322 371 Software Engineering

Chapter 6: Architectural Design

By

Chitsutha Soomlek, Ph.D.

Department of Computer Science

Faculty of Science, KKU

2/2560

February 19-20, 2018

Software Architecture

- Software architecture
  - is the structure or structures of the system
  - describes the overall components of an application and how they relate to each other
  - must model the structure of a system and the manner in which data and procedural components collaborate with one another

Text

Software Component

- Software components are portions of software that do not change and that do not require knowledge of the software using them.
- A software component can be something as simple as a **program module** or an **object-oriented class**.
- It can also be extended to include **databases** and **middleware** that enable the configuration of a network of clients and servers.

Architectural Design

ดีไซน์โครงสร้างจะไม่ลงลึกมาก ดีไซน์ในระบบ High level

- Architectural design is concerned with understanding how a system should be organized and designing the overall structure of that system.

Text

- It involves identifying major system components and their communications.
- The output of the architectural design process is an architectural model that describes how the system is organized as a set of communicating components.

Advantages of Explicitly Designing and Documenting Software Architecture

มีผลกับ non functional requeirement โดยตรง

ควรทำเป็นอดันดับแรกๆเลย ในการออกแบบ

- Software architecture affects the performance, robustness, distributability, and maintainability of a system (mainly non-functional requirements).
- A high-level presentation of the system can support stakeholder communication.
- Making the system architecture explicit at an early stage in the system development requires some analysis.
- A model of a system architecture can be reused in the systems with similar requirements.

ดีไซน์โครงสร้างทีหลัง อาจทะใาห้แก้ไขซอฟแวทั้งระบบ

Architectural Design in Agile

- At the early stage of the development process, an overall system architecture should be established.
- Incremental development of architectures is <u>not</u> usually successful.
- While refactoring components in response to changes is usually relatively easy, refactoring a system architecture is likely to be expensive.
- Recommend -> Service design blueprint/A-DAPT Blueprint

Example: Packing Robot

Figure from http://www.cisco-eagle.com/images/site/Robotics/Robotic_Palletizer_System_2.jpg

Example: Packing Robot

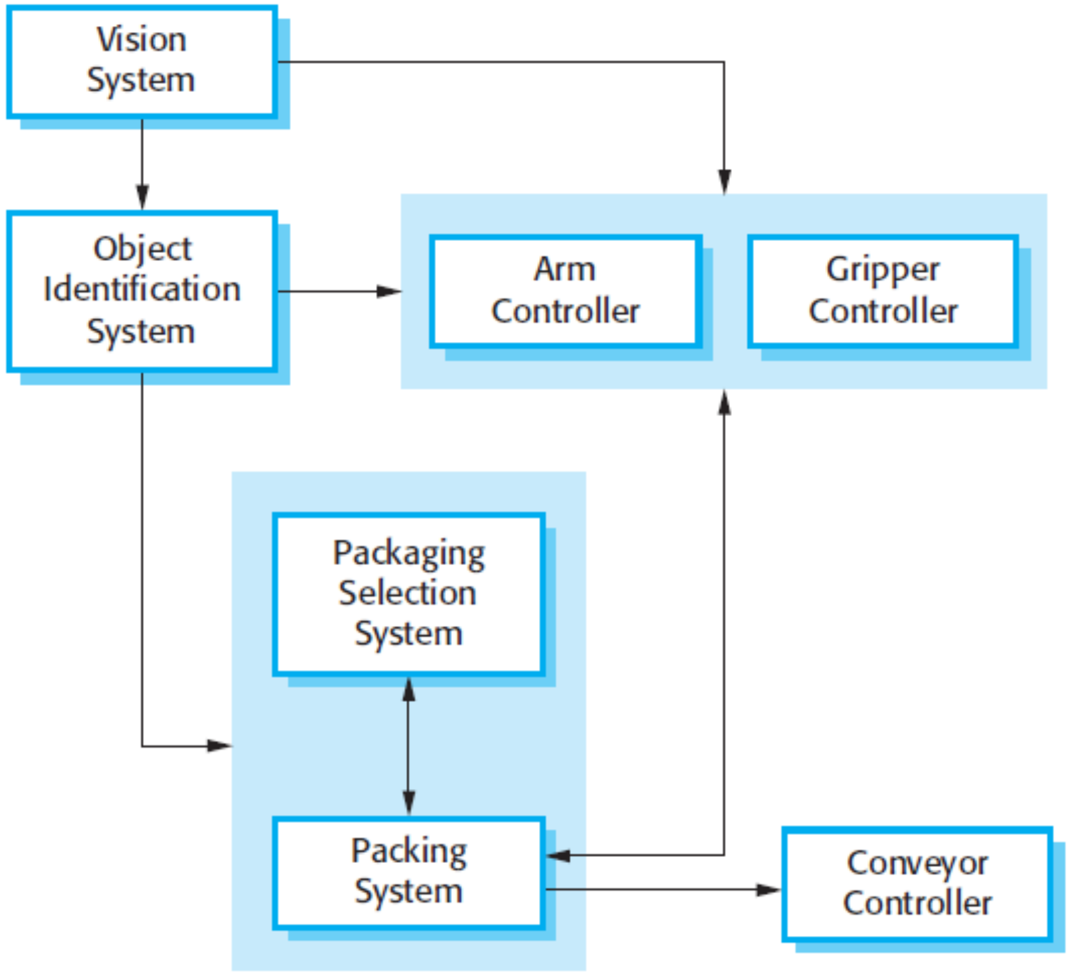


Figure from I. Sommerville, Software Engineering, 9th ed.

Levels of Abstractions

- **Architecture in small** – focuses on
  - the architecture of individual programs
  - the way that an individual program is decomposed into components

การดีไซนซอฟแวที่ซับซ้อนกว่านั้น เอา sub system มาทำงานร่วมกัน

- **Architecture in large** – focuses on 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

Architectural Design Decisions

- Is there a generic application architecture that can act as a template for the system that is being designed?
- How will the system be distributed across a number of cores or processors?
- What architectural patterns or styles might be used?
- What will be the fundamental approach used to structure the system?

Architectural Design Decisions

- How will the structural components in the system be decomposed into subcomponents?
- What strategy will be used to control the operation of the components in the system?
- What architectural organization is best for delivering the non-functional requirements of the system?
- How will the architectural design be evaluated?
- How should the architecture of the system be documented?

Steps in Architectural Design

- Architectural design is divided to two levels:
  - **Data design**: represents the data component of the architecture.
  - **Architectural design**: focuses on the representation of the structure of software components, their properties, and interactions. Alternative architectural styles or patterns are analyzed to derive the structure that is best suited to customer requirements and quality attributes.

Architectural Genre

- **Artificial intelligence**—Systems that simulate or augment human cognition, locomotion, or other organic processes.
- **Commercial and nonprofit**—Systems that are fundamental to the operation of a business enterprise.
- **Communications**—Systems that provide the infrastructure for transferring and managing data, for connecting users of that data, or for presenting data at the edge of an infrastructure.
- **Content authoring**—Systems that are used to create or manipulate textual or multimedia artifacts.

Architectural Genre

- **Devices**—Systems that interact with the physical world to provide some point service for an individual.
- **Entertainment and sports**—Systems that manage public events or that provide a large group entertainment experience.
- **Financial**—Systems that provide the infrastructure for transferring and managing money and other securities.
- **Games**—Systems that provide an entertainment experience for individuals or groups.

Architectural Genre

- **Government**—Systems that support the conduct and operations of a local, state, federal, global, or other political entity.
- **Industrial**—Systems that simulate or control physical processes.
- **Legal**—Systems that support the legal industry.
- **Medical**—Systems that diagnose or heal or that contribute to medical research.
- **Military**—Systems for consultation, communications, command, control, and intelligence (C4I) as well as offensive and defensive weapons**.**

Architectural Genre

- **Operating systems**—Systems that sit just above hardware to provide basic software services.
- **Platforms**—Systems that sit just above operating systems to provide advanced services.
- **Scientific**—Systems that are used for scientific research and applications.
- **Tools**—Systems that are used to develop other systems.
- **Transportation**—Systems that control water, ground, air, or space vehicles.
- **Utilities**—Systems that interact with other software to provide some point service.

Example

- Game systems (immersive interactive applications) require
  - the computation of intensive algorithms,
  - sophisticated computer graphics,
  - streaming multimedia data sources,
  - real-time interactivity via conventional and unconventional inputs, and
  - a variety of other specialized concerns.

Example (Cont.)

- The architecture of game systems:
  - Perform distributed asynchronous parallel processing (shared repository and message-passing)
  - Employ extensible data model
  - Use libraries or native code alike
  - Support distributed code development, testing, reuse, fast system design and integration, maintenance, and evolution

Architectural Patterns

- An architectural pattern or style is a description of a system organization.
- An architectural pattern captures the essence of an architecture that has been used in different software systems.
- Caution!! When making decisions about the architecture of a system, make sure that you understand their strengths and weaknesses.

Designing With Architectural Patterns

1. Choose the most appropriate structure that will enable you to meet the system requirements
2. Decompose structural system units - Decide on the strategy for decomposing components into sub-components
3. Control modeling process:
   - make decisions about how the execution of components is controlled.
   - develop a general model of the control relationships between the various parts of the system

How to Choose the Right Pattern?

- Take the non-functional system requirements into consideration
  - Performance
  - Security
  - Safety
  - Availability
  - Maintainability

When Performance Matters

- The architecture should be designed to localize critical operations within a small number of components, with these components all deployed on the same computer rather than distributed across the network.
- Putting a few relatively large components rather than small, fine-grain components in your design, which reduces the number of component communications.
- Caution!! Beware of complexity

When Security Matters

- A layered structure for the architecture should be used.
- An architectural design can enhance the security level by protecting the most critical assets in the innermost layers, with a high level of security validation applied to these layers.

When Safety Matters

- The architecture should be designed so that safety-related operations are all located in either a single component or in a small number of components.
- This reduces the costs and problems of safety validation and makes it possible to provide related protection systems that can safely shut down the system in the event of failure.

When Availability Matters

- The architecture should be designed to include redundant components so that it is possible to replace and update components without stopping the system.
- Considering fault-tolerant system architectures for high-availability systems.

When Maintainability Matters

- The system architecture should be designed using fine-grain, self-contained components that may readily be changed.
- Producers of data should be separated from consumers and shared data structures should be avoided.

Categorization of Software Architectures

- Data flow architecture
  - Pipes and filters
  - Batch sequential
- Independent components
  - Tiered and client-server
  - Parallel communicating processes
  - Event systems
  - Service-oriented
- Object-oriented architectures
- Virtual machines
  - Interpreters
  - Rule-based systems
- Repository/Data-centered architectures
  - Databases
  - Hypertext systems
  - Blackboards
- Layered architectures
- Model-View-Controller (MVC)

Data-Flow Architectures

- Applications are best viewed as data flowing among processing units. เหมาะกับระบบที่มีดาต้าไหลเข้าไหลออก
- This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data.
- Commonly used in data processing applications (both batch- and transaction-based) where inputs are processed in separate stages to generate related outputs.

