Client–Server Computing

Week 7: Architectural Design, Part 2

Edmund Yu, PhD
Associate Professor
esyu@syr.edu

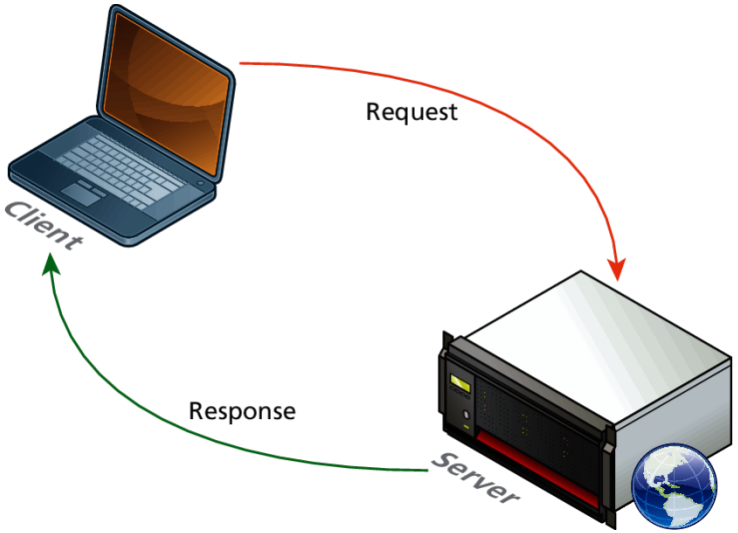
SYRACUSE UNIVERSITY
ENGINEERING & COMPUTER SCIENCE

Client–Server Computing

- Distributed systems that are accessed over the Internet are normally organized as **client–server systems**.
  - In a client–server system, the user interacts with a program running on his local computer such as a web browser (the client), which interacts with another program running on a remote computer, such as a web server (the server).
  - The remote computer provides services, such as access to web pages, which are available to the clients.

Client–Server Computing

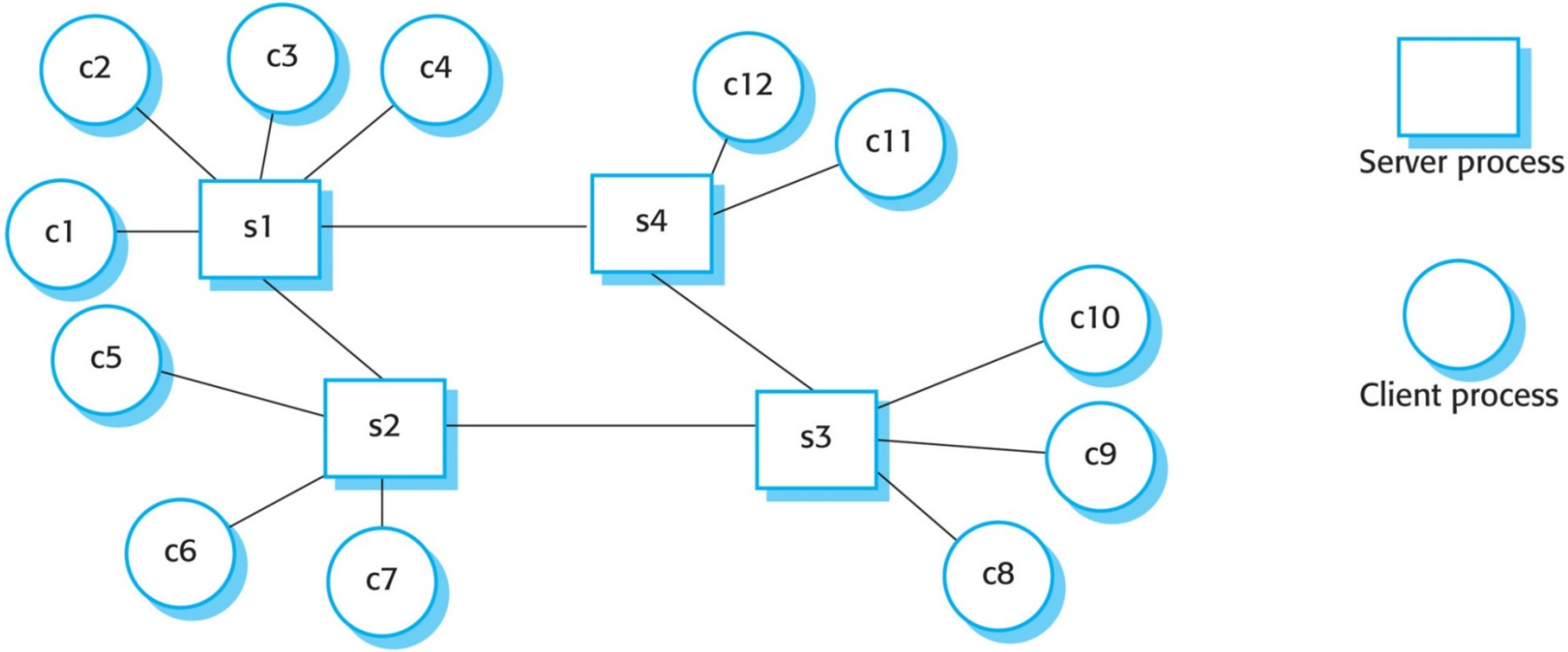
- More formally, the client initiates a **request** to a server and gets a **response** that could include some resource like an HTML file, an image, or some other data.



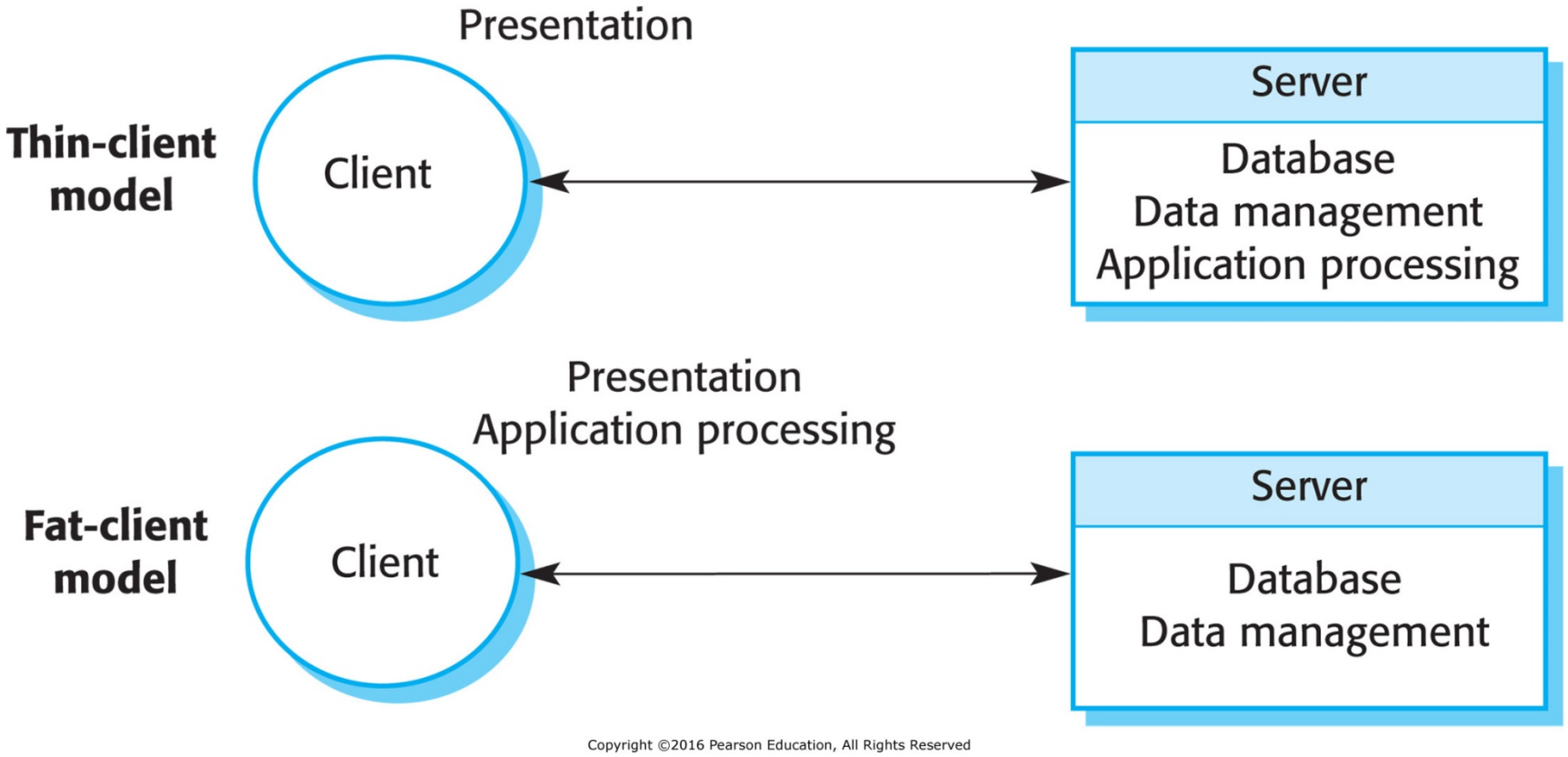
- The **request-response loop** is the most basic mechanism on the server for receiving requests and transmitting data in response.

2-Tier Client–Server Systems

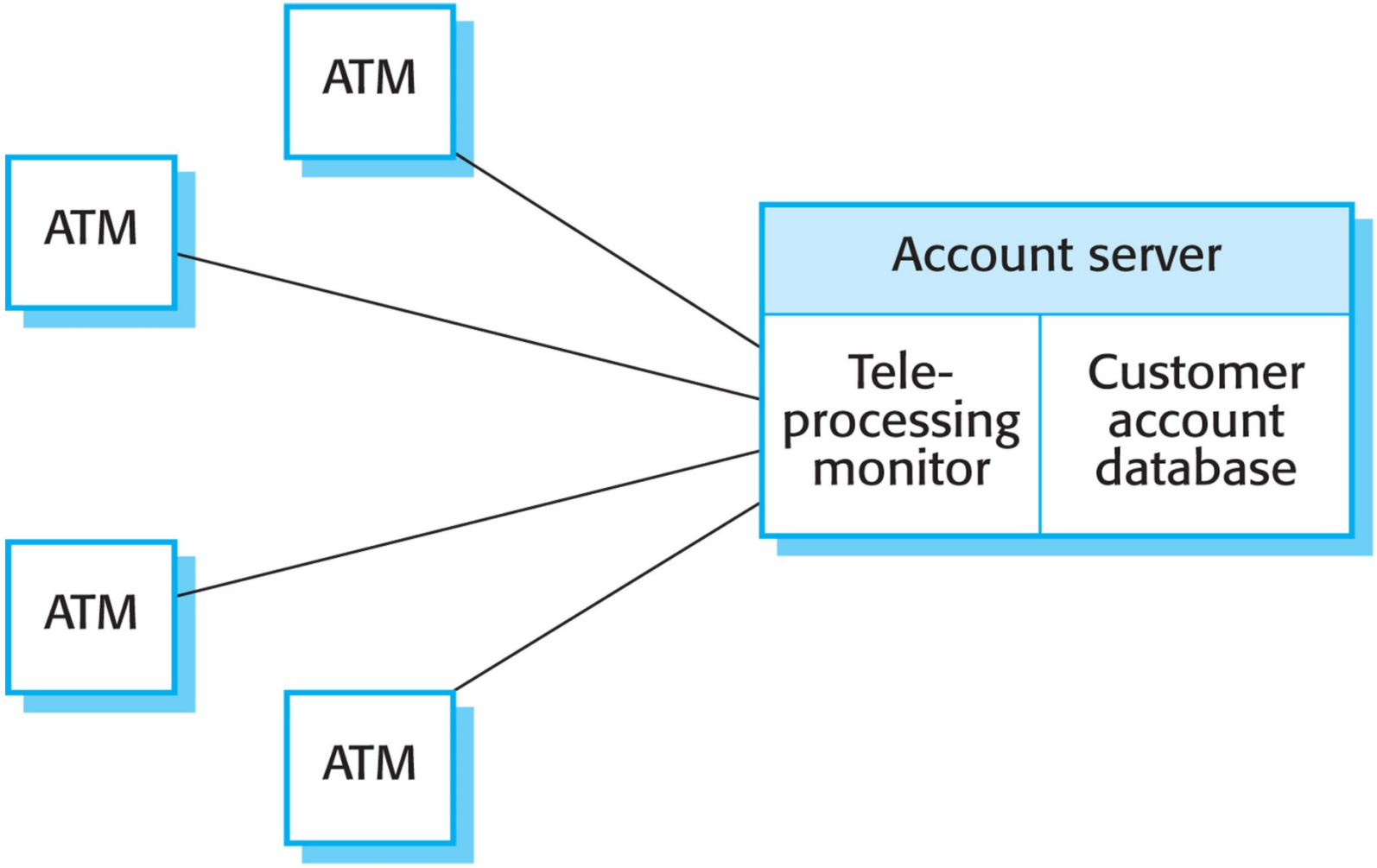
- In a two-tier client–server system, the system is implemented as a single logical server plus an indefinite number of clients that use that server.



