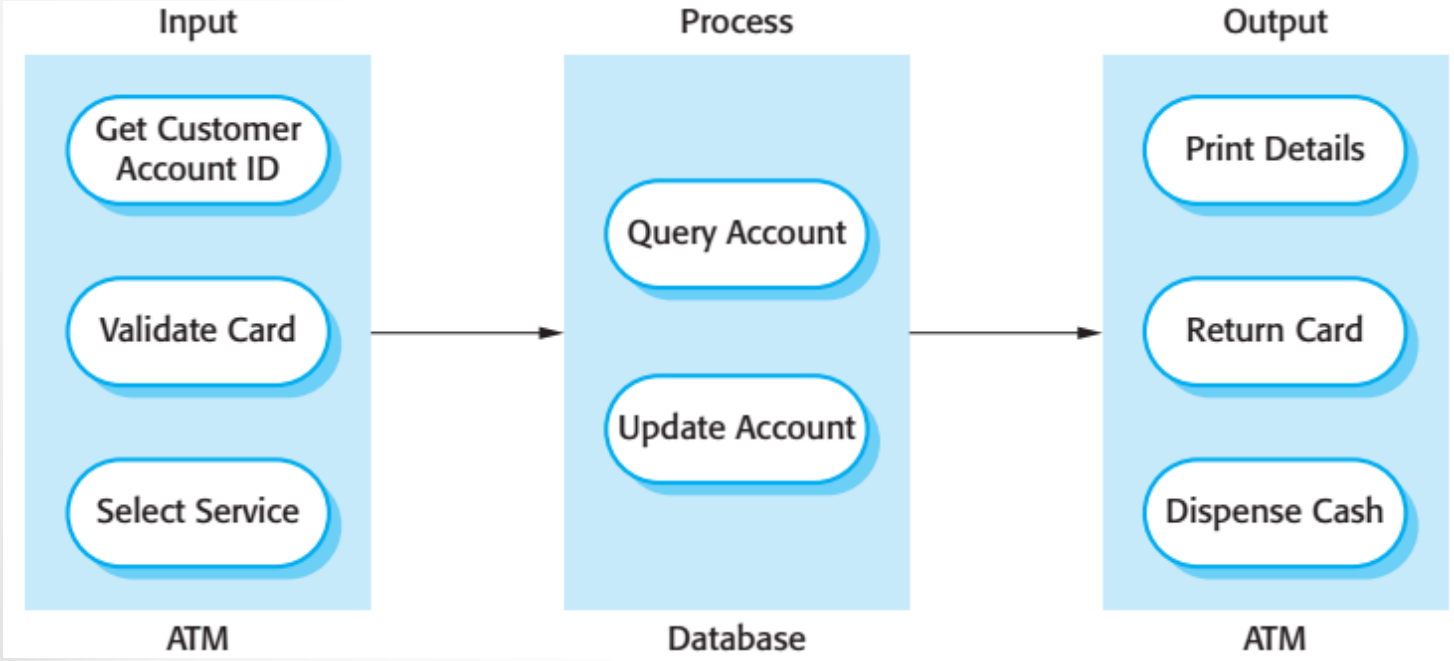



a pipe and filter system used for processing invoices

# Pipe and Filter Architecture pattern (3)

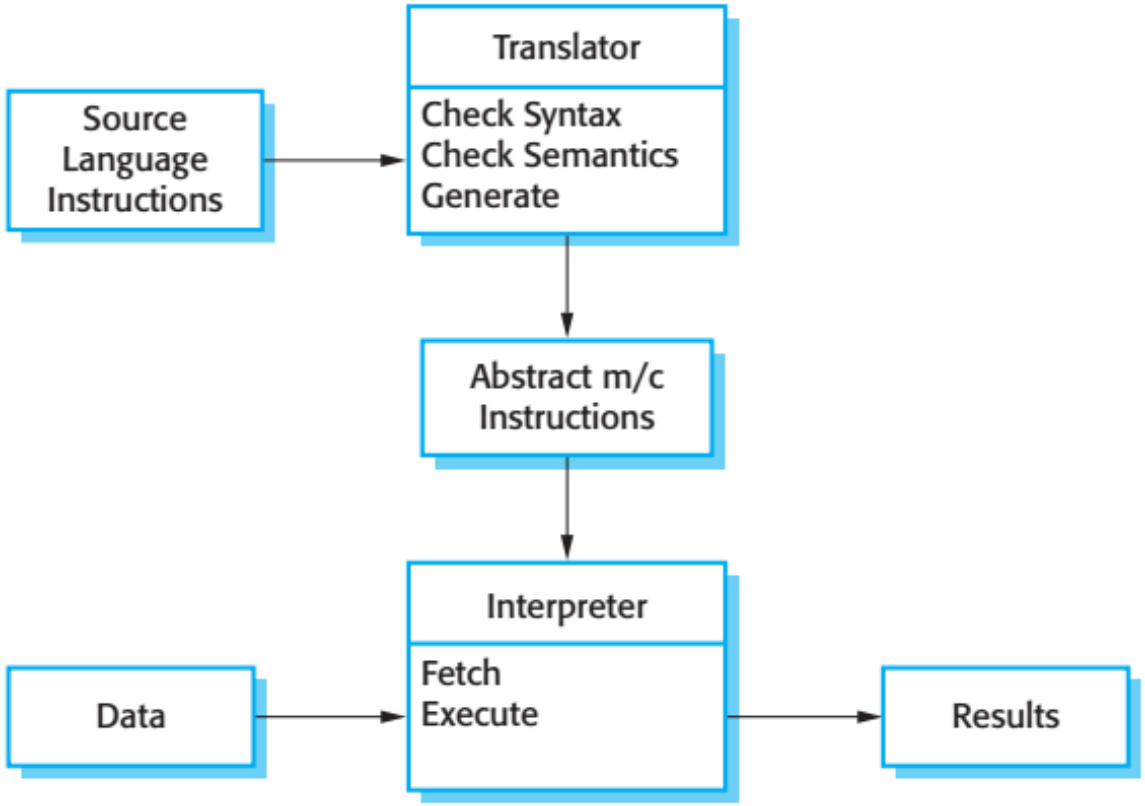



The structure of transaction processing systems (TPS)



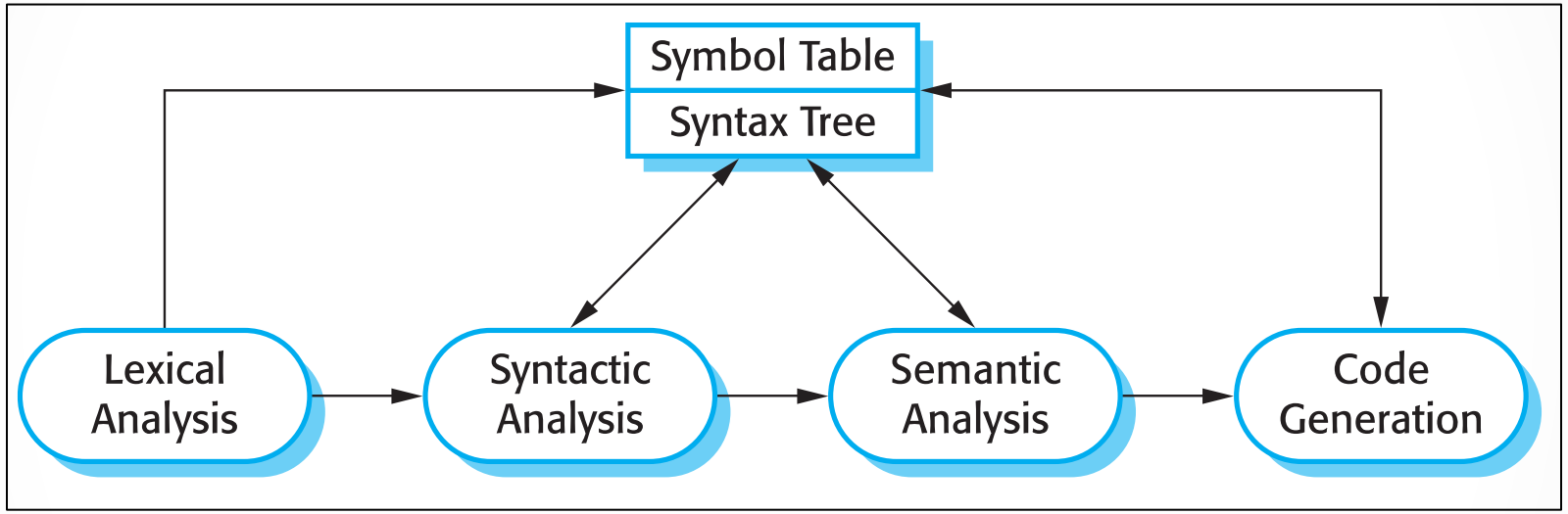

The software architecture of an ATM system

# Pipe and Filter Architecture pattern (4)




The generic architecture of a language processing system

# Pipe and Filter Architecture pattern (5)




A pipe and filter compiler architecture

# Pipe and Filter Architecture pattern (6)

*Advantages:*

- Easy to understand and supports transformation reuse
- Workflow style matches the structure of many business processes
- Evolution by adding transformations is straightforward
- Can be implemented as either a sequential or concurrent system

*Disadvantages:*

- The format for data transfer has to be agreed upon between communicating transformations
- Each transformation must parse its input and unparse its output to the agreed form
- This increases system overhead and may mean that it is impossible to reuse functional transformations that use incompatible data structures

# End of Presentation