Example

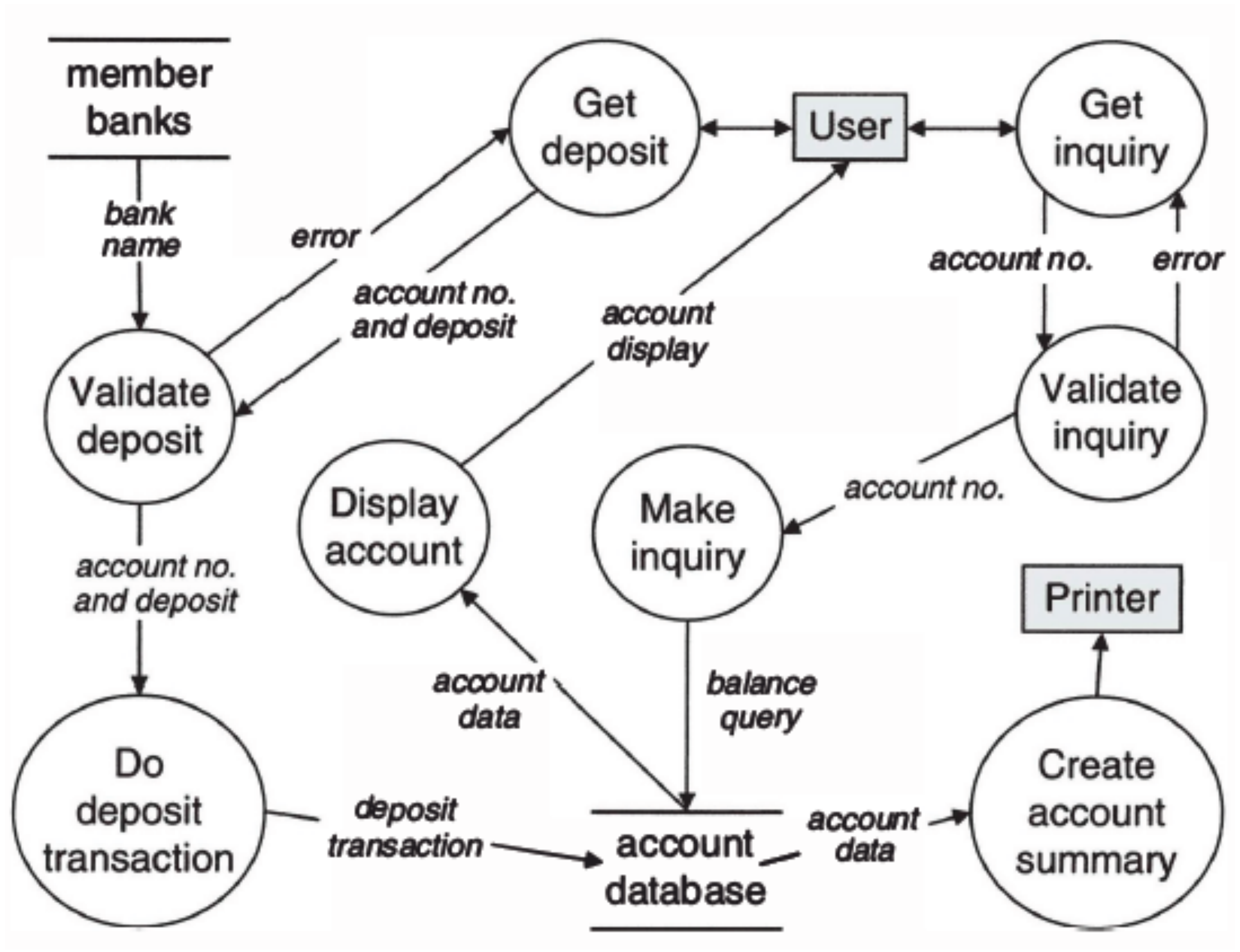


Figure from E. J. Braude and M. E. Bernstein, “Software Engineering: Modern Appproahces”, 2nd ed., 2016

Example

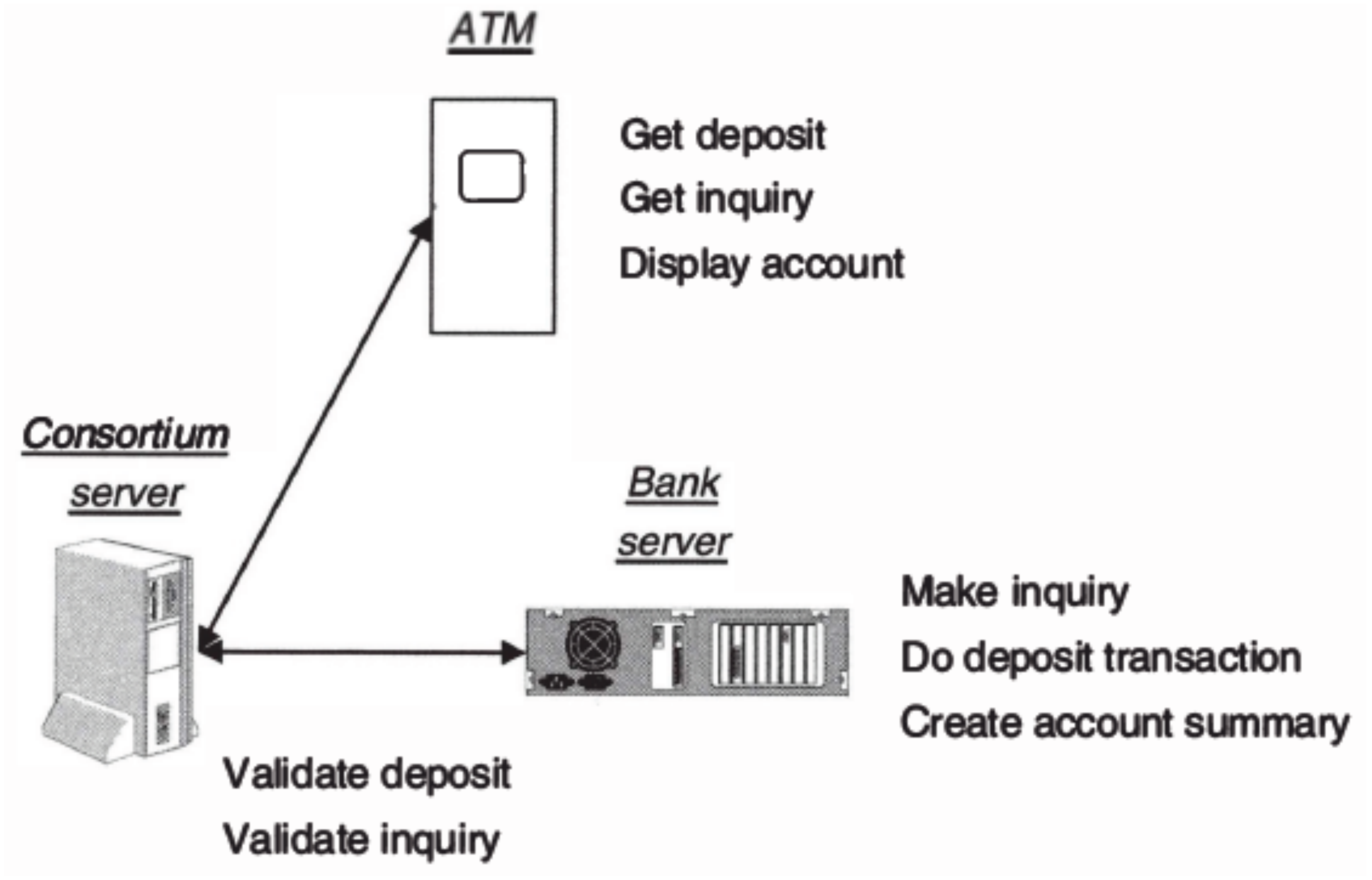
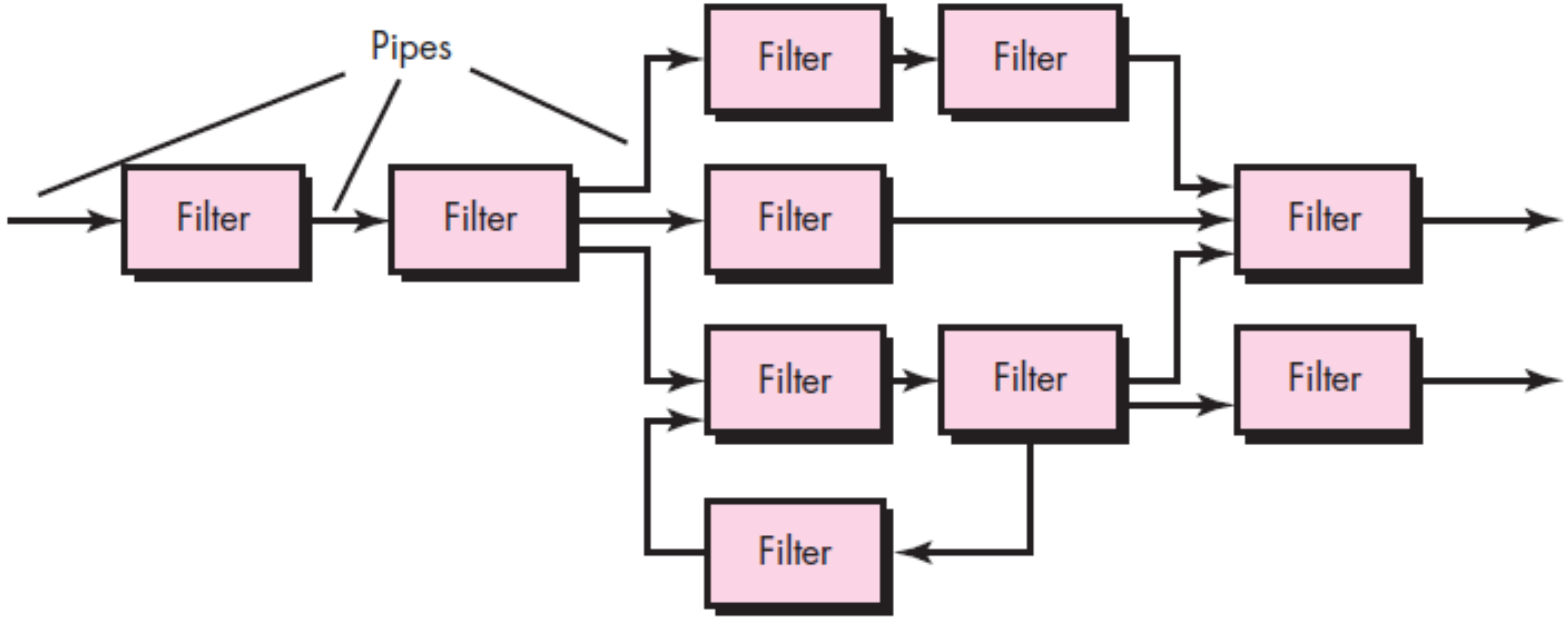


Figure from E. J. Braude and M. E. Bernstein, “Software Engineering: Modern Appproahces”, 2nd ed., 2016

Pipe-and-Filter Pattern



Pipes and filters

Figure from R.S. Pressman, Software Engineering: A Practitioner's Approach , 7th ed.

Pipe-and-Filter Pattern

- Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form. However, the filter does not require knowledge of the workings of its neighboring filters.
- The name ‘pipe and filter’ comes from the original Unix system where it was possible to link processes using ‘pipes’. These passed a text stream from one process to another.

Example

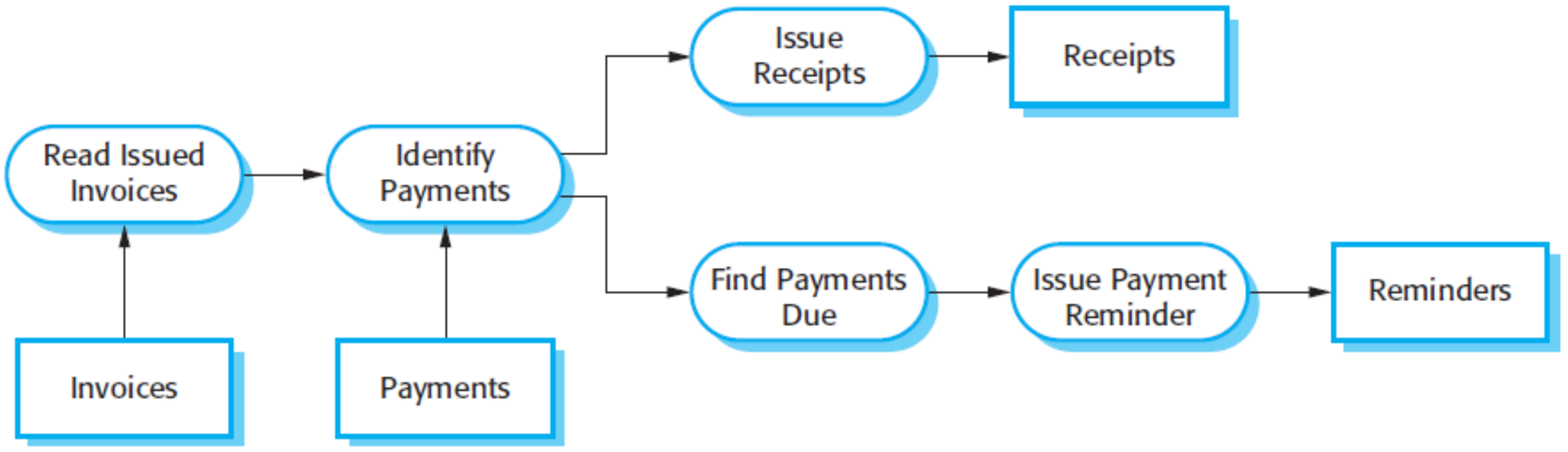
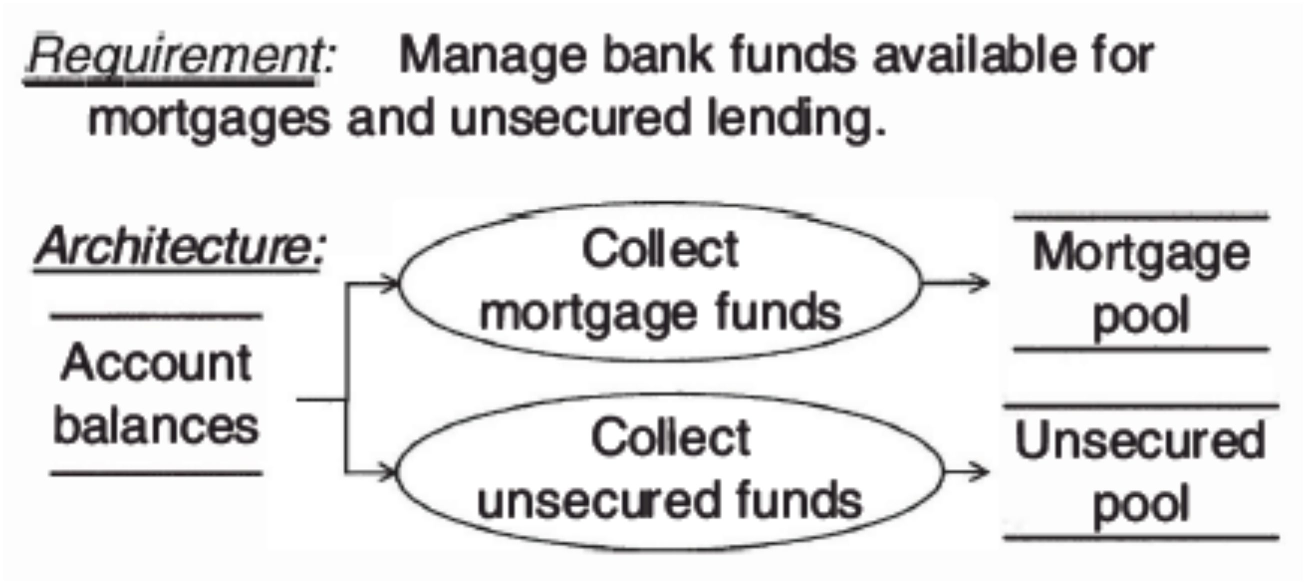