Thin- and Fat-Client Architectural Models

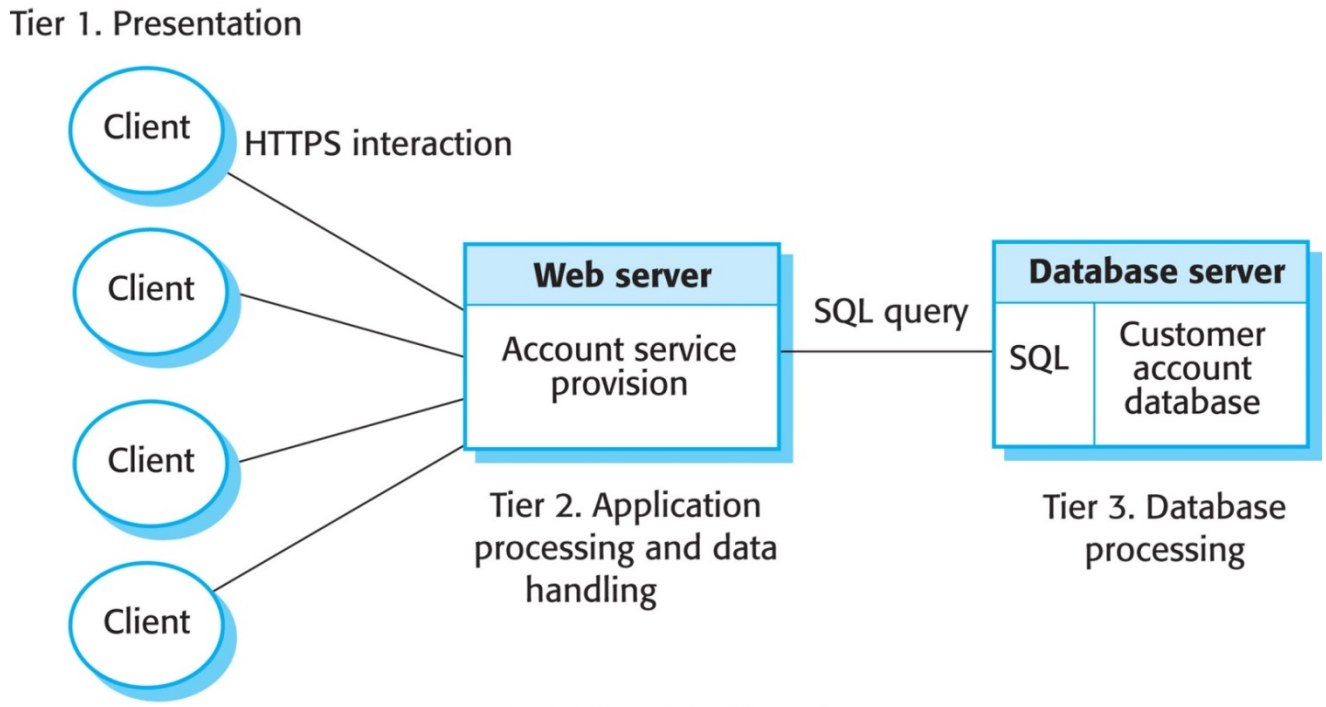


A Fat-Client Architecture for an ATM System



Multitier CS Architectures

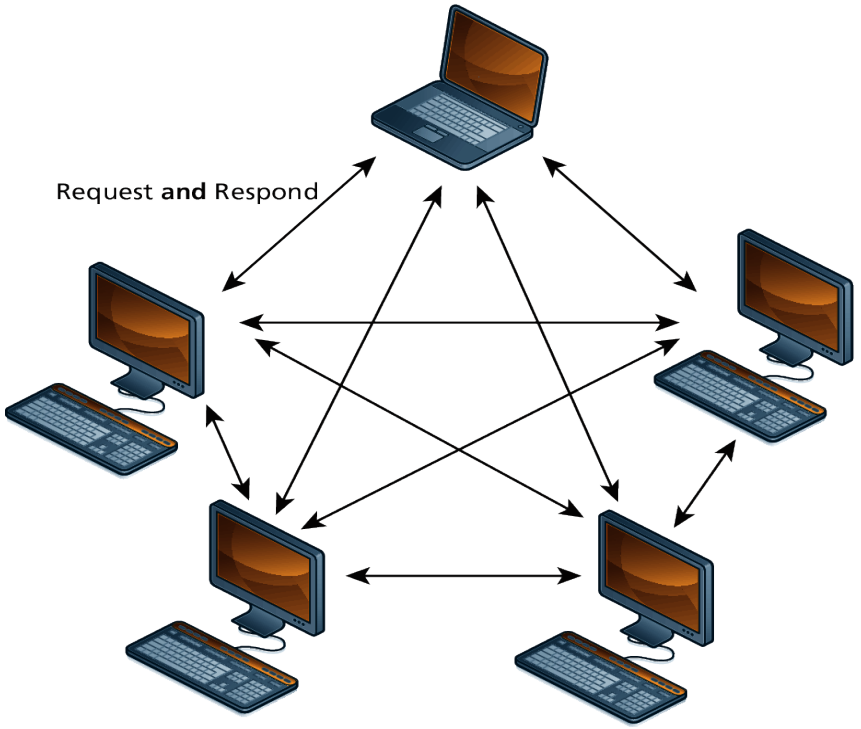
- In a “multitier client–server” architecture, the different layers of the system, namely presentation, application processing, data management,, and database, are separate processes that may execute on different processors.



The Peer-to-Peer Alternative

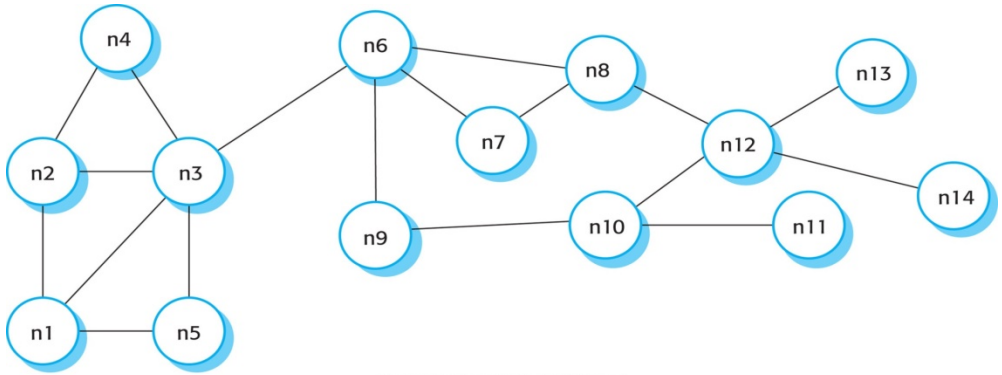
- In the **peer-to-peer model**, where each computer is functionally identical, each node is able to send and receive directly with one another.

- In such a model each peer acts as both a client and server able to upload and download information.



Peer-to-Peer Systems

- More formally, peer-to-peer (p2p) systems are decentralized distributed systems where computations may be carried out by any node (machine) in the network.
  - The overall system is designed to take advantage of the computational power and storage of a large number of networked computers.
  - Most p2p systems have been personal systems, but there is increasing business use of this technology.



Peer-to-Peer Systems

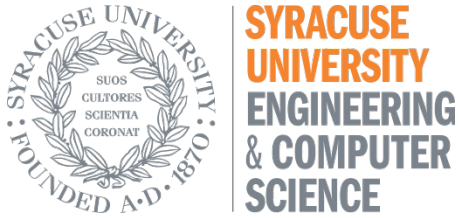
- It is appropriate to use a peer-to-peer architectural model for a system in two circumstances:
  1. The system is **computationally intensive**, and it is possible to separate the processing into a large number of independent computations.
     - E.g., computational drug discovery system.
       - Each node checks a separate molecule to see if it has the characteristics required to suppress the growth of cancers.
  2. The system primarily involves the **exchange of information** between individual computers/processors.
     - E.g., file-/music-/video-sharing systems (gnutella, torrents, etc.), voice/video communication systems (Skype).

SYRACUSE UNIVERSITY
ENGINEERING & COMPUTER SCIENCE

Distributed System Design Issues

Week 7: Architectural Design, Part 2

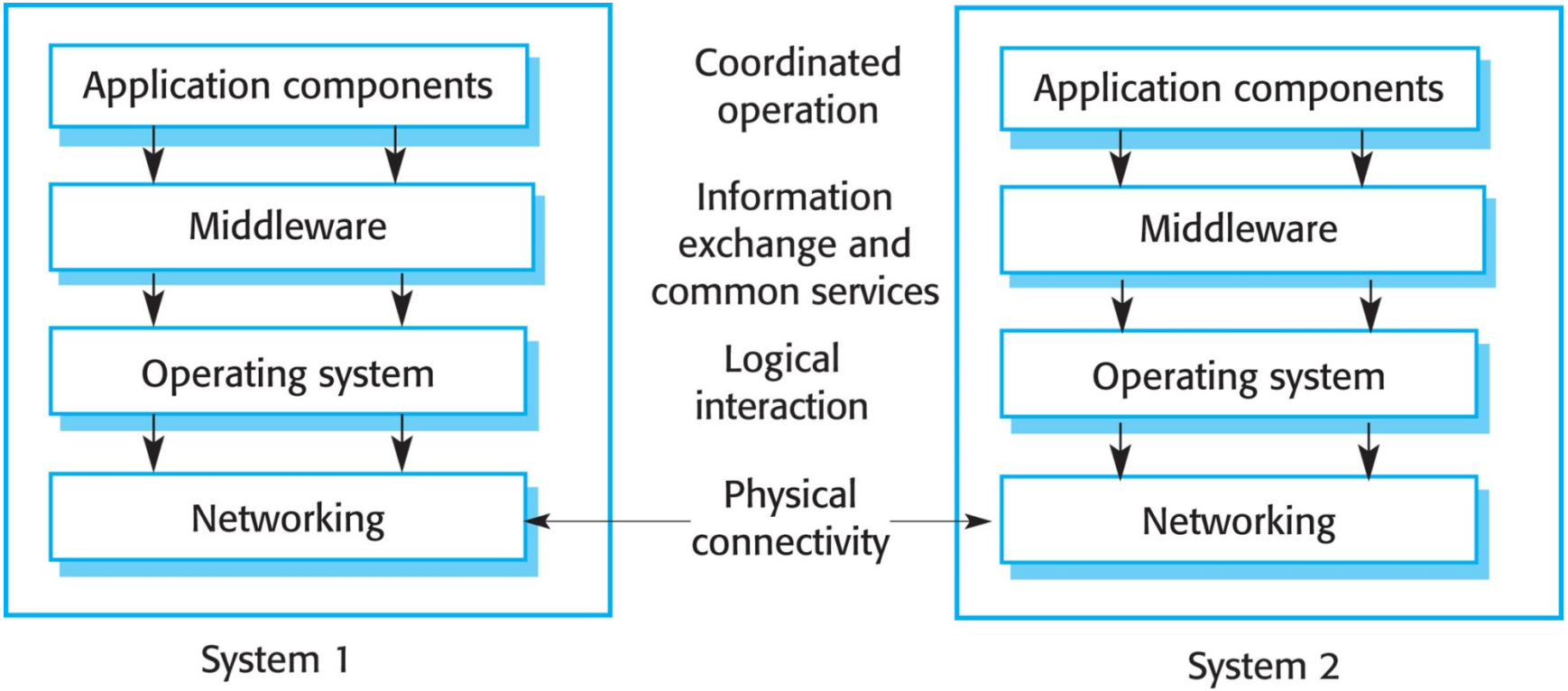
Edmund Yu, PhD
Associate Professor
esyu@syr.edu



DS Issue 1: Transparency

- Ideally, users should not be aware that a system is distributed, and services should be independent of distribution characteristics.
  - In practice, this is impossible because parts of the system are independently managed and because of network delays.
    - Often better to make users aware of distribution so that they can cope with problems.
  - To achieve transparency, resources should be abstracted and addressed logically rather than physically.
    - Middleware maps logical to physical resources.

Middleware in a Distributed System



DS Issue 2: Openness

- Open distributed systems are systems that are built according to generally accepted standards.
  - Components from any supplier can be integrated into the system and can interoperate with the other system components.
  - Openness implies that system components can be independently developed in any programming language, and, if these conform to standards, they will work with other components.
  - Web service standards for service-oriented architectures were developed to be open standards.

DS Issue 3: Scalability

- The scalability of a system reflects its ability to deliver a high-quality service as demands on the system increase.
  - **Size**: It should be possible to add more resources to a system to cope with increasing numbers of users.
  - **Distribution**: It should be possible to geographically disperse the components of a system without degrading its performance.
  - **Manageability**: It should be possible to manage a system as it increases in size, even if parts of the system are located in independent organizations.
- There is a distinction between scaling up and scaling-out.
  - Scaling up is more powerful system.
  - Scaling out is more system instances.