Figure from I. Sommerville, Software Engineering, 9th ed.

Batch Sequential

- The processing elements are only given batches of data. สเตปไม่เยอะแต่มีดาต้าเข้ามาเยอะ
- The functions are executed using all of the input data of a given run, taken together. Text



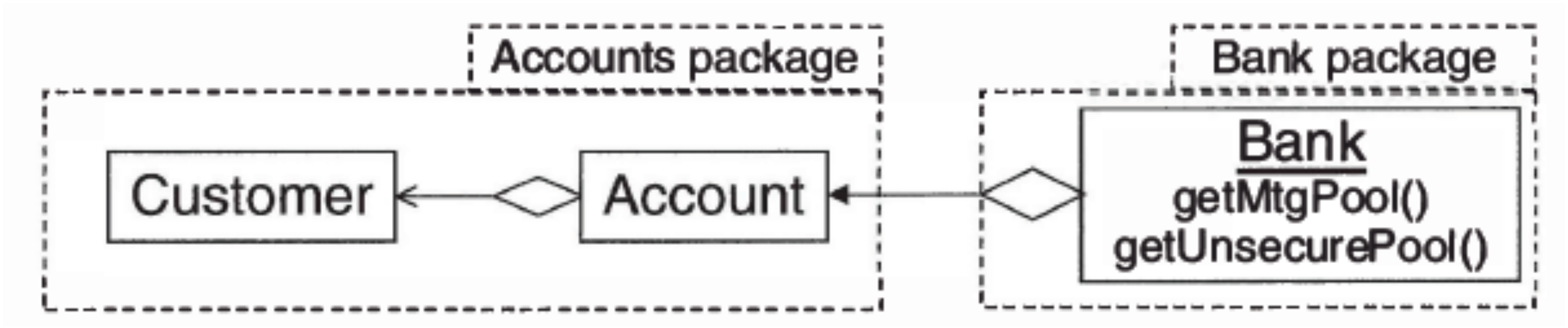


Figure from E. J. Braude and M. E. Bernstein, "Software Engineering: Modern Appproahces", 2nd ed., 2016

Data-Flow Architectures

Advantages

- Modularity
- 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.

Independent Components Architecture

- The architecture consists of components operating in parallel and communicating with each other from time to time.
- Independent components can be created by developers for various purposes and added to the platform without affecting existing functionality.

Tiered and Client-Server Architectures

- 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.
- 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.

Example

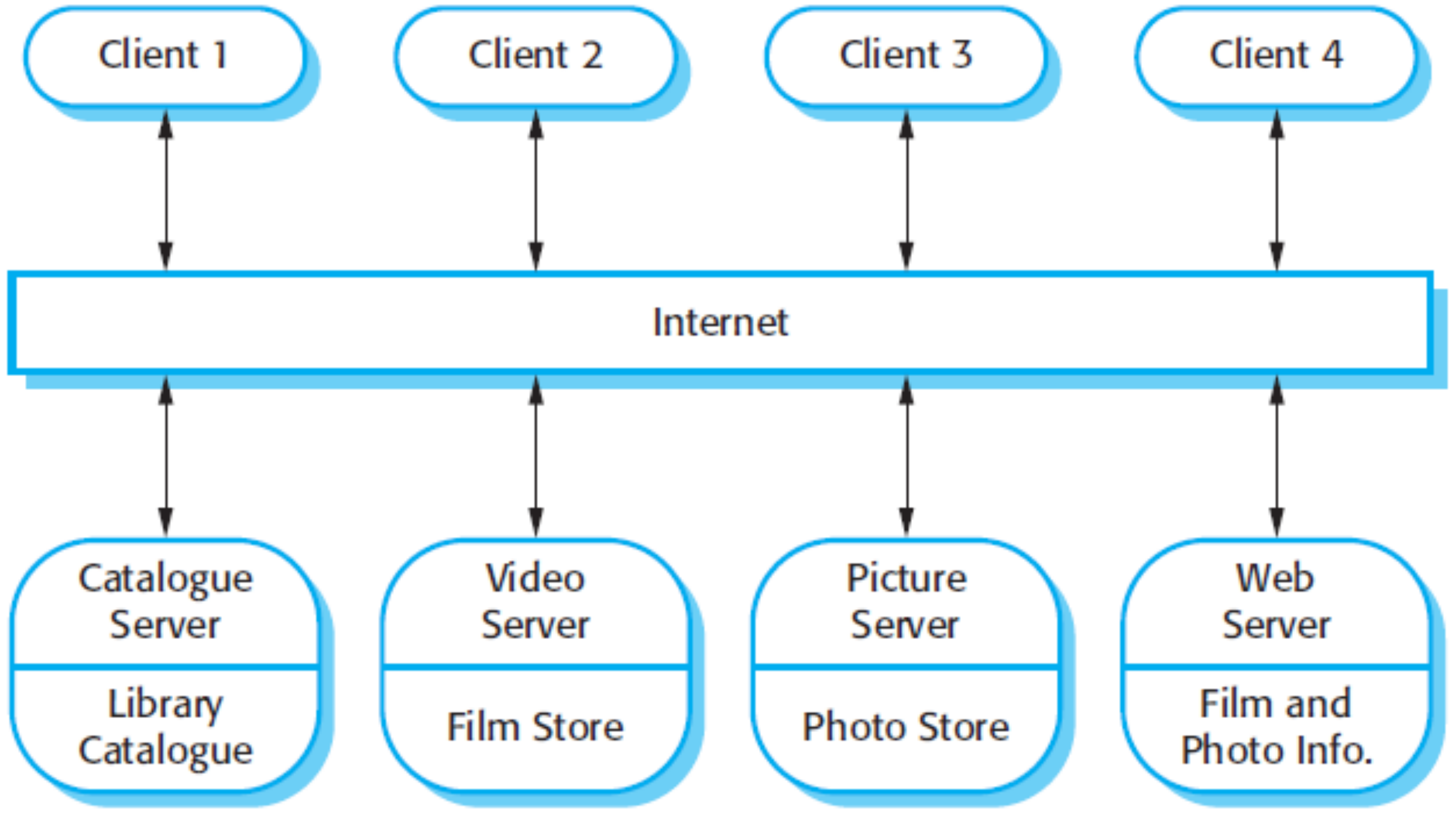


Figure from I. Sommerville, Software Engineering, 9th ed.

Tiered and Client-Server Architectures

Advantages

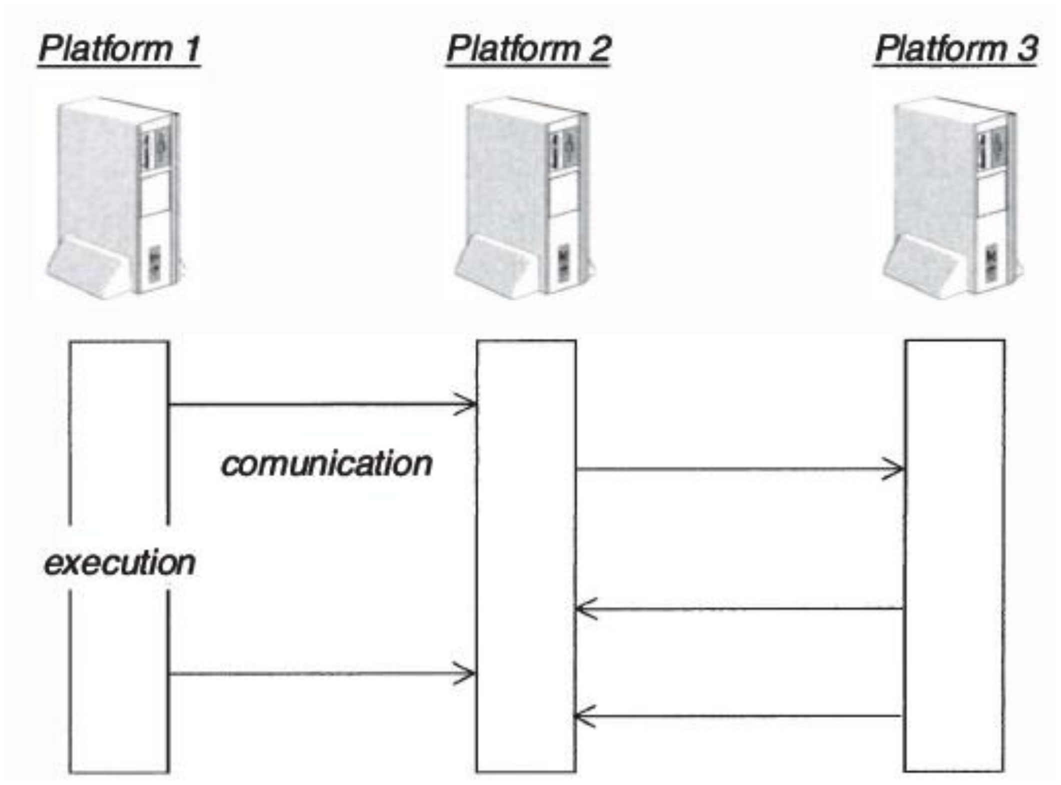
- Low coupling
- Servers can be distributed across a network.
- General functionality (e.g., a printing service) can be available to all clients and does not need to be implemented by all services.

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.

Parallel Communicating Process Architecture

- This architecture is characterized by several processes, or threads, executing at the same time.



Architectural Patterns for Control

- The patterns here reflect common control ways of organizing the control in a system.
- There are two major categories of the patterns:
  - **Centralized control** - there is a component in charge which calls on services from other components in the system. The reflects the method/procedure/ subroutine calls in programming languages.
  - **Event-based control** - the system responds to asynchronous events from the system's environment.

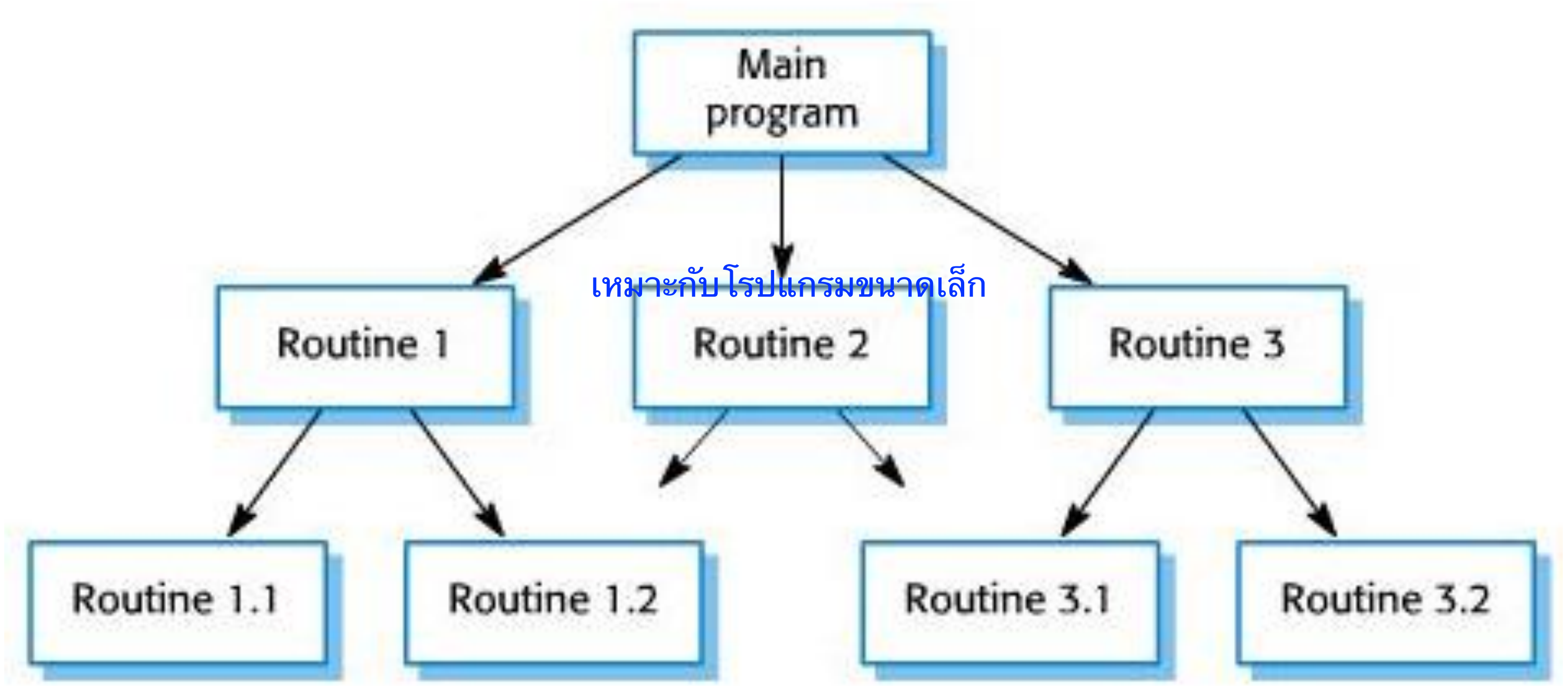
Centralized Control Model

- One component is designated as the controller and is responsible for managing the execution of other components.
- Centralized control models fall into two classes, depending on whether the controlled components execute sequentially or in parallel.
  - **Call and return architectures**
  - **The Manager model**

Call and Return Architectures

- This is the familiar top-down subroutine model where control starts at the top of a subroutine hierarchy and, through subroutine calls, passes to lower levels in the tree.
- The currently executing subroutine has responsibility for control and can either call other routines or return control to its parent.
- The subroutine model is only applicable to sequential systems.

Example



เหมาะกับโปรแกรมขนาดเล็ก

Figure from http://ifs.host.cs.st-andrews.ac.uk/Books/SE9/Web/Architecture/web-images/CallReturn.jpg

Call and Return Architectures

- This architectural style enables you to achieve a program structure that is relatively easy to modify and scale. A number of sub-styles exist within this category:
  - **Main program/subprogram architectures**: This classic program structure decomposes function into a control hierarchy where a “main” program invokes a number of program components that in turn may invoke still other components.
  - **Remote procedure call architectures**: The components of a main program/subprogram architecture are distributed across multiple computers on a network.

Main program/Subprogram Architectures

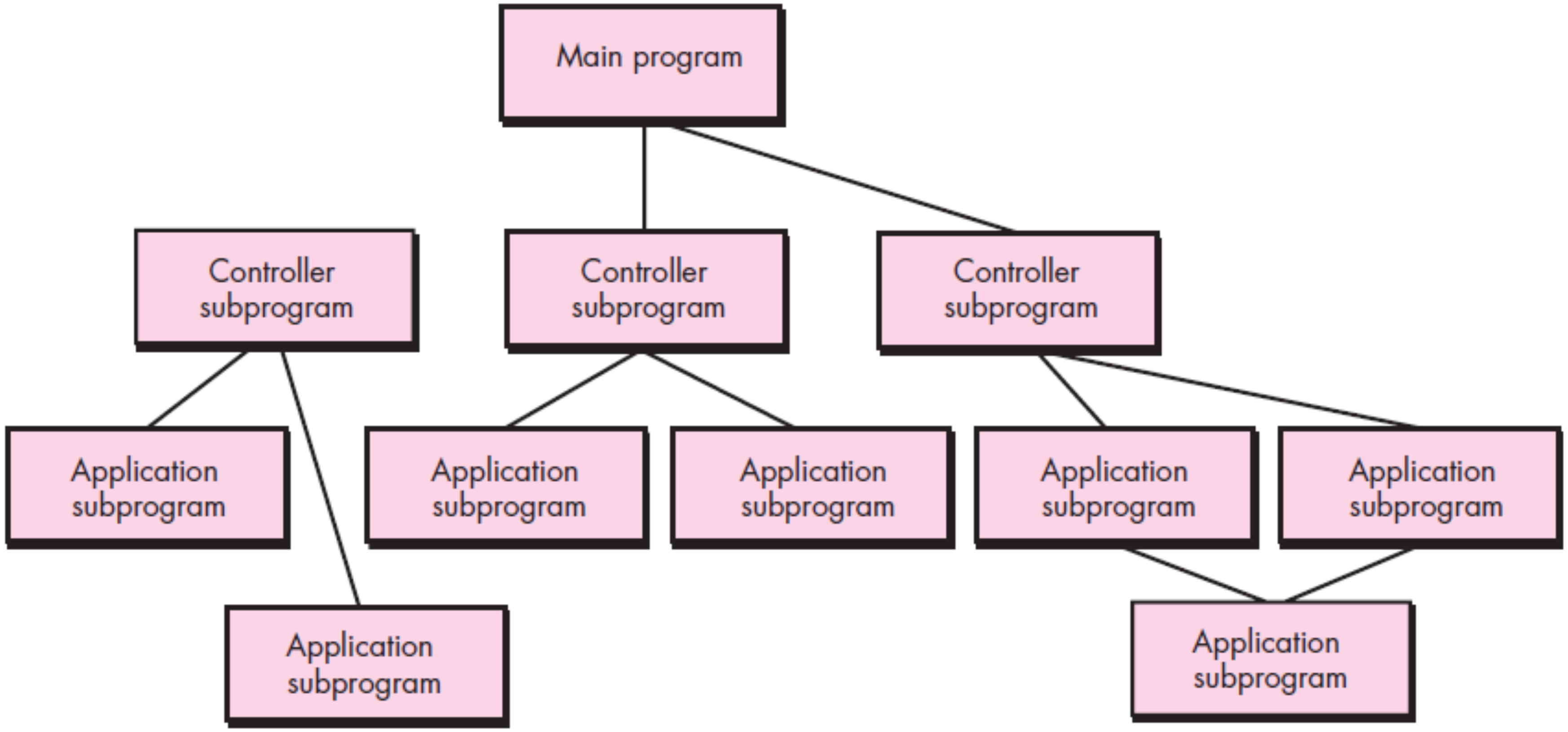
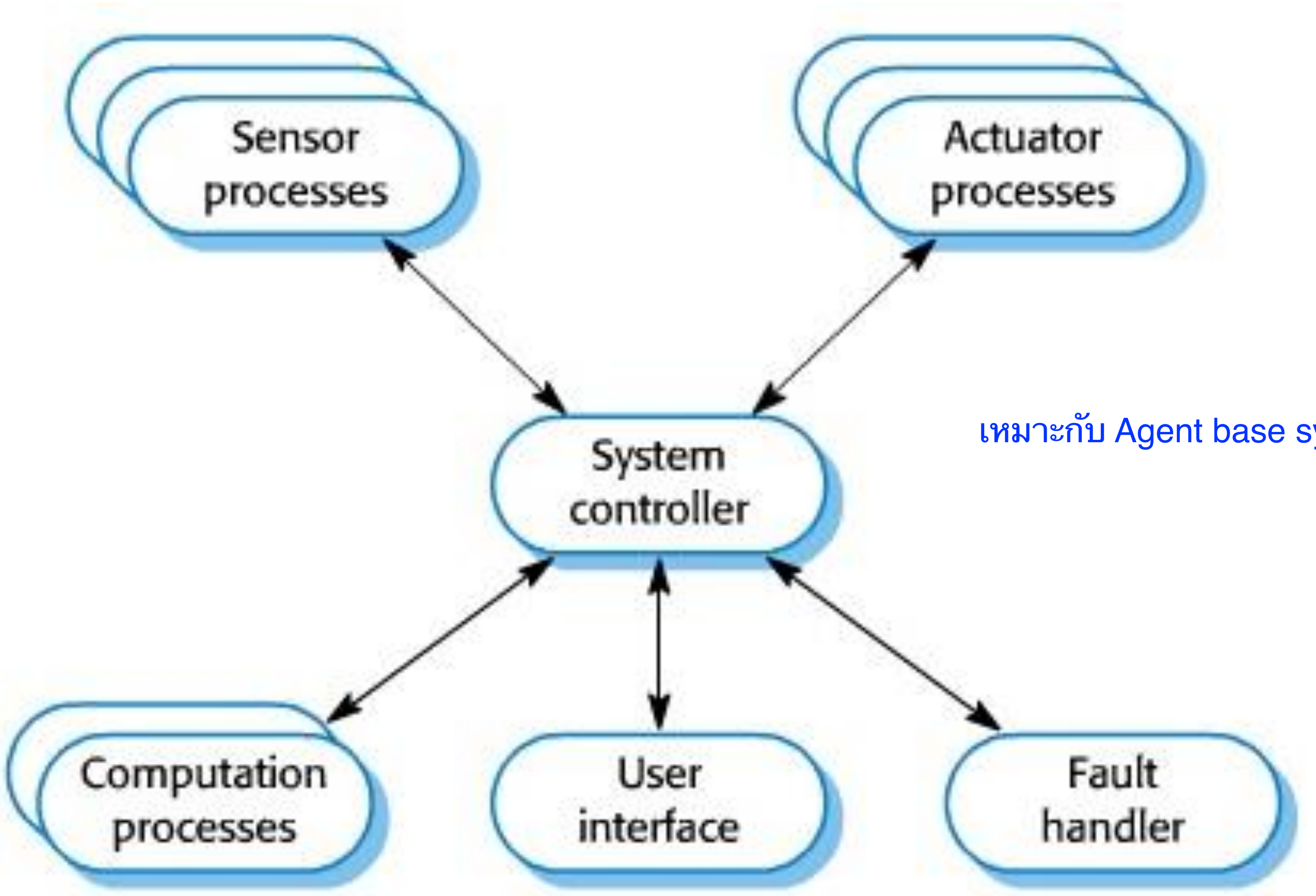


Figure from R.S. Pressman, Software Engineering: A Practitioner's Approach , 7th ed.

The Manager Model

- This model is applicable to concurrent systems and is often used in ‘soft’ real-time systems which do not have very tight time constraints.
- One system component is designated as a system manager and controls the starting, stopping, coordination and scheduling of other system processes. A process is a component or module that can execute in parallel with other processes.
- A form of this model may also be applied in sequential systems where a management routine calls particular components depending on the values of some state variables.

Example



เหมาะกับ Agent base system

Figure from http://ifs.host.cs.st-andrews.ac.uk/Books/SE9/Web/Architecture/web-images/CentralisedControl.jpg

Event-Driven Control Model

- Event-driven control models are driven by externally generated events.
- Events can be a signal that can take a range of values or a command input from a menu.
- There are many types of event-driven systems. For example,
  - **Broadcast models** - An event is broadcast to all components. Any component that has been programmed to handle that event can respond to it.
  - **Interrupt-driven models** - These are exclusively used in real-time systems where external interrupts are detected by an interrupt handler. They are then passed to some other component for processing.