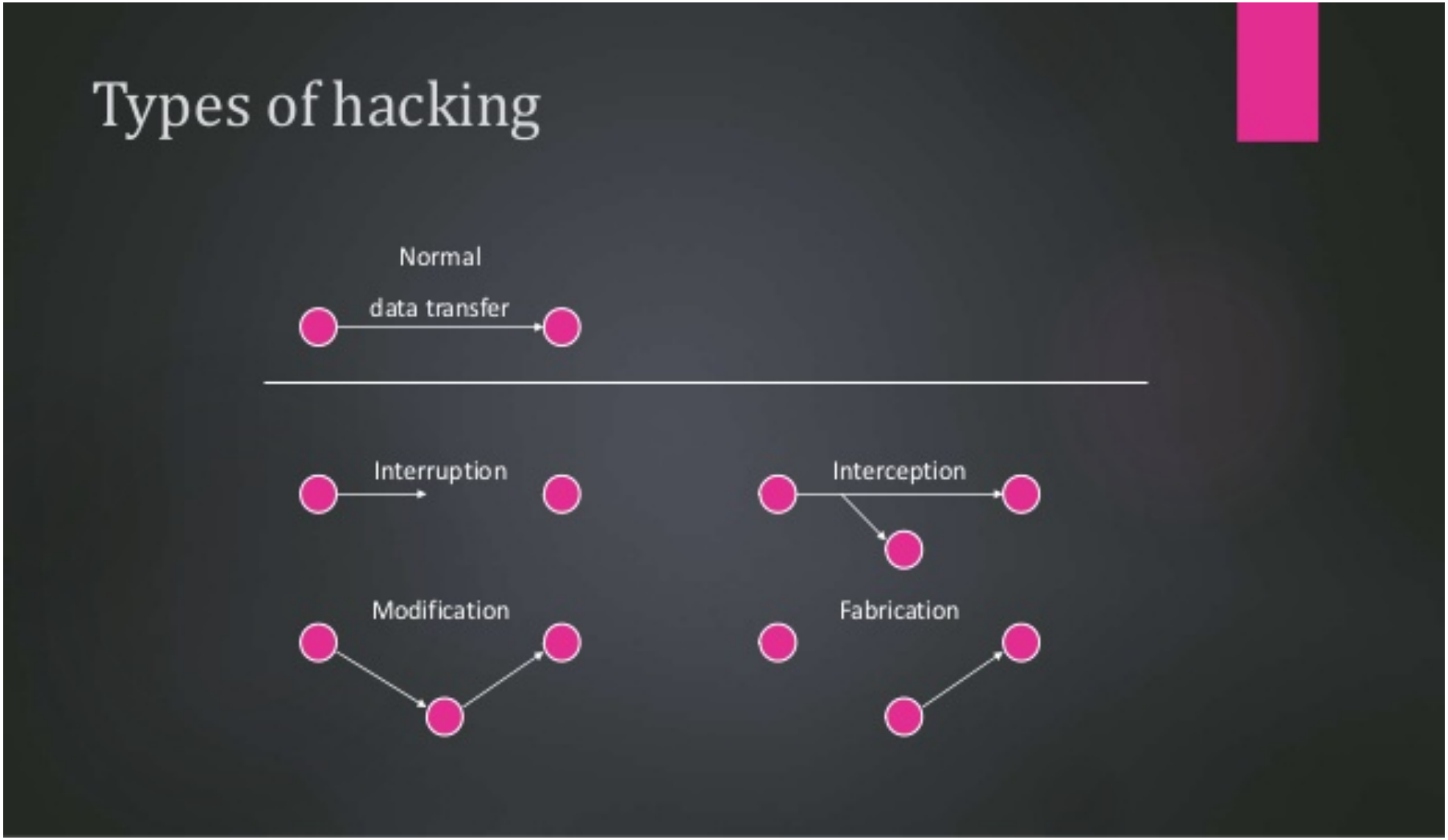
DS Issue 4: Security

- When a system is distributed, the number of ways that the system may be attacked is significantly increased, compared to centralized systems.
  - If a part of the system is successfully attacked then the attacker may be able to use this as a "**back door**" into other parts of the system.
  - Difficulties in a distributed system arise because different organizations may own parts of the system.
    - These organizations may have mutually incompatible security policies and security mechanisms.
    - **Attack the weakest link.**

Types of Attack

- The types of attack that a distributed system must defend itself against are:
  - Interception, where communications between parts of the system are intercepted by an attacker so that there is a loss of confidentiality
  - Interruption, where system services are attacked and cannot be delivered as expected
    - Denial-of-service attacks involve bombarding a node with illegitimate service requests so that it cannot deal with valid requests.
  - Modification, where data or services in the system are changed by an attacker
  - Fabrication, where an attacker generates information that should not exist and then uses this to gain some privileges

Types of Attack



DS Issue 5: Quality of Service

- The quality of service (QoS) offered by a distributed system reflects the system's ability to deliver its services dependably and with a response time and throughput that is acceptable to its users.
  - Quality of service is particularly critical when the system is dealing with **time-critical data** such as sound or video streams.
  - In these circumstances, if the quality of service falls below a threshold value then the sound or video may become so degraded that it is impossible to understand.

DS Issue 6: Failure Management

- In a distributed system, it is inevitable that failures will occur, so the system has to be designed to be resilient to these failures.

  "You know that you have a distributed system when the crash of a system that you've never heard of stops you getting any work done."

  - Leslie Lamport, ***Security Engineering****: A Guide to Building Dependable Distributed Systems*

DS Issue 6: Failure Management

- Distributed systems:
  - Should include mechanisms for discovering if a component of the system has failed
  - Should continue to deliver as many services as possible in spite of that failure
  - Should automatically recover from the failure

SYRACUSE UNIVERSITY
ENGINEERING & COMPUTER SCIENCE

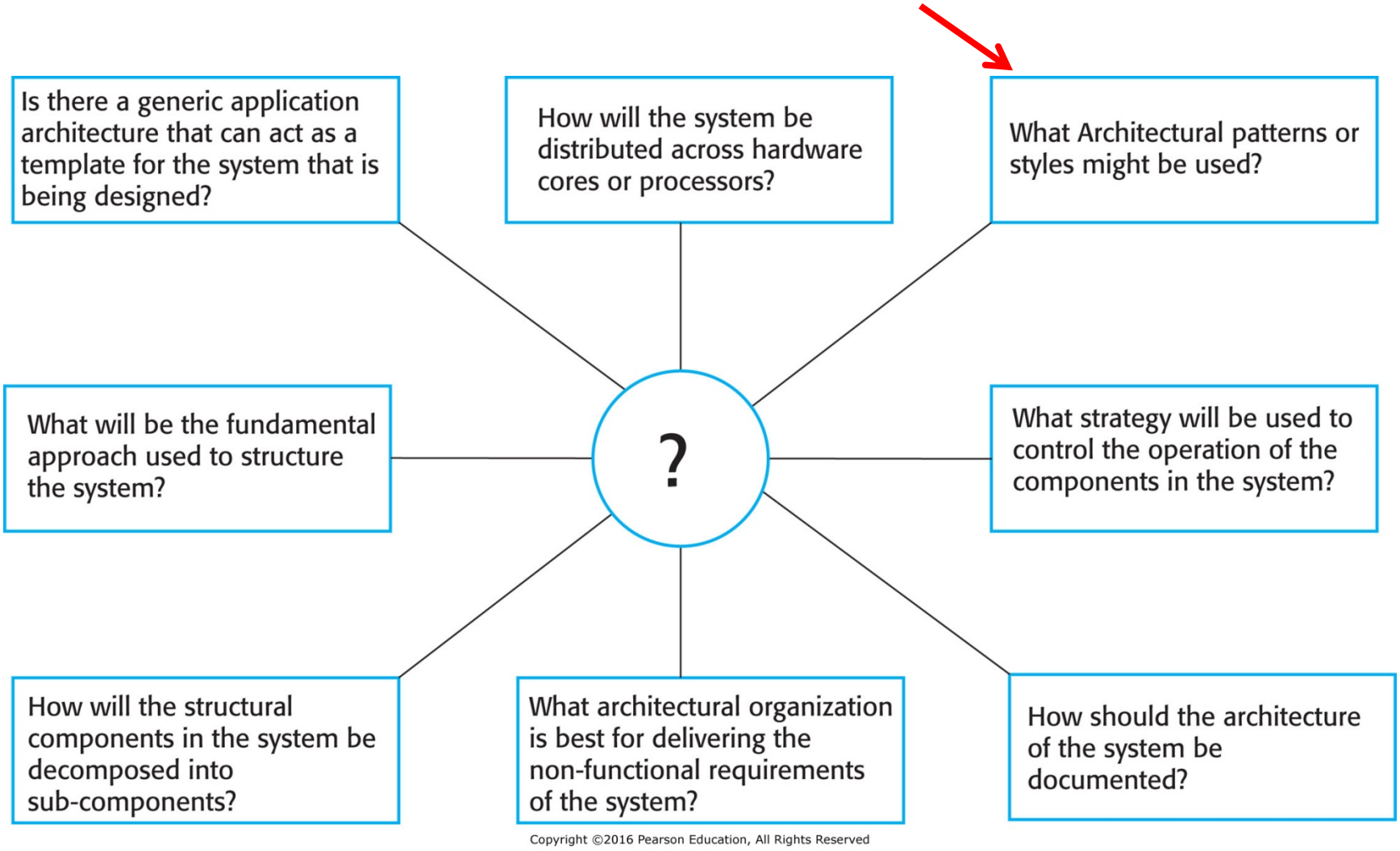
The Client-Server and Layered Patterns

Week 7: Architectural Design, Part 2

Edmund Yu, PhdD
Associate Professor
esyu@syr.edu

SYRACUSE UNIVERSITY
ENGINEERING
& COMPUTER
SCIENCE

Architectural Design Decisions

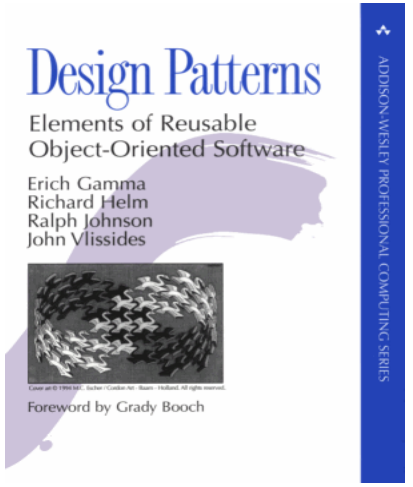


Design Patterns

- Patterns are a means of representing, sharing, and reusing knowledge.
  - Someone has already solved your problems. Why reinvent the wheel?
  - Instead of **code reuse**, with patterns you get **experience reuse**.
  - The best way to use patterns is to **load your brain** with them and then **recognize places** in your designs and existing applications where you can **apply them**.

Gang of Four

- **Erich Gamma, Richard Helm, Ralph Johnson, and John Vlisides,** in their ***Design Patterns*** book, define 23 design patterns divided into three types.
  - **Creational patterns**—create objects for you
  - **Structural patterns**—help you compose groups of objects into larger structures
  - **Behavioral patterns**—help you define and control the communication between objects in your system