Example of Broadcast Model

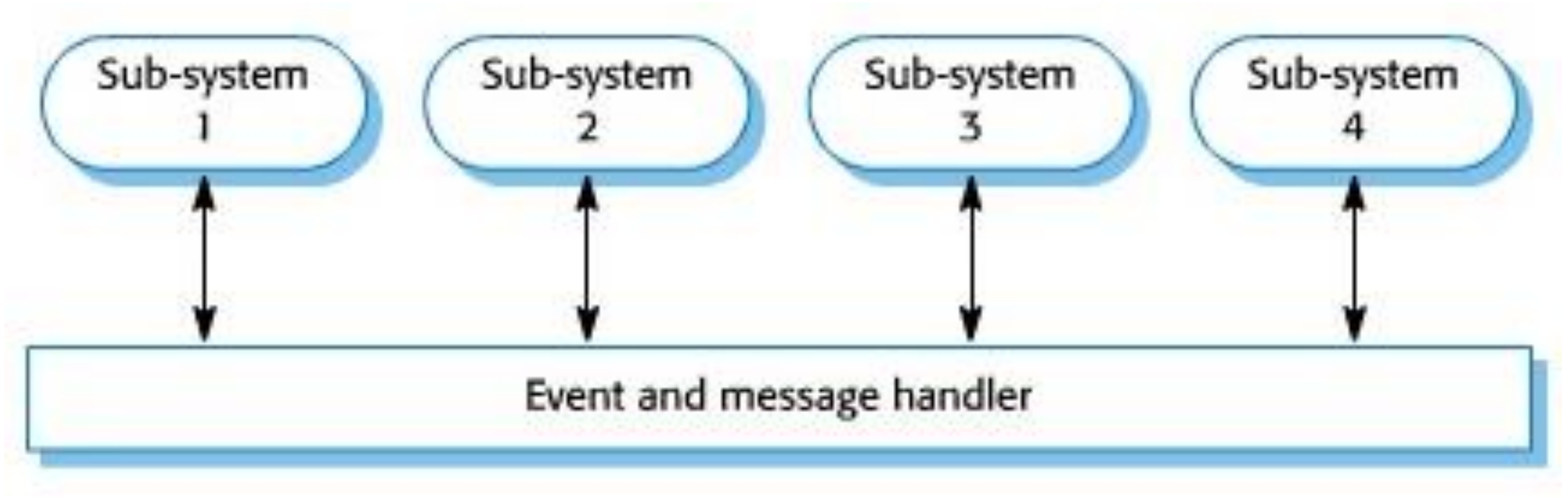
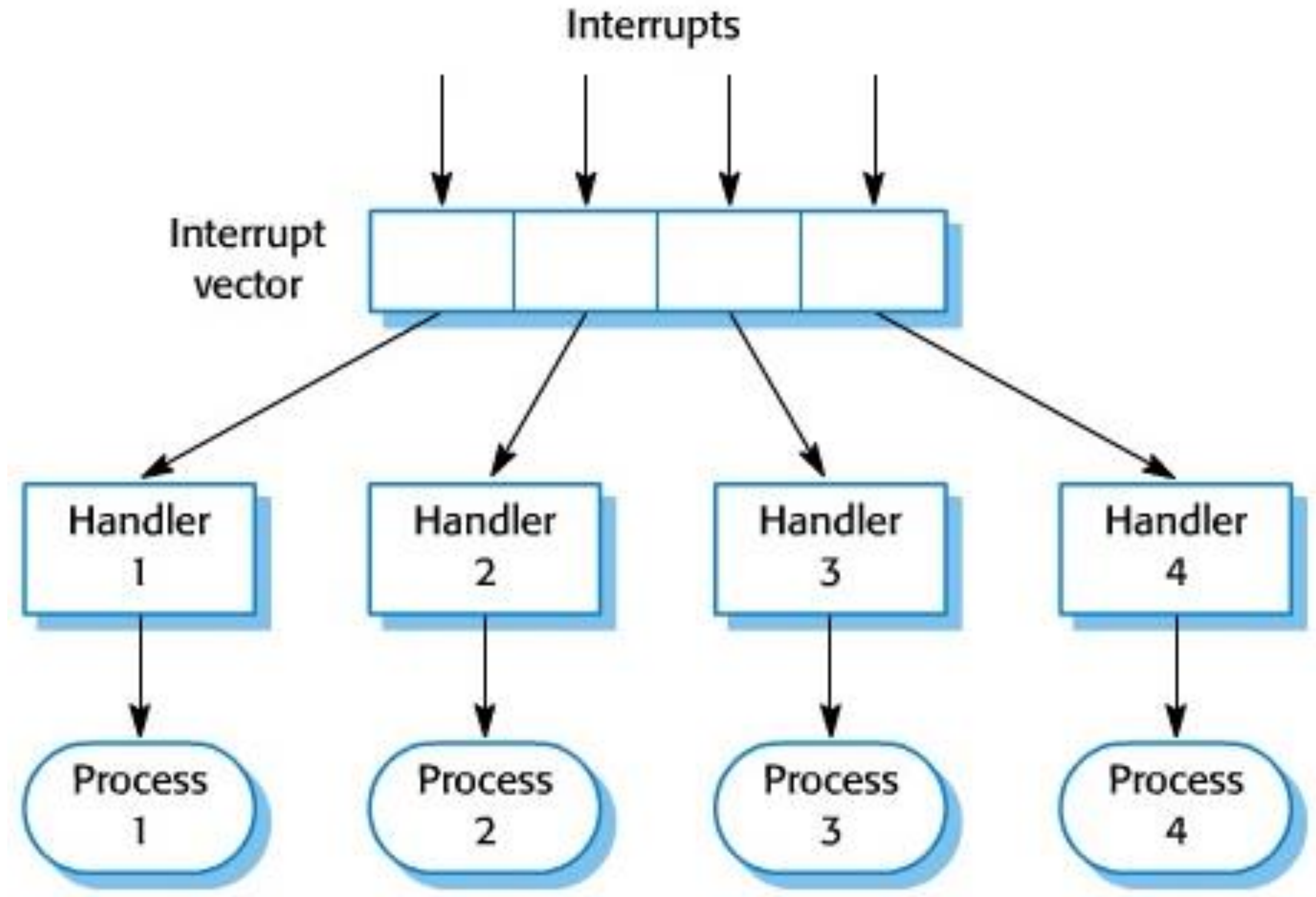


Figure from http://ifs.host.cs.st-andrews.ac.uk/Books/SE9/Web/Architecture/web-images/EventBroadcast.jpg

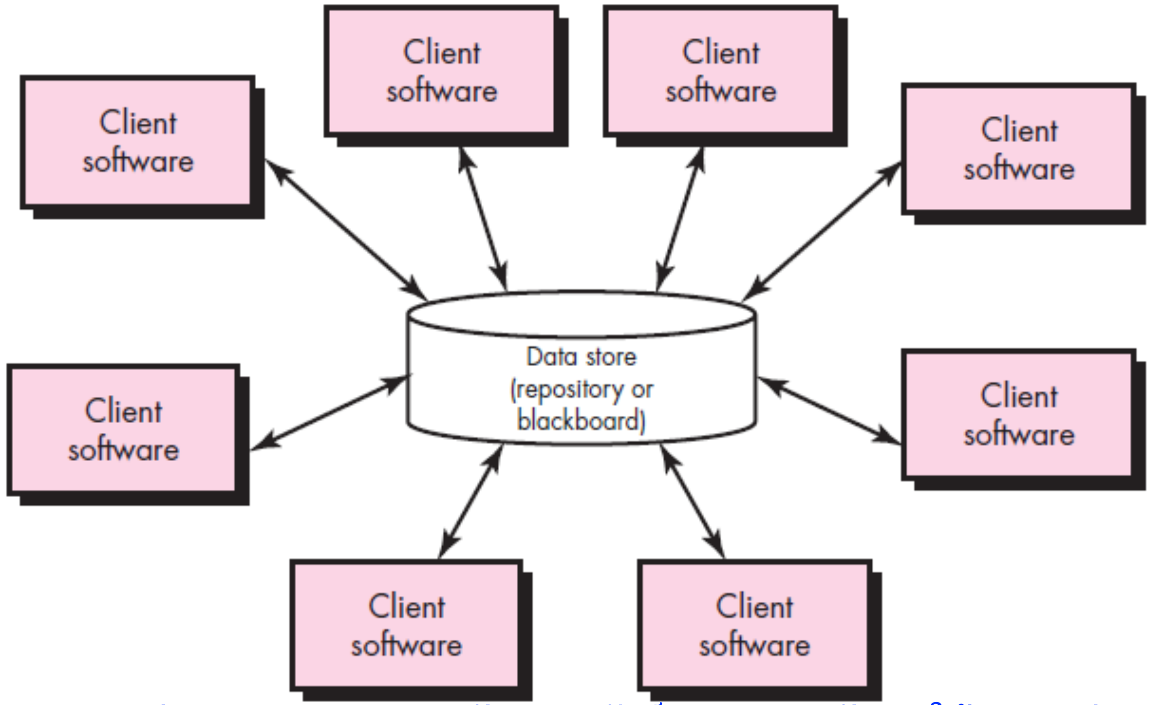
Example of Interrupt-Driven Model



http://ifs.host.cs.st-andrews.ac.uk/Books/SE9/Web/Architecture/web-images/InterruptControl.jpg

Repository/Data-Centered Architecture

- A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. เหมาะกับระบบที่มี data ตรงกลาง



มี Component หลายตัว แต่ละตัวเป็นอิสระจากกัน แต่ใช้ Data เดียวกัน

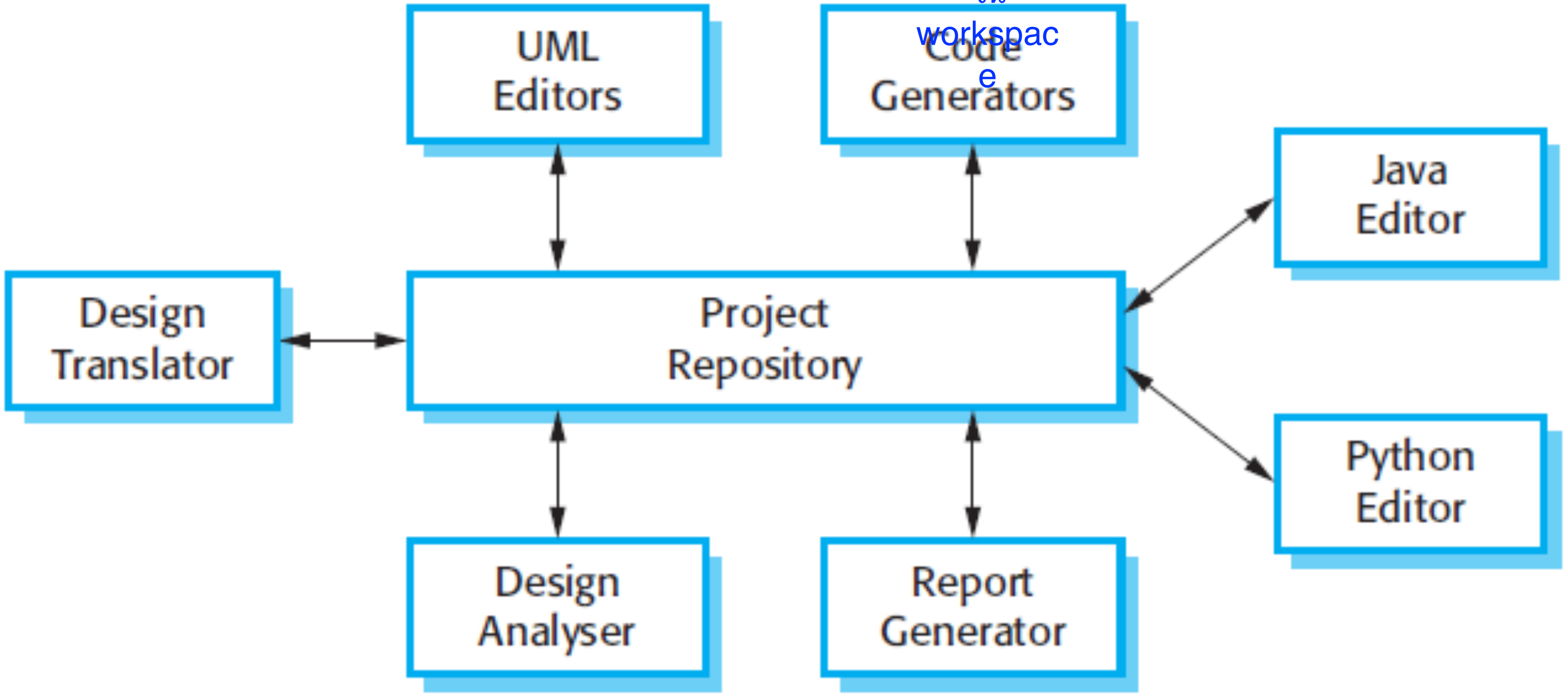
Figure from R.S. Pressman, Software Engineering: A Practitioner's Approach , 7th ed.