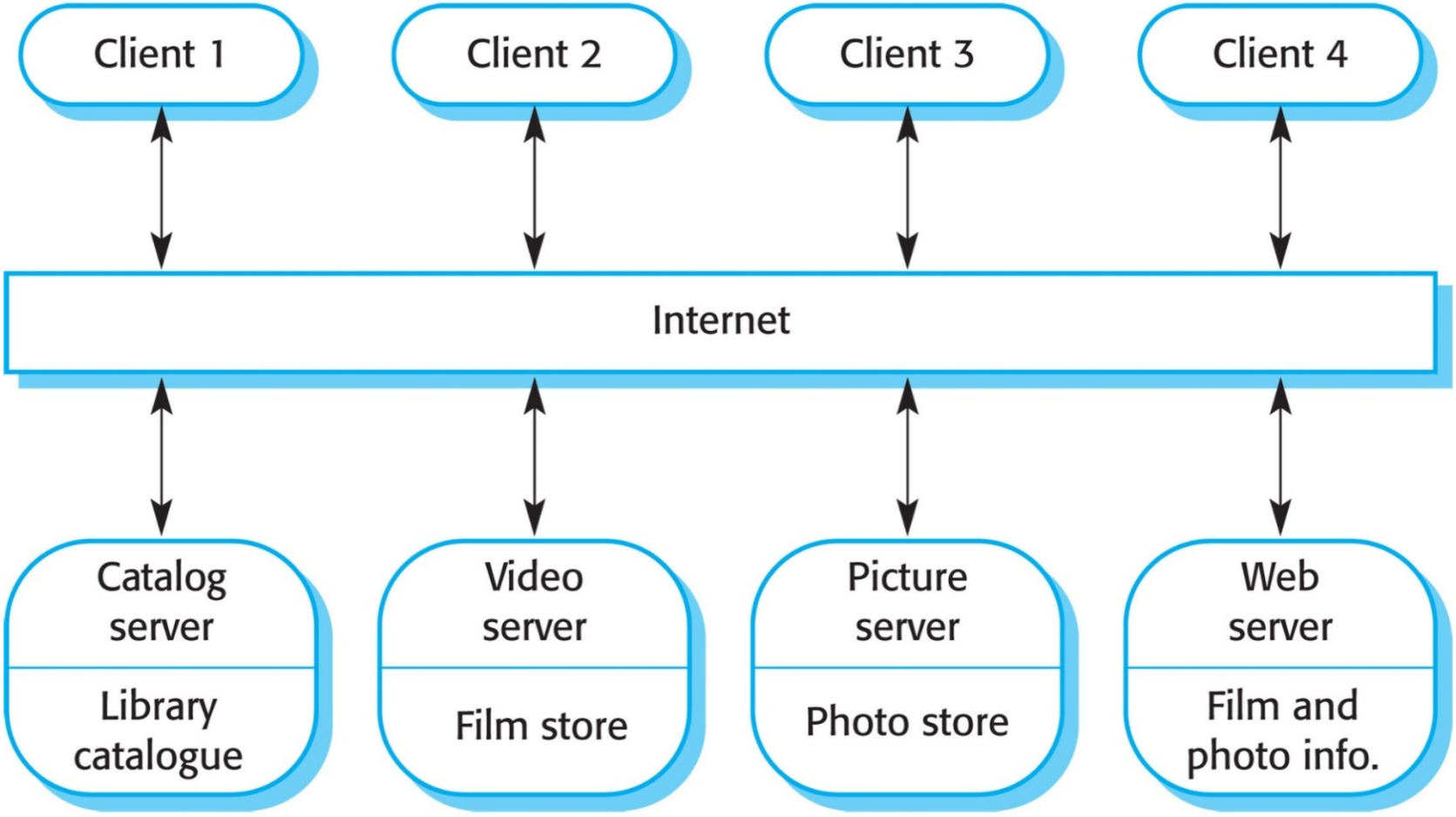
- The original design patterns were intended for lower-level, detailed design, but the same concept can be and have been applied to architectural design, hence architectural patterns.

The Client–Server Pattern

Name	Client–server
Description	In a client–server architecture, the system is presented as a set of services, with each service delivered by a separate server. Clients are users of these services and access servers to make use of them.
Example	Figure 6.13 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 and so is susceptible to denial-of-service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system. Management problems may arise if servers are owned by different organizations.

The CS Pattern: An Example

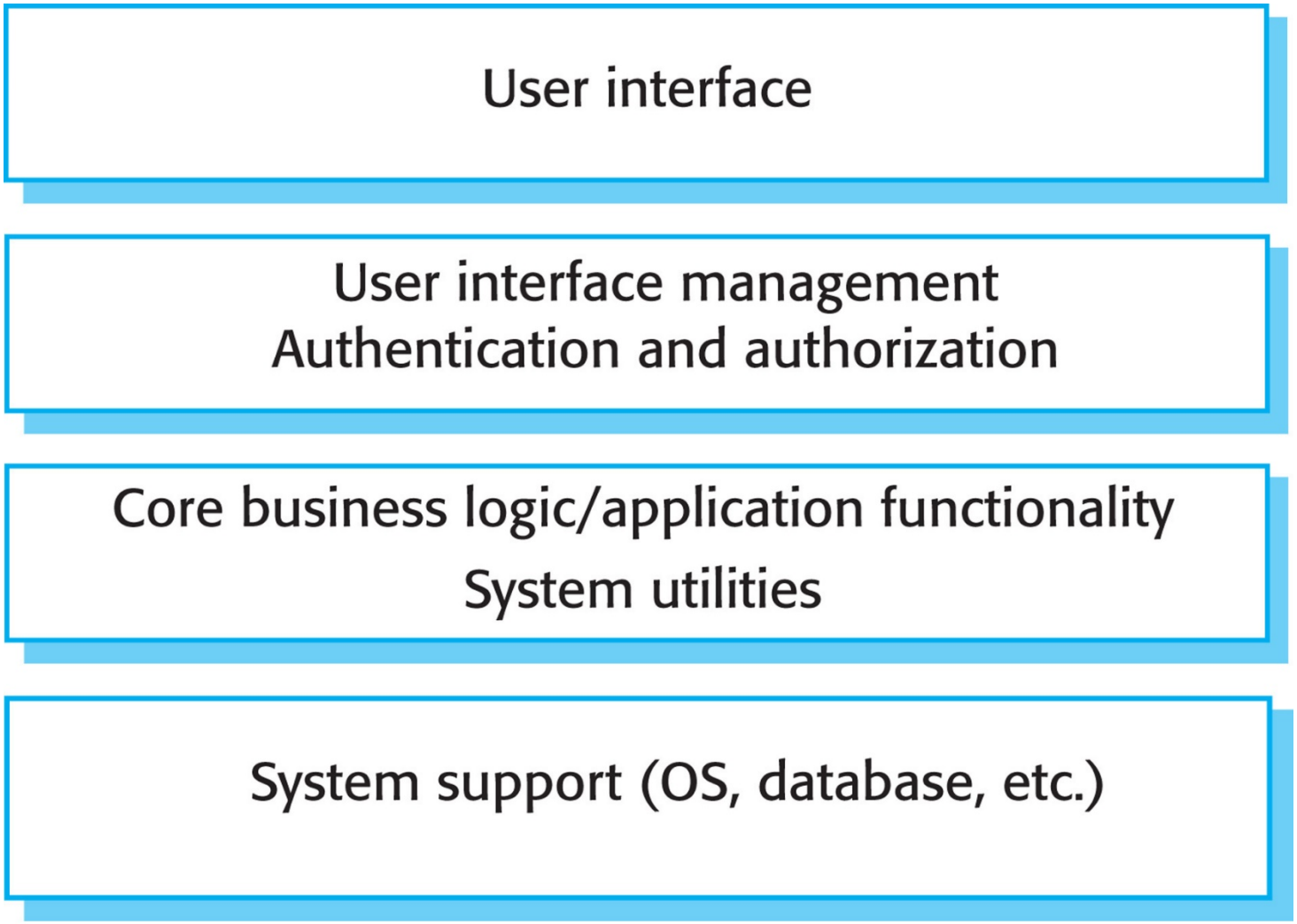
A client–server architecture for a film library



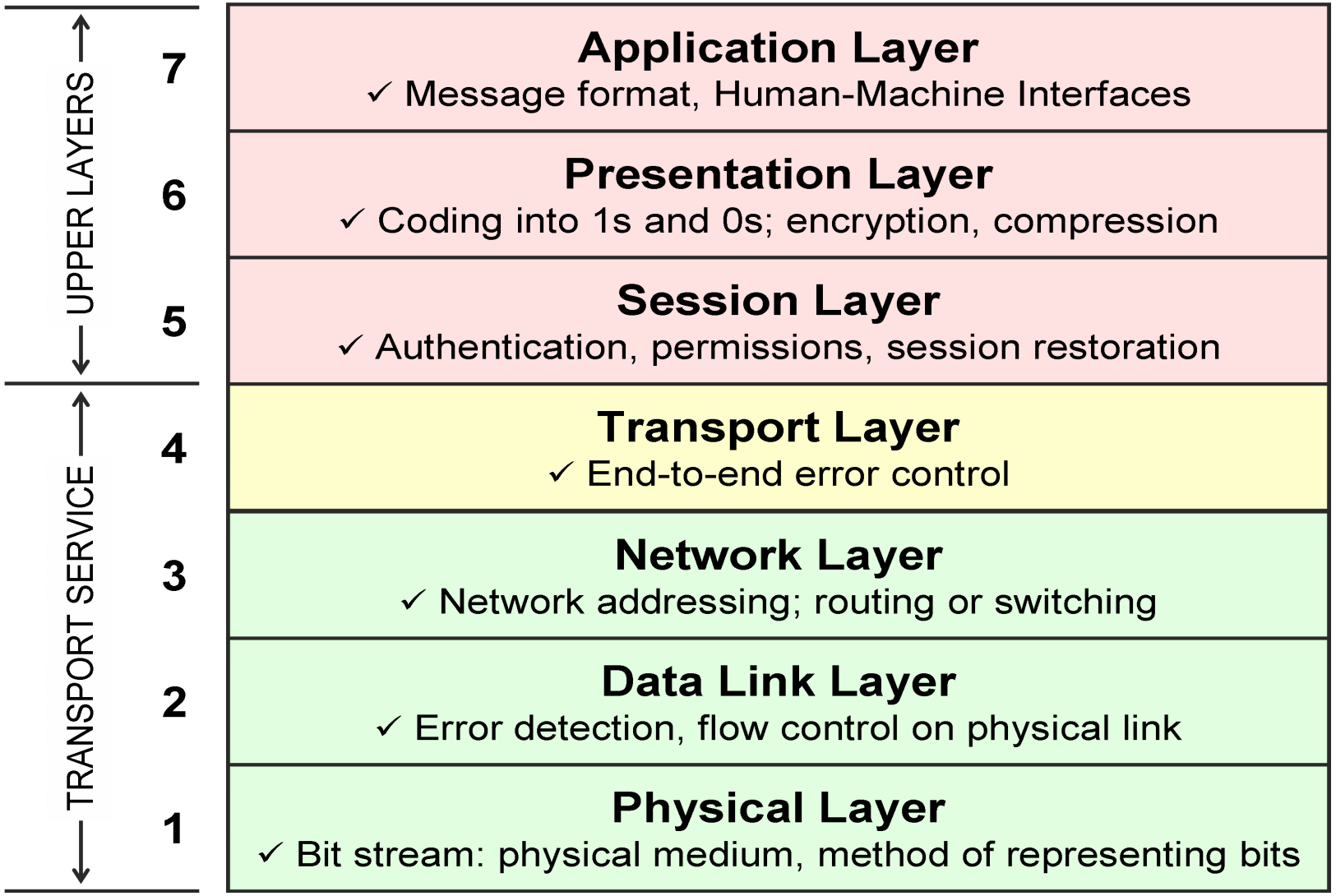
The Layered Architecture Pattern

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 6.8.
Example	A layered model of a digital learning system to support learning of all subjects in schools (Figure 6.9).
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 multilevel security.
Advantages	Allows replacement of entire layers as 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