Repository/Data-Centered Architecture

- 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.
- This architecture should be 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.

Example

IDE กับ file ของโค้ดที่เก็บใน workspace



ข้อเสียคือใช้พร้อมกันไม่ได้ อาจเกิดการทับซ้อน

Figure from I. Sommerville, Software Engineering, 9th ed.

Repository/Data-Centered Architecture

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.
- May be inefficiencies in organizing all communication through the repository.
- Distributing the repository across several computers may be difficult.

Object-Oriented Architectures

- The components of a system encapsulate data and the operations that must be applied to manipulate the data.
- Communication and coordination between components are accomplished via message passing.

Layered architectures

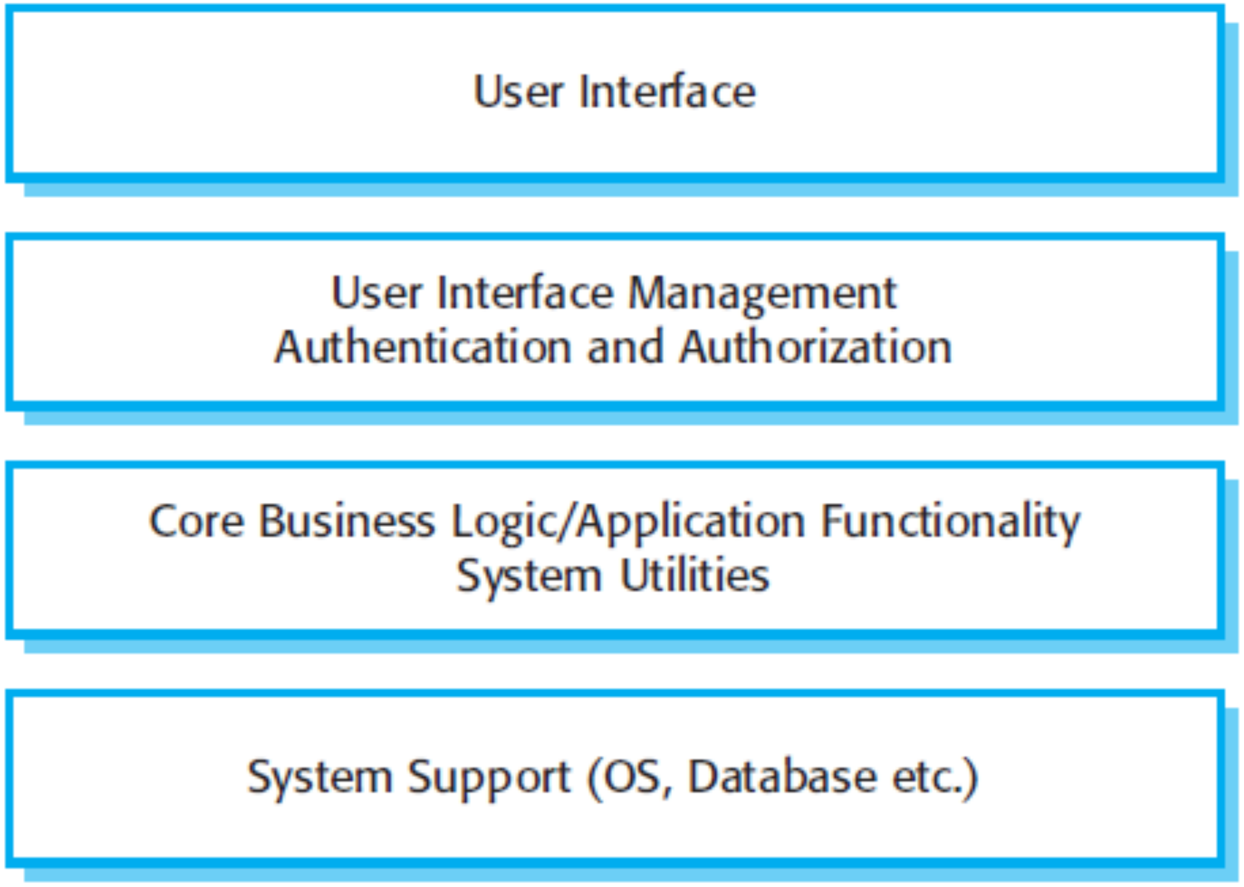

User Interface
User Interface Management
Authentication and Authorization
Core Business Logic/Application Functionality
System Utilities
System Support (OS, Database etc.)

Figure from I. Sommerville, Software Engineering, 9th ed.

Layered architectures

- 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.
- A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set.
- At the outer layer, components service user interface operations.
- At the inner layer, components perform operating system interfacing. Intermediate layers provide utility services and application software functions.

Layered architectures

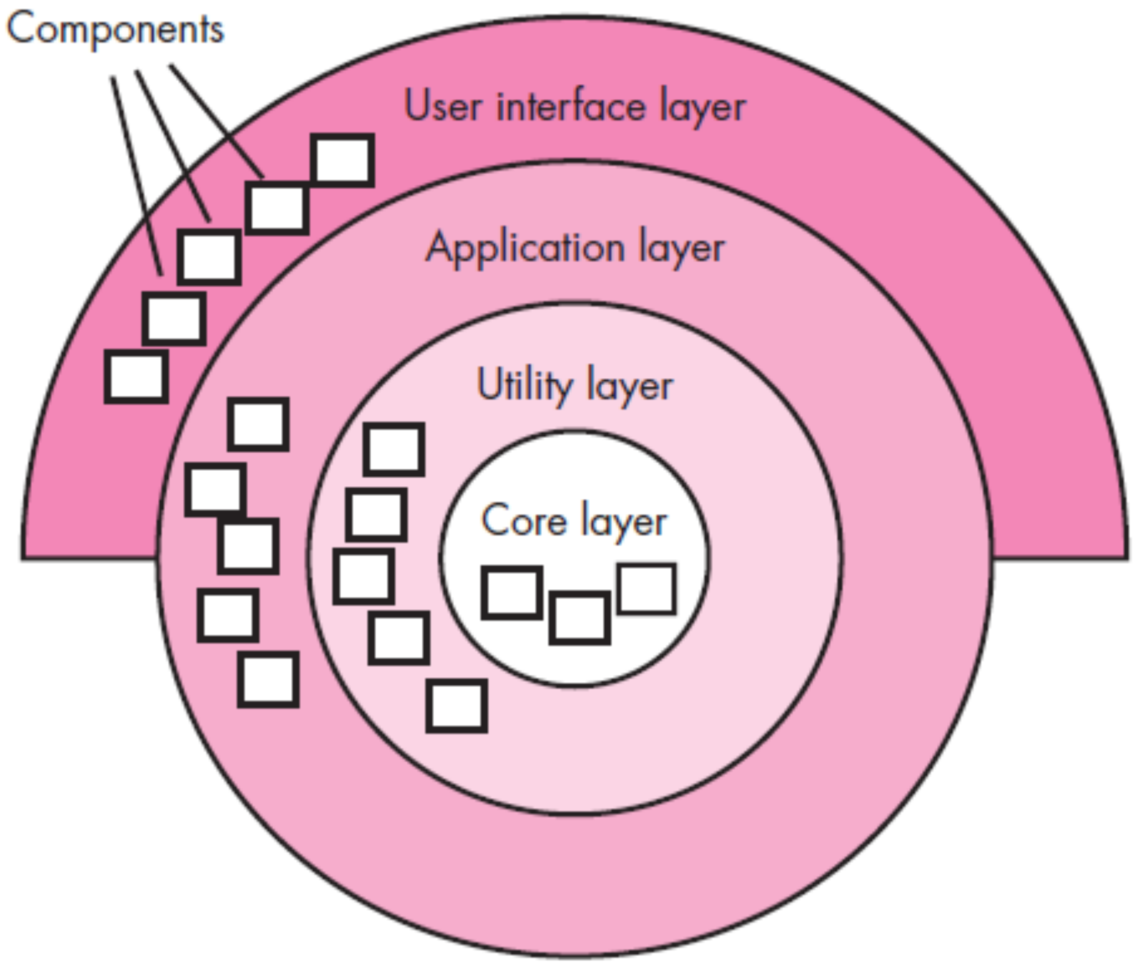


Figure from R.S. Pressman, Software Engineering: A Practitioner's Approach , 7th ed.

Layered architectures

- Used when:
  - building new facilities on top of existing systems;
  - the development is spread across several teams with each team responsibility for a layer of functionality;
  - there is a requirement for multi-level security;
  - multi-platform implementations of an application system is required (Only the inner, machine-dependent layers need be re-implemented to take account of the facilities of a different operating system or database).

Example

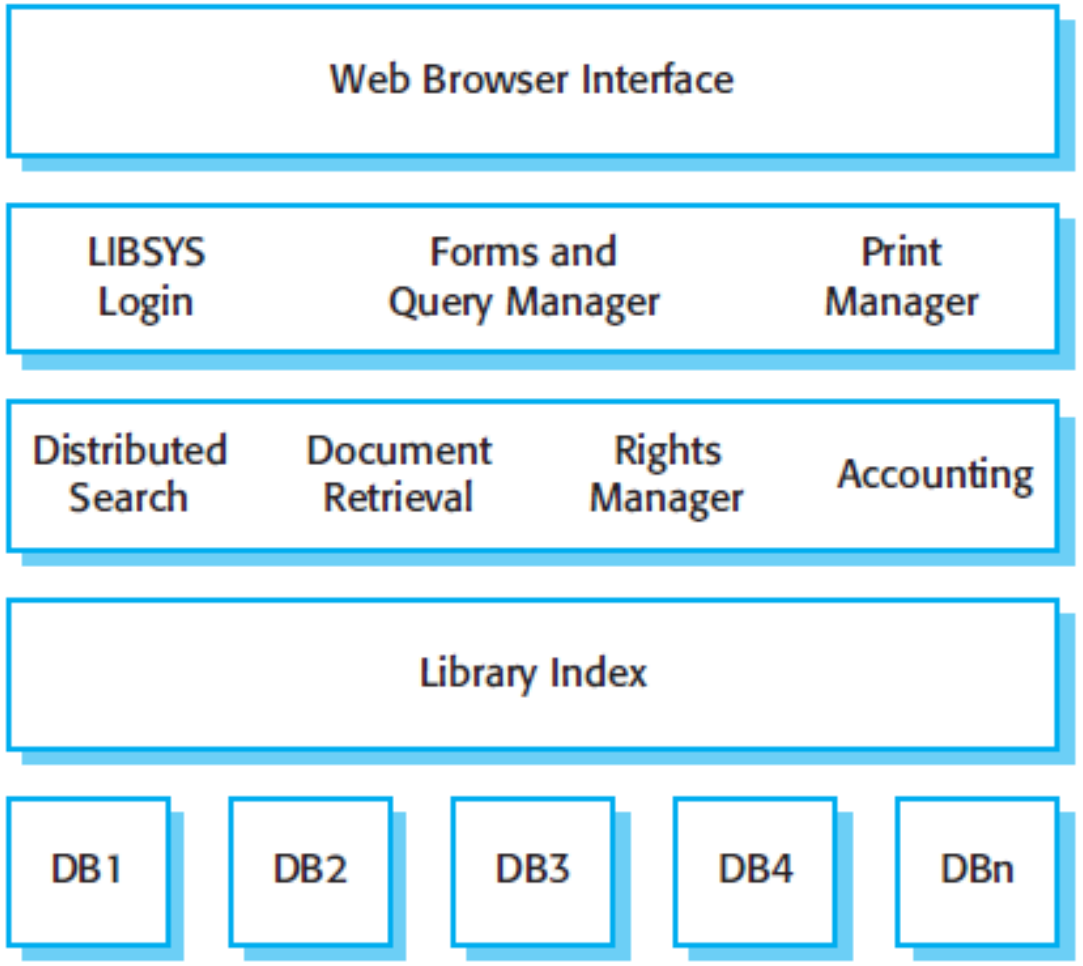


Figure from I. Sommerville, Software Engineering, 9th ed.

Example

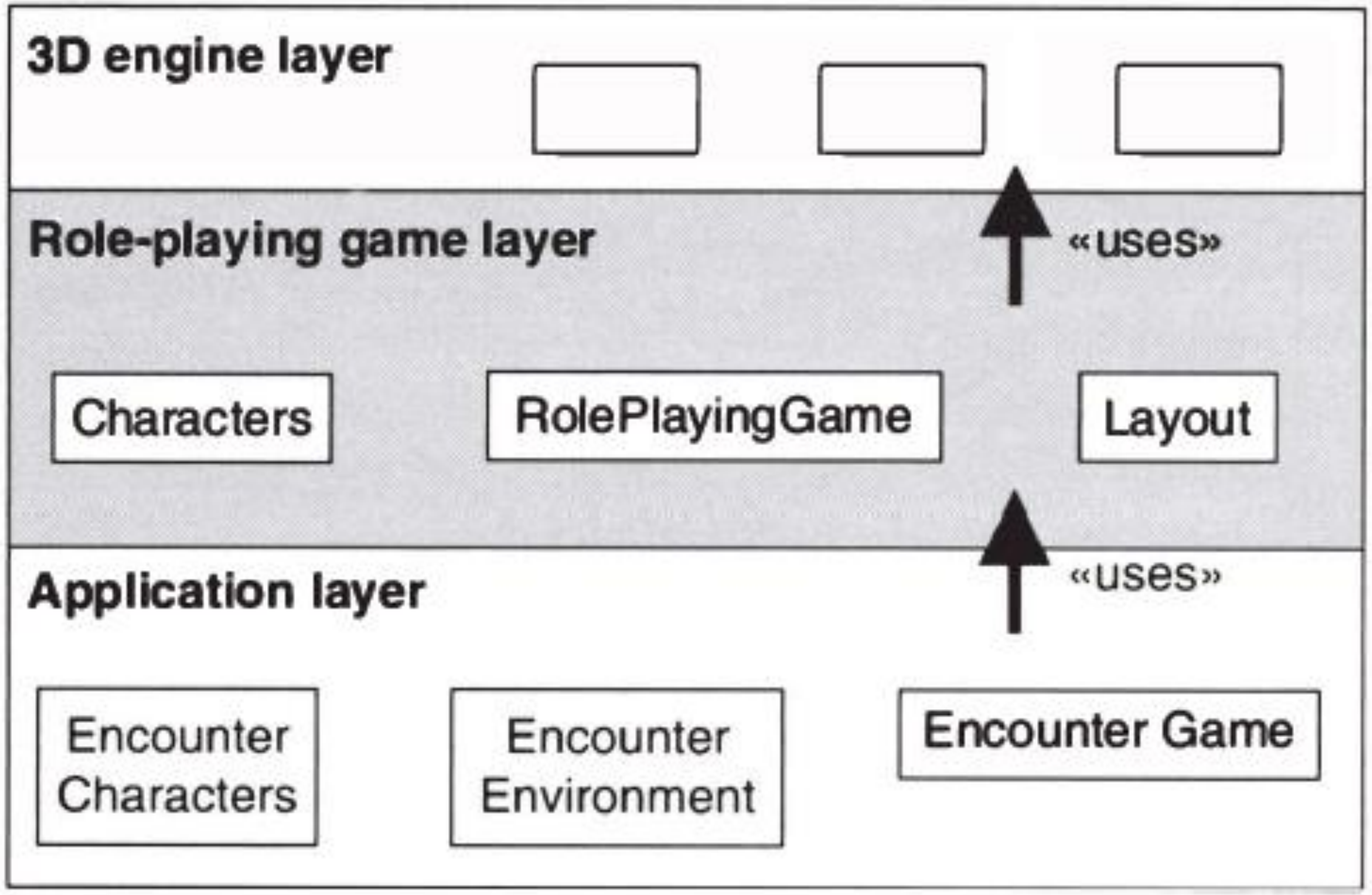


Figure from E. J. Braude and M. E. Bernstein, “Software Engineering: Modern Apppproahces”, 2nd ed., 2016

Layered architectures

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.
- Supports the incremental development of systems.
- The architecture is changeable and portable.

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.

Model-View-Controller (MVC)

- This pattern is the basis of interaction management in many web-based systems.
- 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)

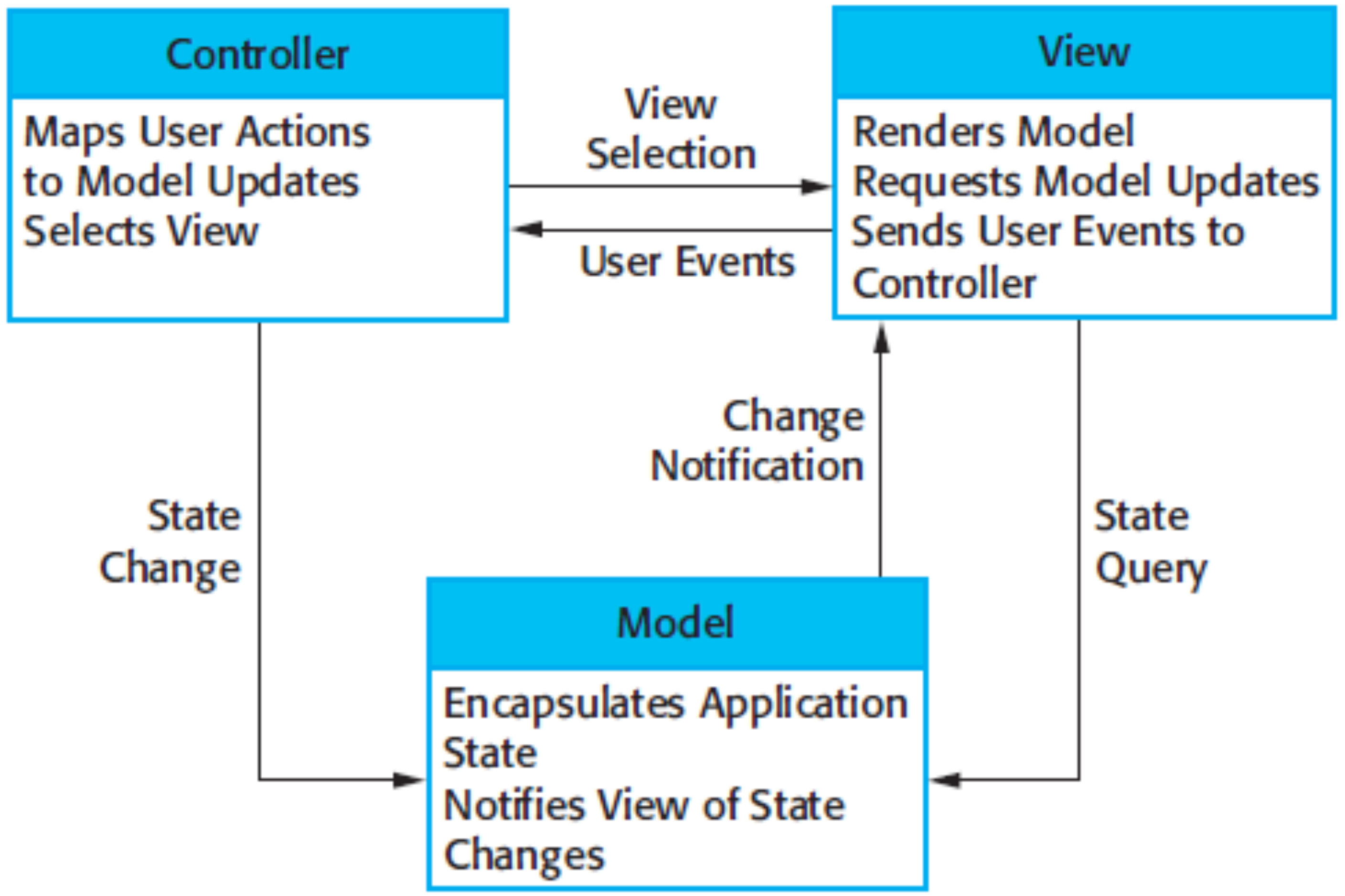


Figure from I. Sommerville, Software Engineering, 9th ed.

Model-View-Controller (MVC)

- Used for implementing user interfaces.
- Used when there are multiple ways to view and interact with data.
- Used when the future requirements for interaction and presentation of data are unknown.
- Used when you want to separate internal representations of information from the ways that information is presented to or accepted from the user.

Example

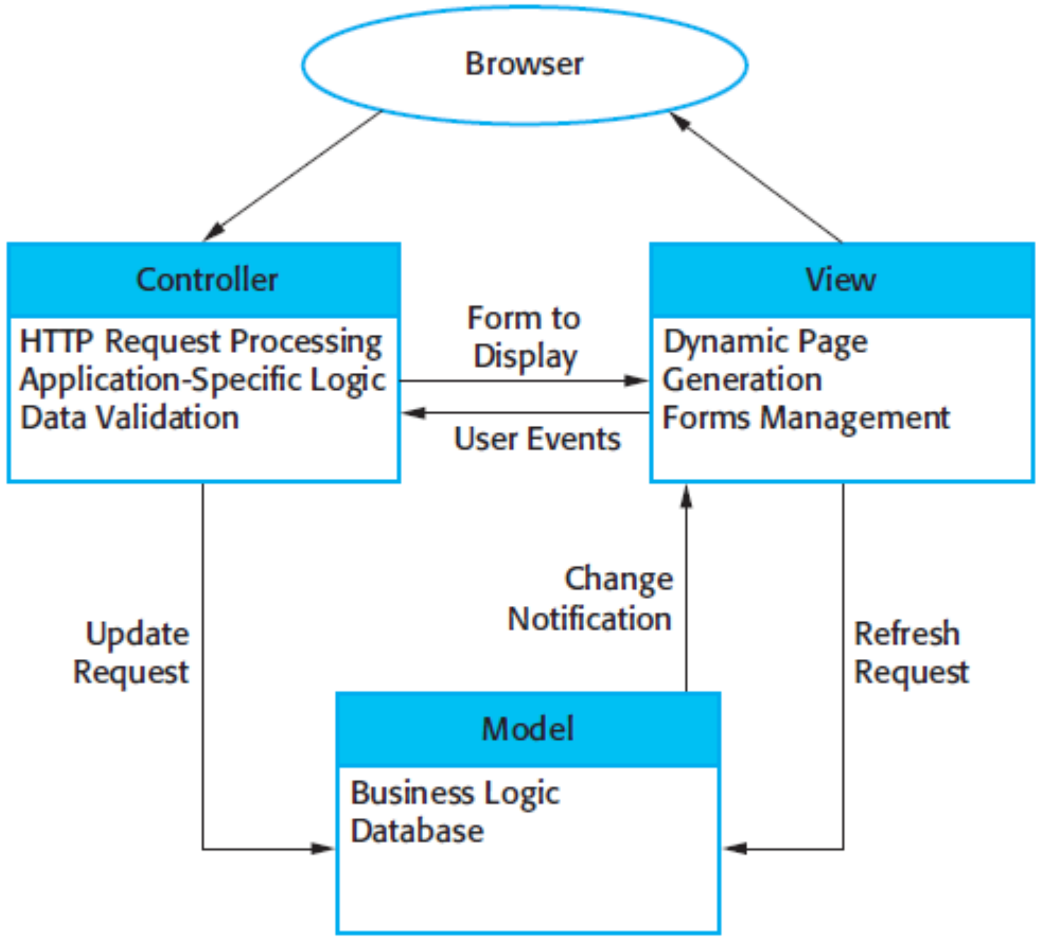


Figure from I. Sommerville, Software Engineering, 9th ed.

Model-View-Controller (MVC)

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.