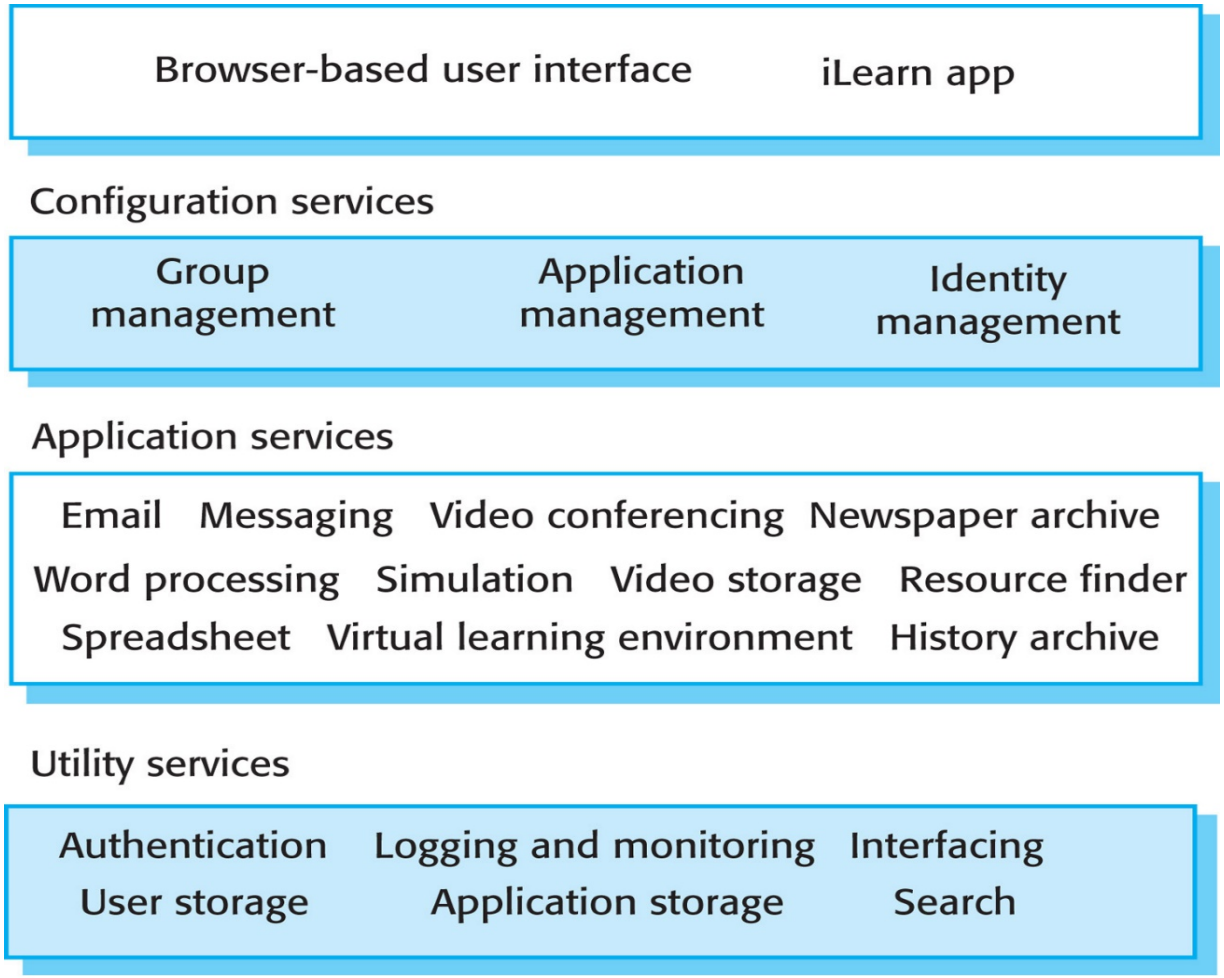


OSI 7 Layer Model

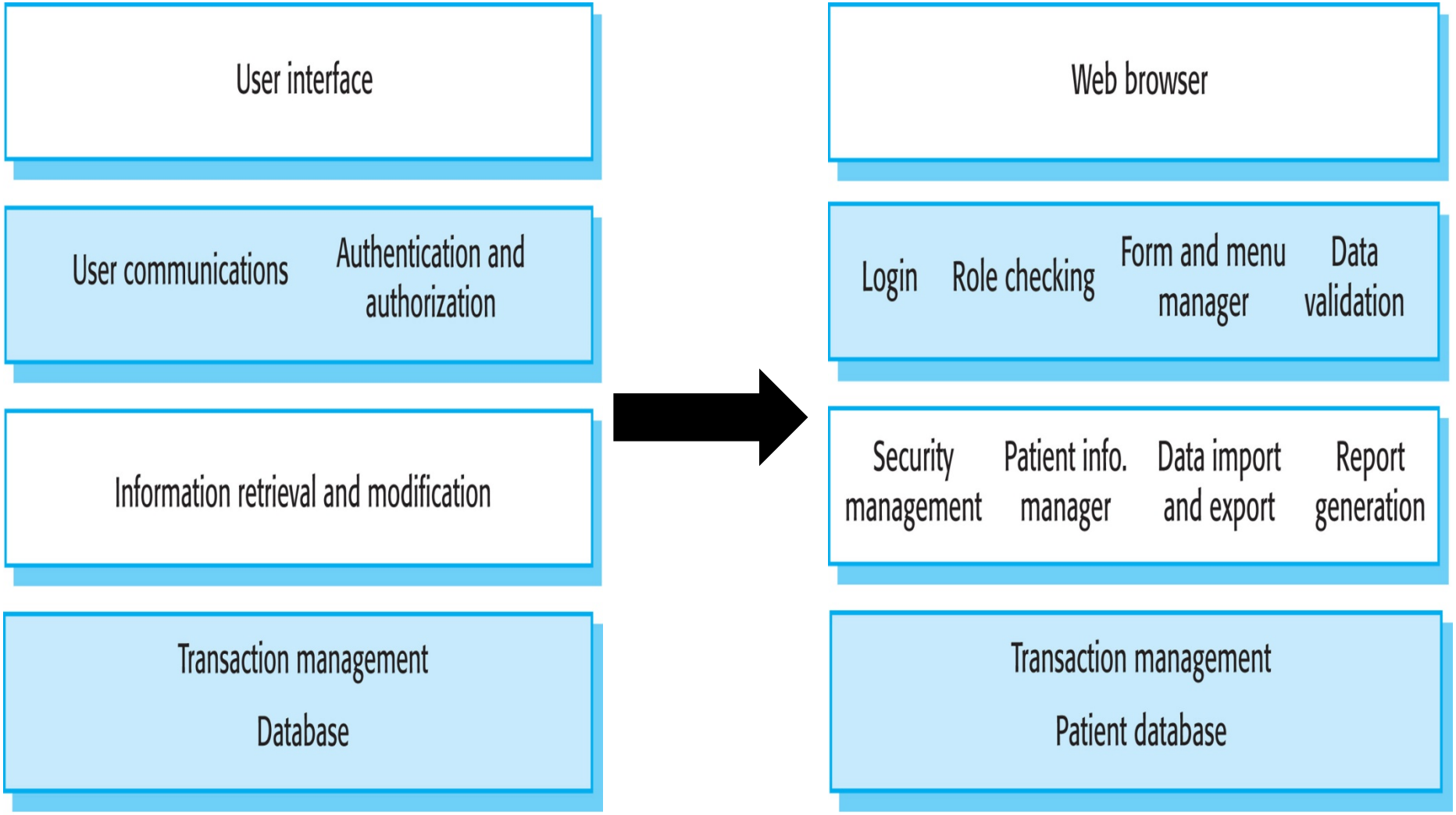


The Layered Architecture Pattern

The Architecture of iLearn: An Example

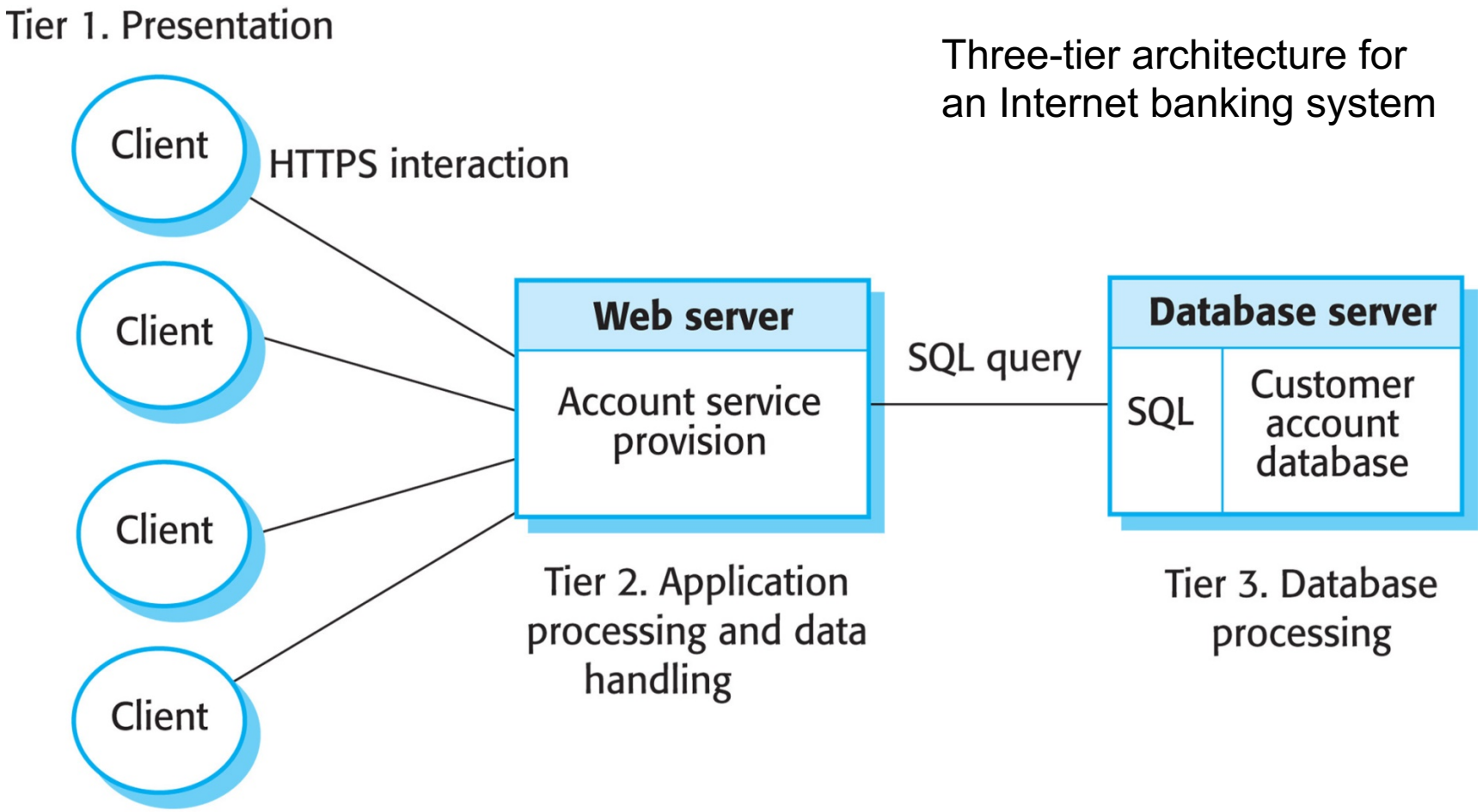


The Architecture of the Mentcare System



Copyright ©2016 Pearson Education, All Rights Reserved

The Layered Architecture Pattern



Three-tier architecture for an Internet banking system

SYRACUSE UNIVERSITY
ENGINEERING & COMPUTER SCIENCE

The MVC and Other Design Patterns

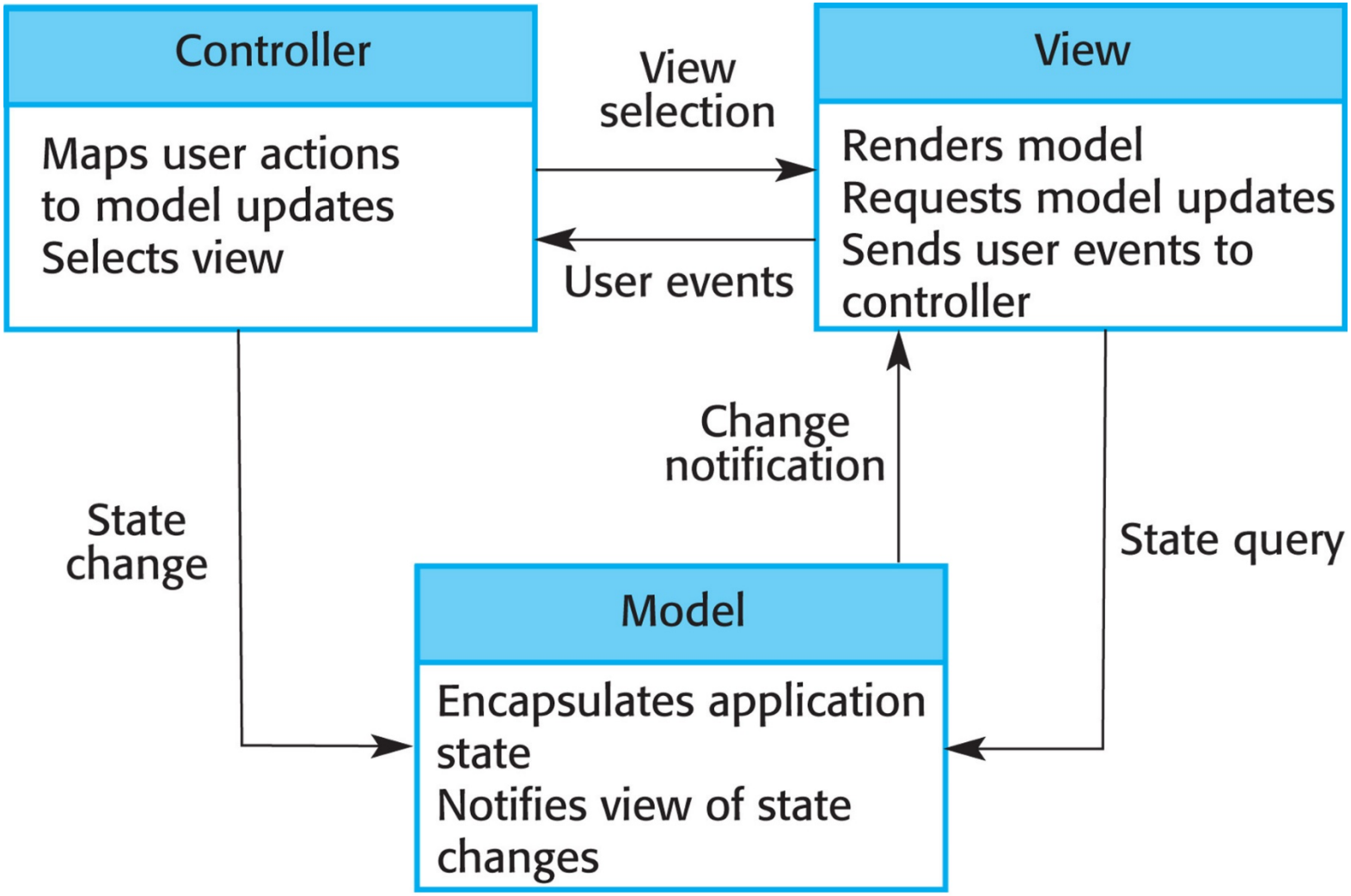
Week 7: Architectural Design, Part 2

Edmund Yu, PhD
Associate Professor
esyu@syr.edu

The MVC Pattern

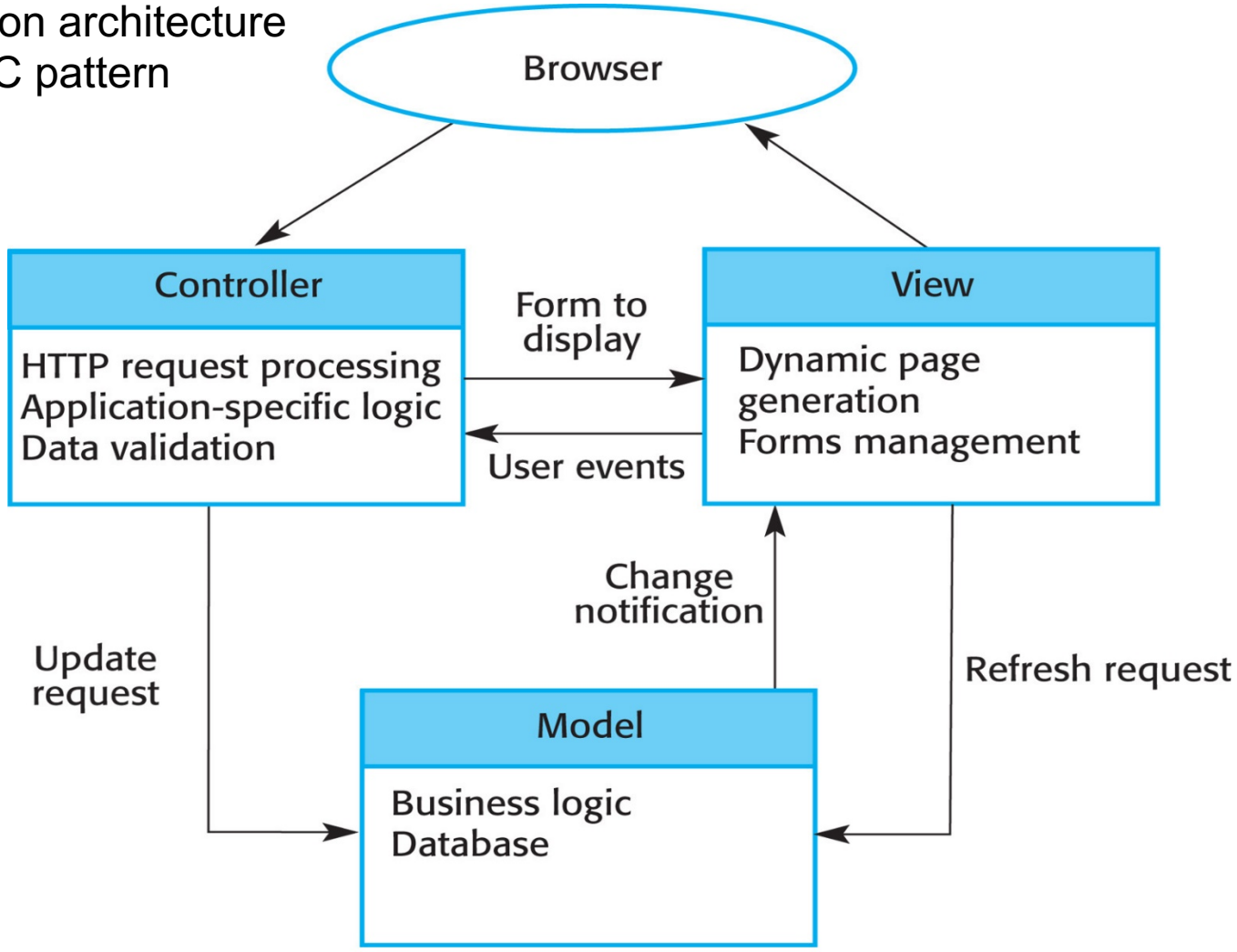
Name	MVC (Model-View-Controller)
Description	Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines 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 6.5.
Example	Figure 6.6 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	May involve additional code and code complexity when the data model and interactions are simple.

The Organization of the MVC

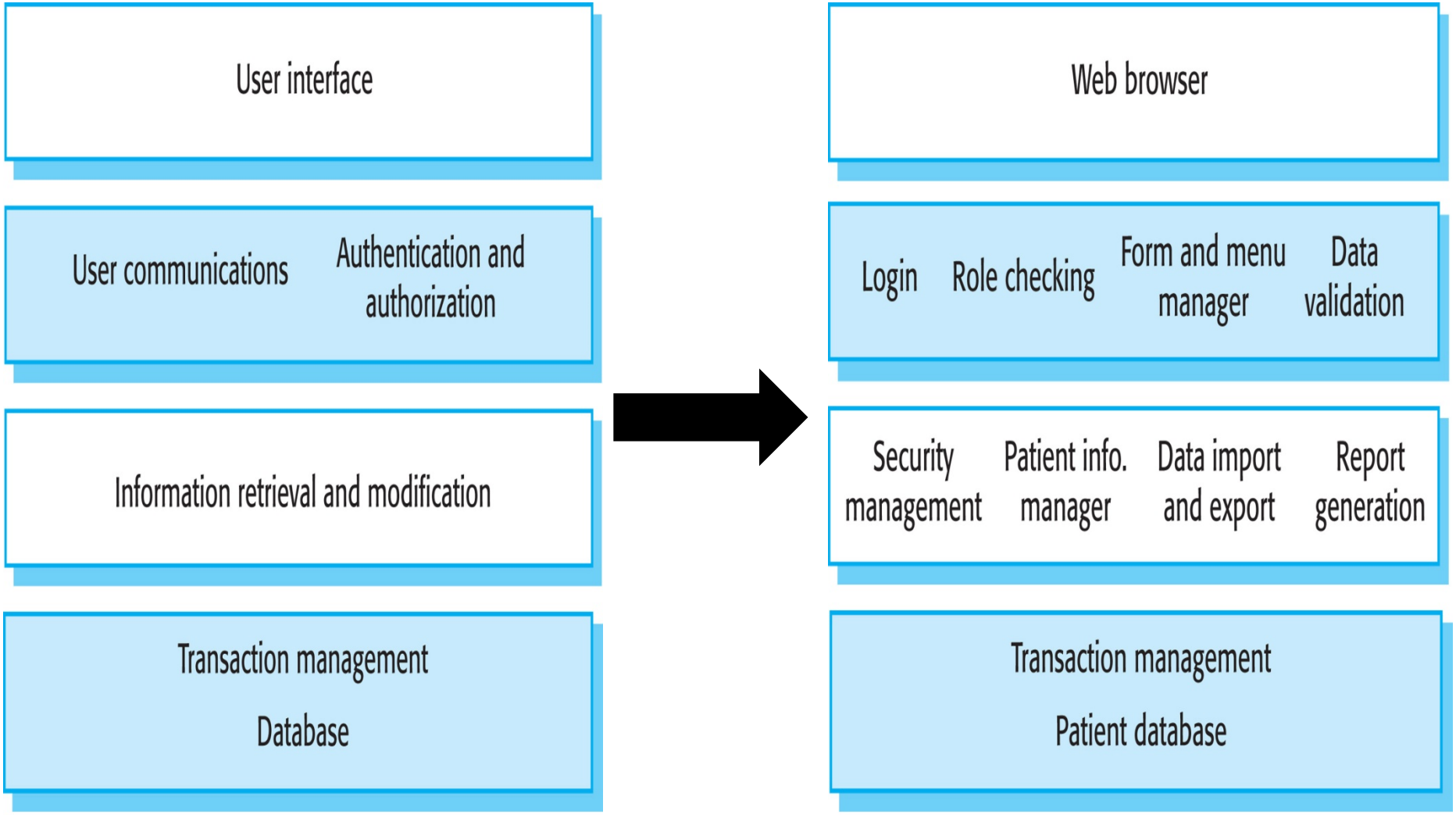


Web Application Architecture

Web application architecture using the MVC pattern



The Architecture of the Mentcare System



Copyright ©2016 Pearson Education, All Rights Reserved

Repository Architecture

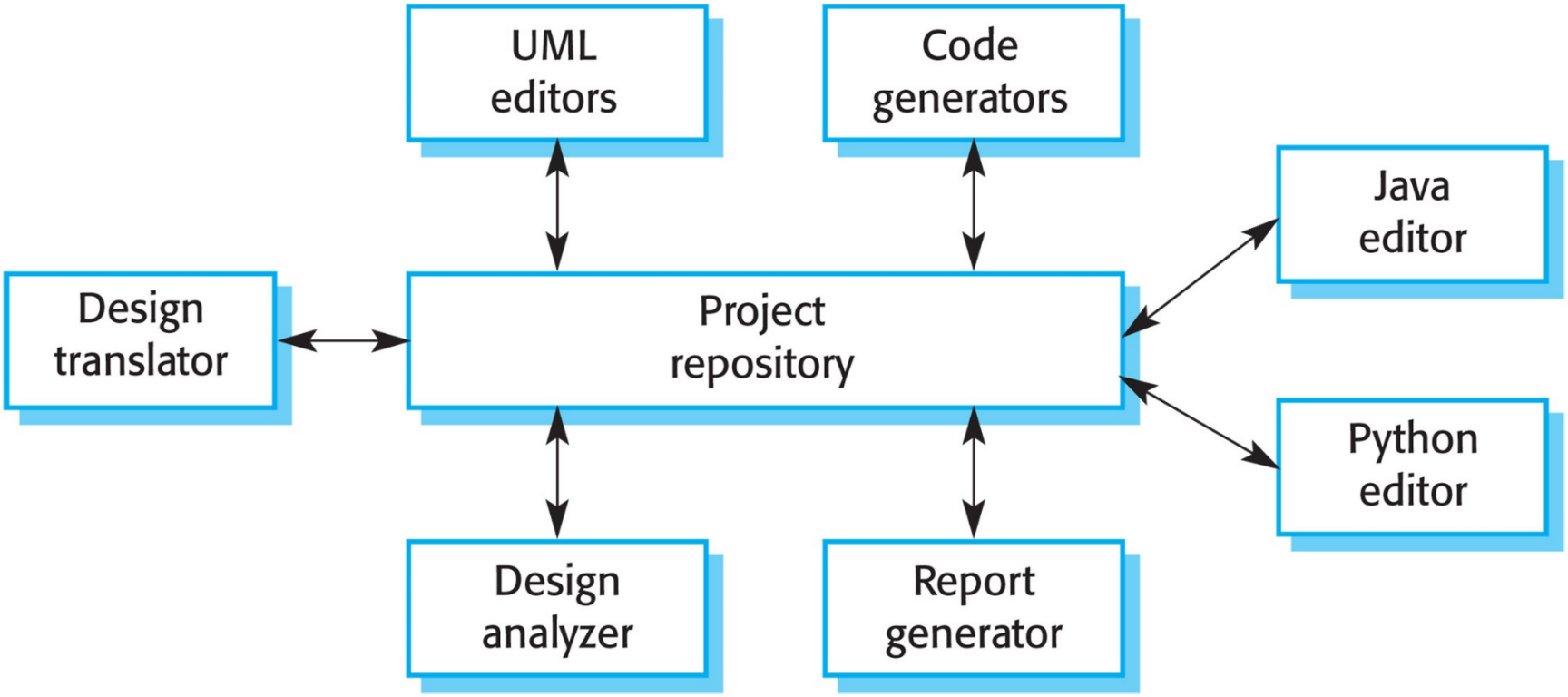
- When subsystems must exchange data, it may be done in two ways.
  - Shared data is held in a central database or **repository** and may be accessed by all subsystems.
  - Each subsystem maintains its own database and passes data explicitly to other subsystems.

- When large amounts of data are to be shared, the repository model of sharing is most commonly used, because this is an efficient data-sharing mechanism.

The Repository Pattern

Name	Repository
Description	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.
Example	Figure 6.11 is an example of 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.
When used	You should use this pattern 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.
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.