Architectural Views

- Krutchen 4+1 view model of software architecture, suggests that there should be four fundamental architectural views, which are related using use cases or scenarios
- **Logical view** - shows the key abstractions in the system as objects or object classes
- **Process view** - shows how, at run-time, the system is composed of interacting processes
- **Development view** - shows how the software is decomposed into components that are implemented by a single developer or development team
- **Physical view** - shows the system hardware and how software components are distributed across the processors in the system

Example: Transaction Processing Systems

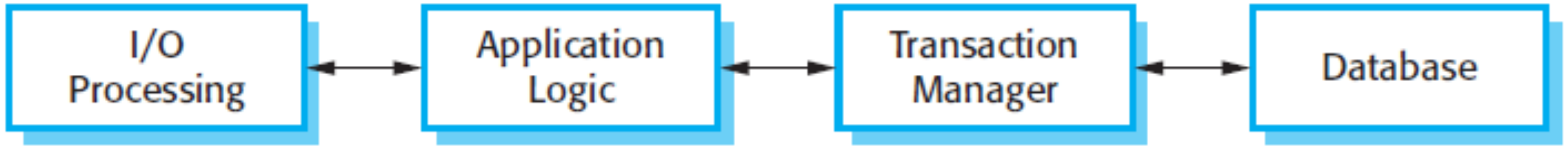


Figure from I. Sommerville, Software Engineering, 9th ed.

Example: Transaction Processing Systems

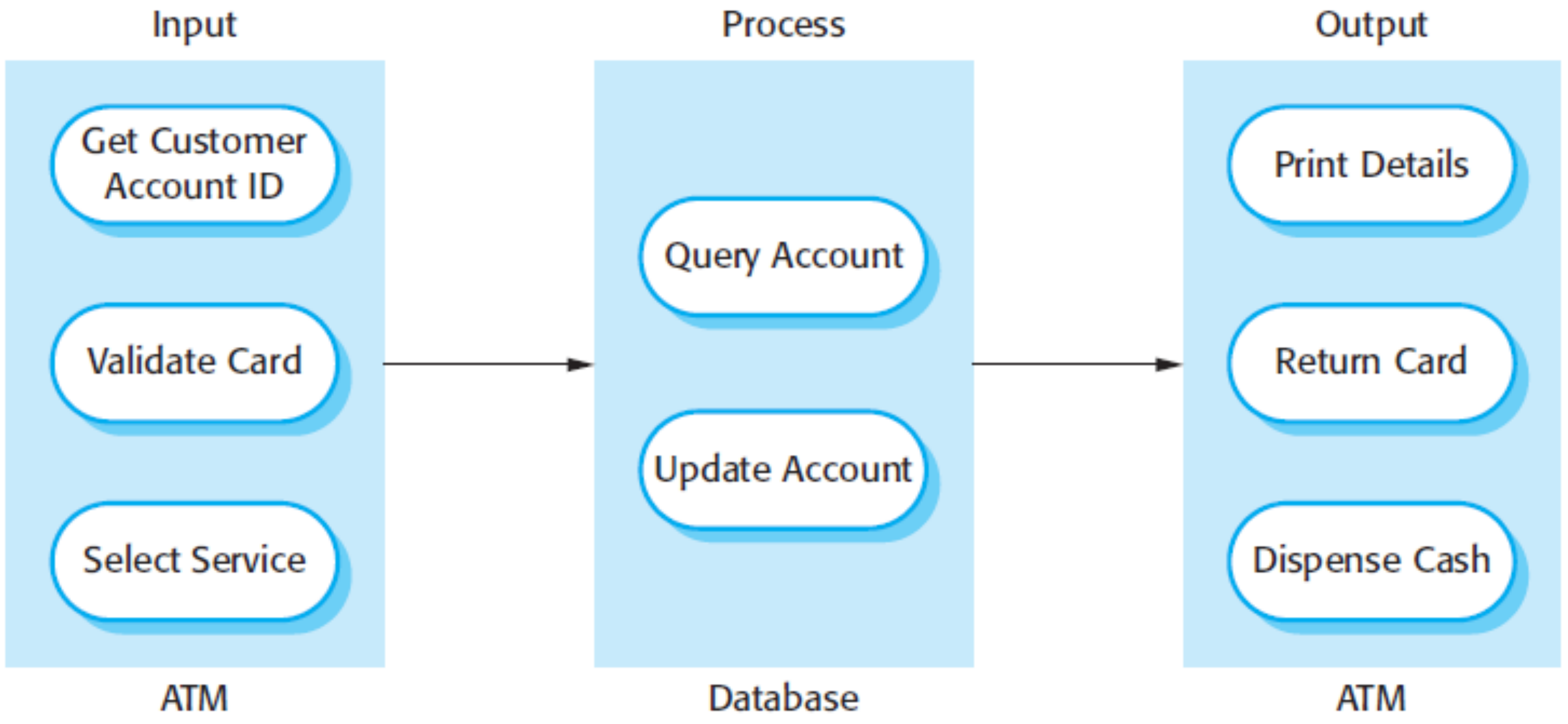


Figure from I. Sommerville, Software Engineering, 9th ed.

Example: Layered Information System Architecture

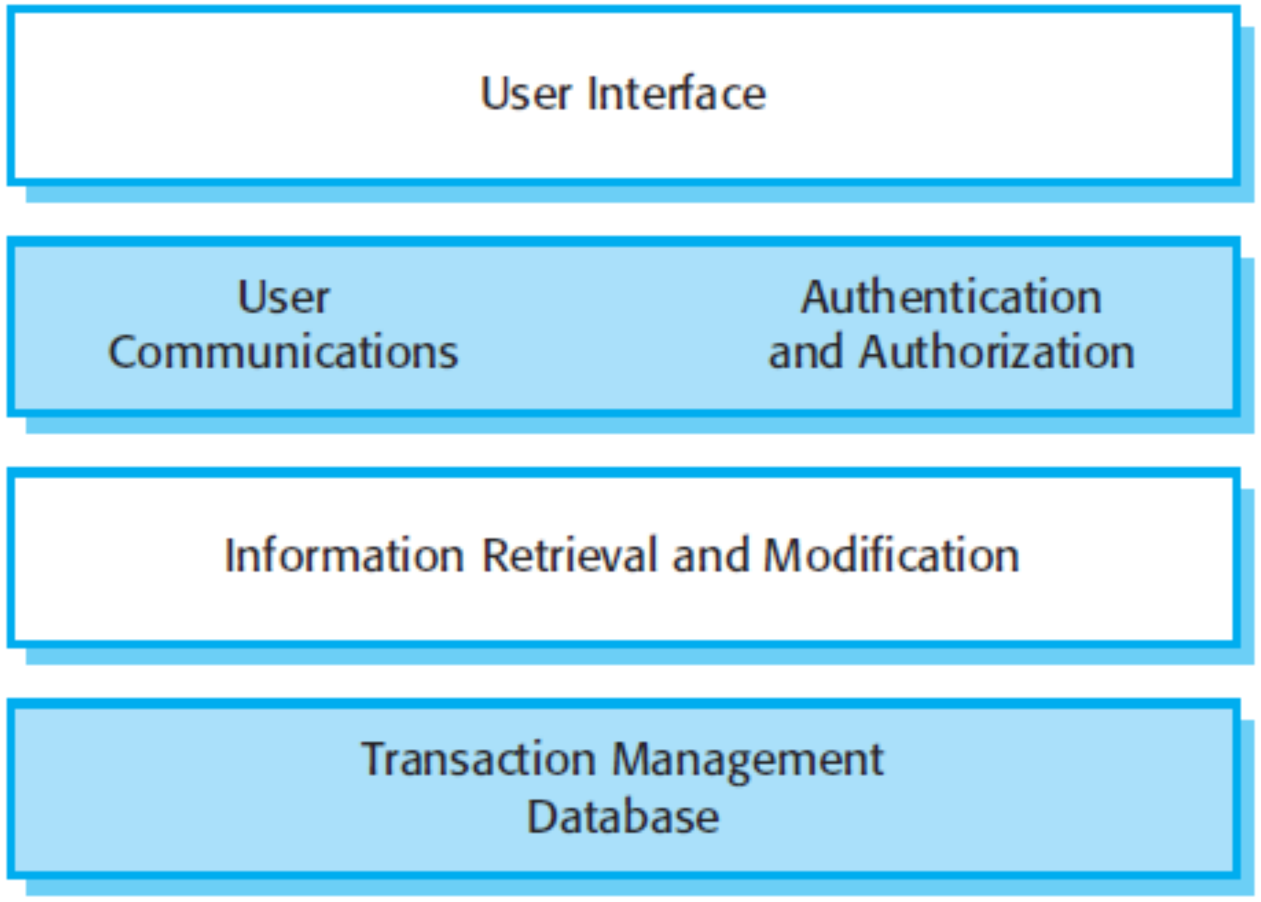

User Interface
User Communications
Authentication and Authorization
Information Retrieval and Modification
Transaction Management Database

Figure from I. Sommerville, Software Engineering, 9th ed.

Example: Layered Information System Architecture

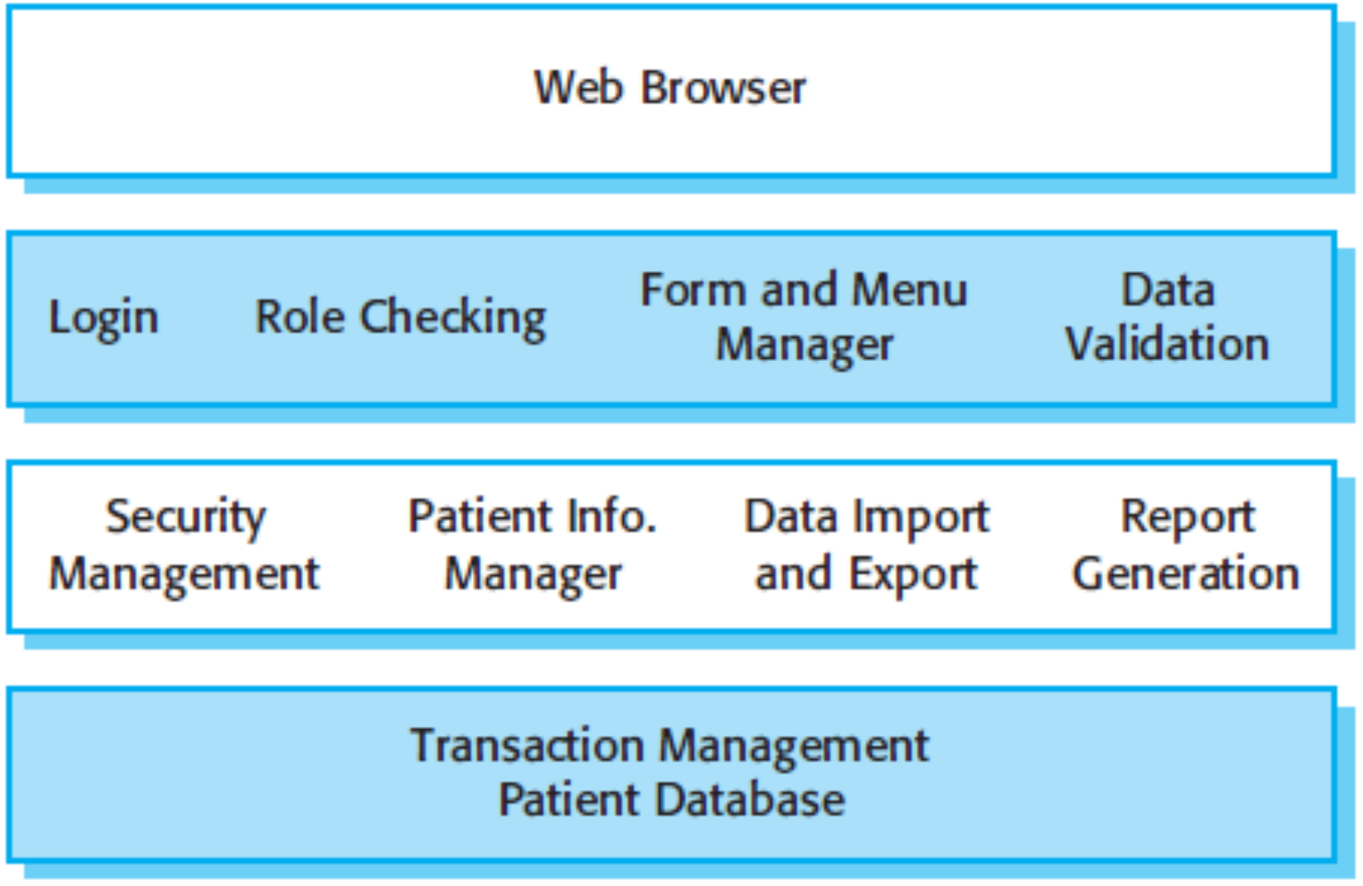


Figure from I. Sommerville, Software Engineering, 9th ed.