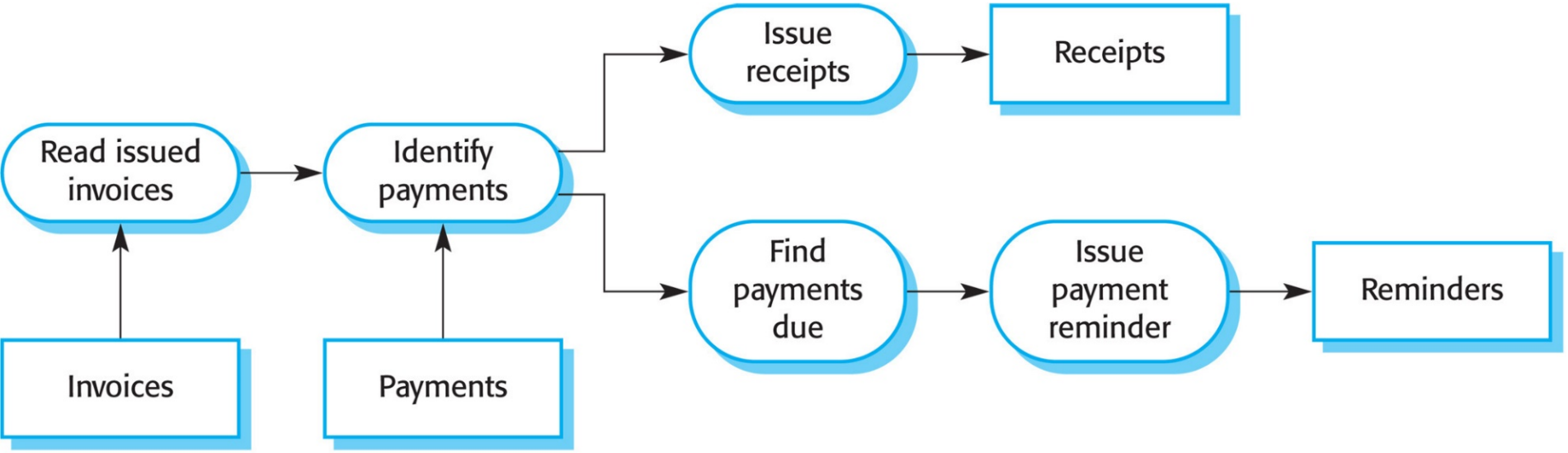
A Repository Architecture for an IDE



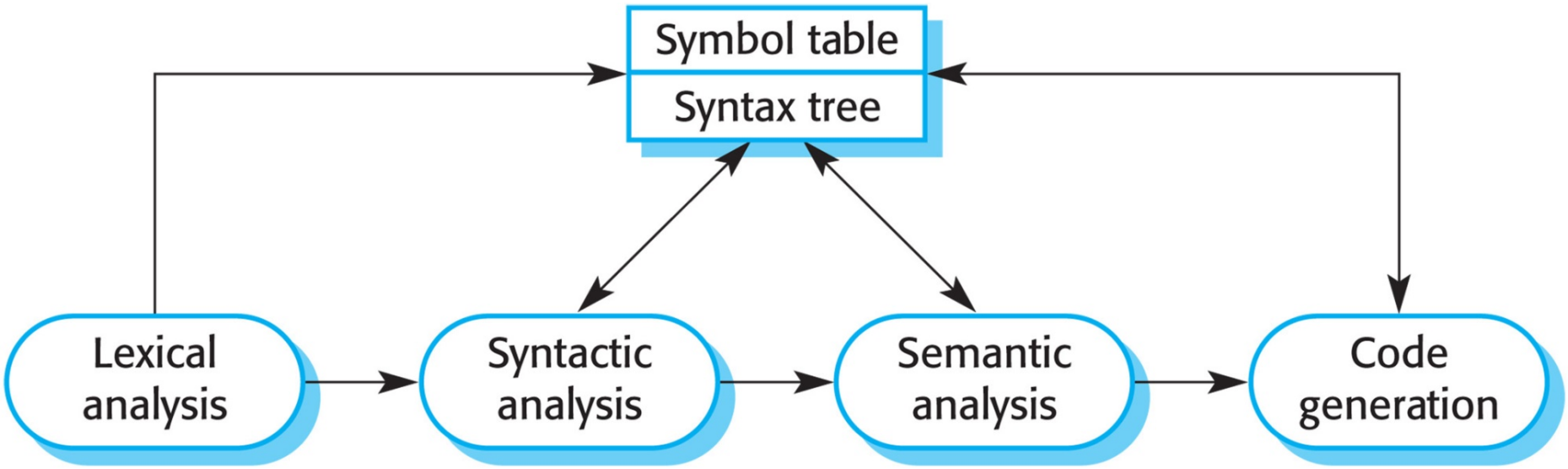
The Pipe and Filter Pattern

Name	Pipe and filter
Description	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.
Example	Figure 6.15 is an example of a pipe and filter system used for processing invoices.
When used	Commonly used in data-processing applications (both batch and transaction-based) where inputs are processed in separate stages to generate related outputs.
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 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 architectural components that use incompatible data structures.

An Example of the Pipe and Filter Architecture



A Pipe and Filter Compiler Architecture



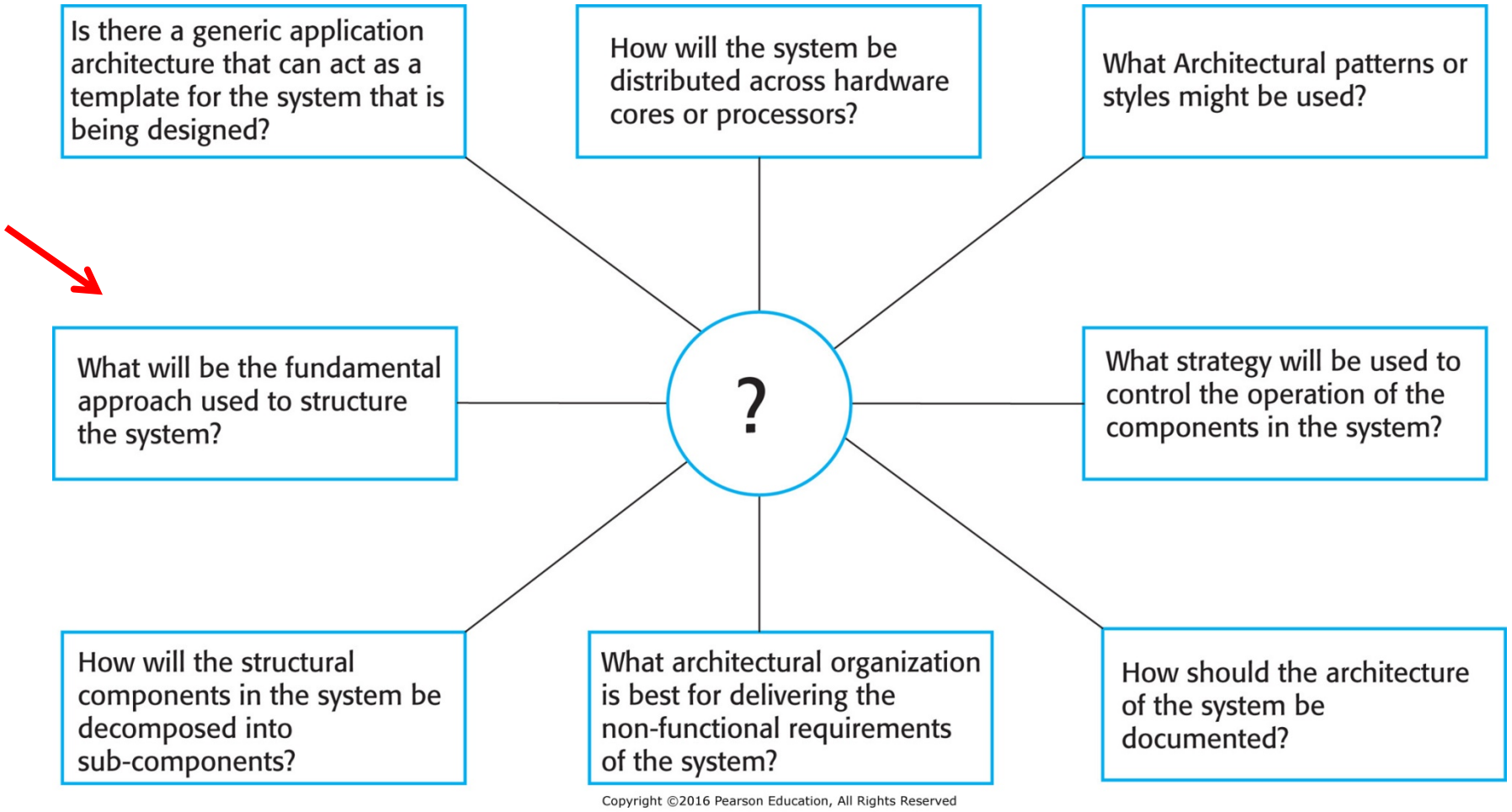
SYRACUSE UNIVERSITY
ENGINEERING & COMPUTER SCIENCE

Other Architectural Design Issues

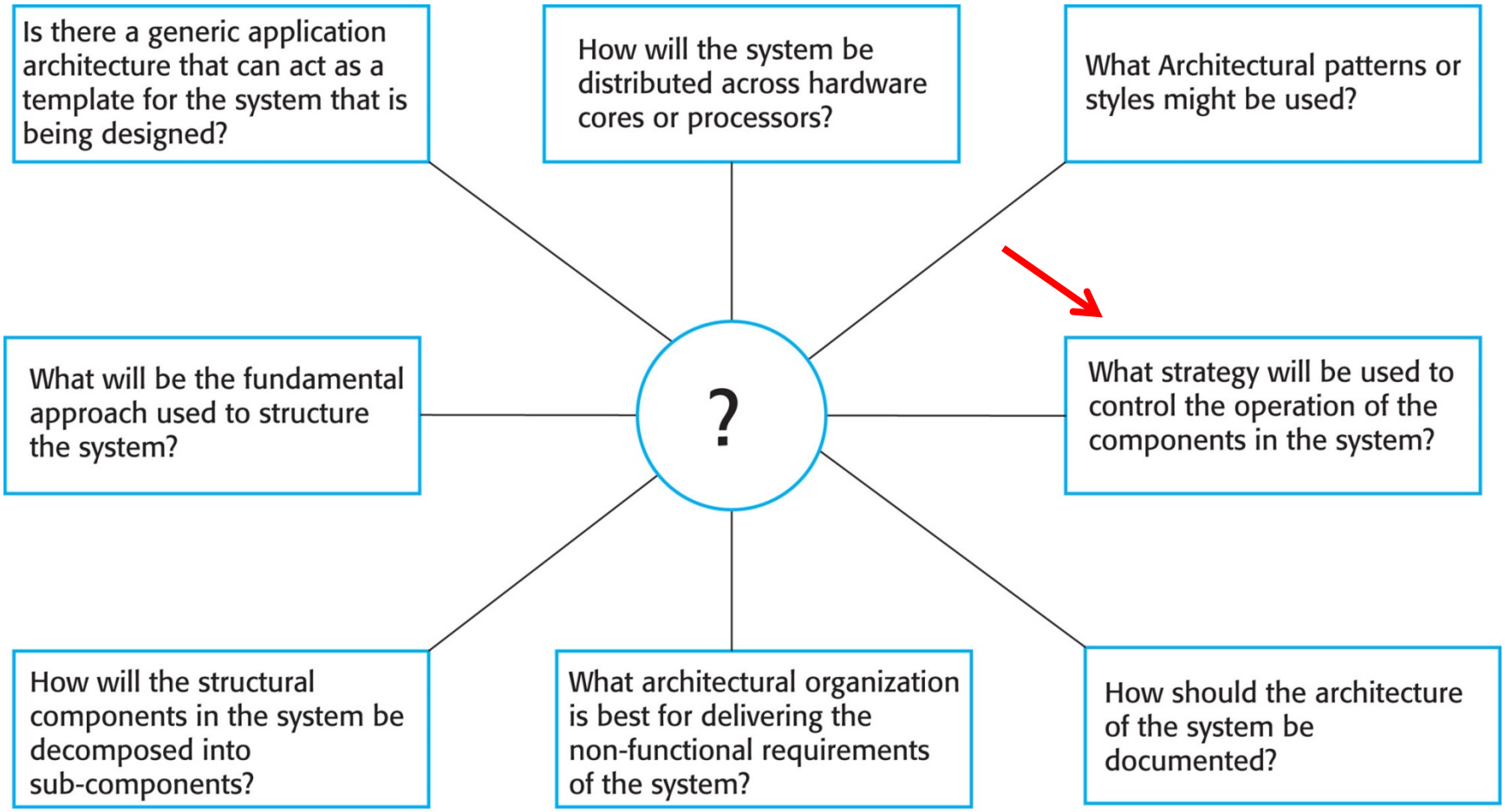
Week 7: Architectural Design, Part 2

Edmund Yu, PhD
Associate Professor
esyu@syr.edu

Architectural Design Decisions



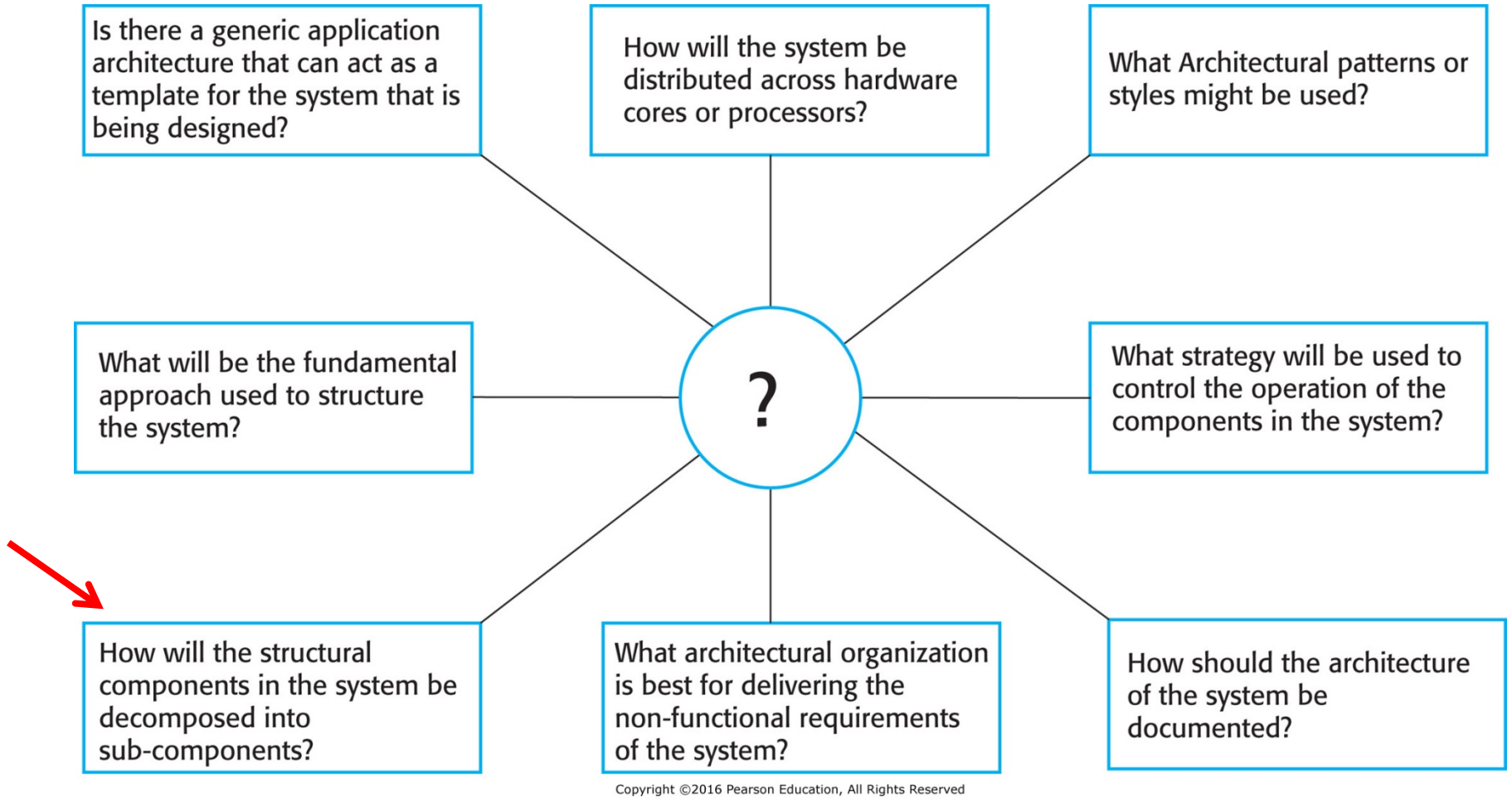
Architectural Design Decisions



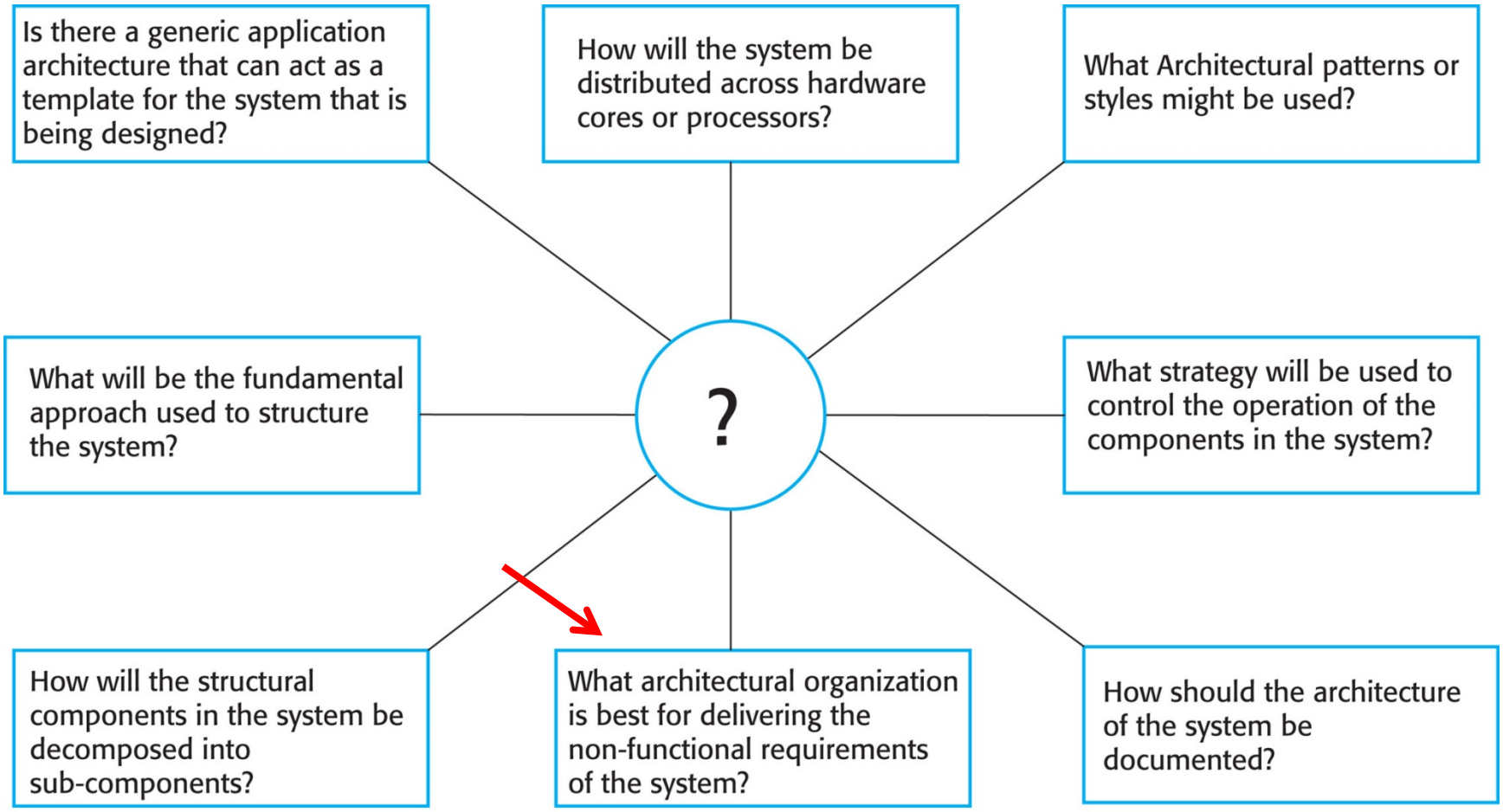
Control Strategies

- Architectural patterns may reflect the organization of a system from different viewpoints.
- The patterns introduced earlier reflect common control ways of organizing the control in a system.
  - **Centralized control**, where there is a component in charge that calls on services from other components in the system
    - The reflects the method/procedure/subroutine calls in programming languages.
  - **Event-based control,** where the system responds to asynchronous events from the system's environment
    - Event-based control is the norm in real-time systems and in many interactive systems with inputs from keyboards, mice, and so on.

Architectural Design Decisions



Architectural Design Decisions



Nonfunctional Requirements

- Because of the close relationship between nonfunctional requirements and software architecture, the particular architectural style and structure that you choose for a system should depend on the nonfunctional system requirements:

  1. Performance
  2. Security
  3. Safety
  4. Availability
  5. Maintainability

Performance

- If performance is a critical requirement:
  - The architecture should be designed to localize critical operations within a small number of components.
  - Components should all be deployed on the same computer rather than distributed across the network.
  - A few relatively large components rather than small, fine-grain components should be used to reduce the number of component communications.
  - Run-time system organizations that allow the system to be replicated and executed on different processors should be considered.

Security

- If security is a critical requirement:
  - A layered structure for the architecture should be used.
    - The most critical assets protected in the innermost layers
    - A high level of security validation applied to these layers

Safety

- If safety is a critical requirement:
  - The architecture should be designed so that safety-related operations are all located in either one 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.

Availability

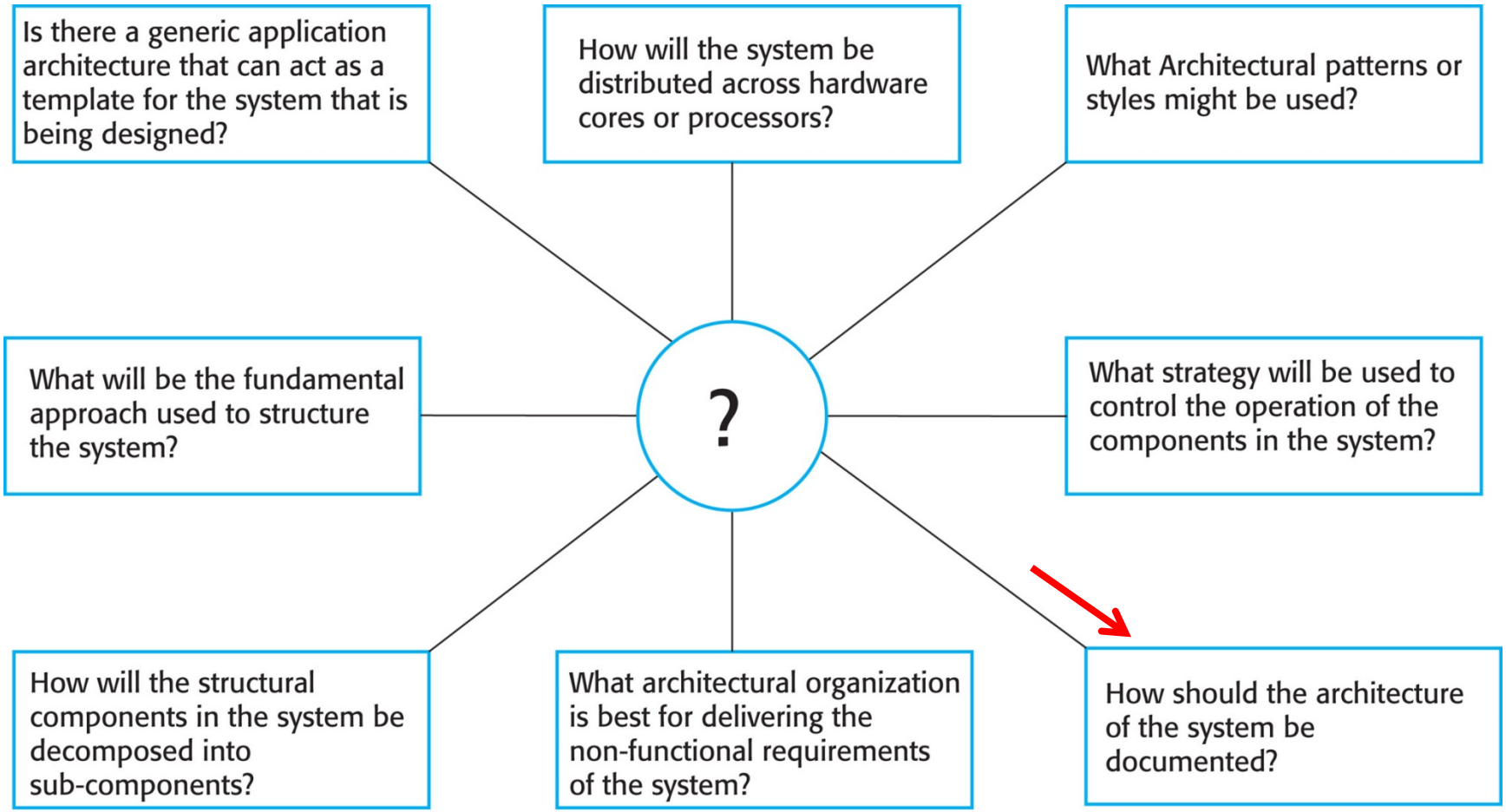
- If availability is a critical requirement:
  - The architecture should be designed to include redundant components…
    - So that it is possible to replace and update components without stopping the system.

- More fault-tolerant system architectures for high-availability systems are described in chapter 11

Maintainability

- If maintainability is a critical requirement:
  - 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.
  - Shared data structures should be avoided.

Architectural Design Decisions

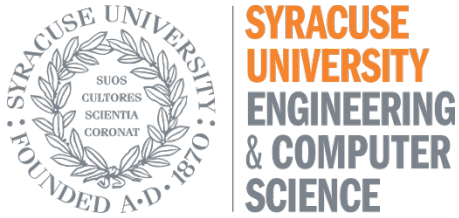


SYRACUSE UNIVERSITY
ENGINEERING & COMPUTER SCIENCE

Model-Driven Architecture

Week 7: Architectural Design, Part 2

Edmund Yu, PhD
Associate Professor
esyu@syr.edu



Model-Driven Architecture

- Model-driven architecture (MDA) is an approach to software development in which **models,** usually UML models, rather than programs, are the principal outputs of the development process.
  - The programs that execute on a hardware/software platform are then generated automatically from the models.
  - Proponents of MDA argue that this raises the level of abstraction in software engineering so that engineers no longer have to be concerned with programming language details or the specifics of execution platforms.

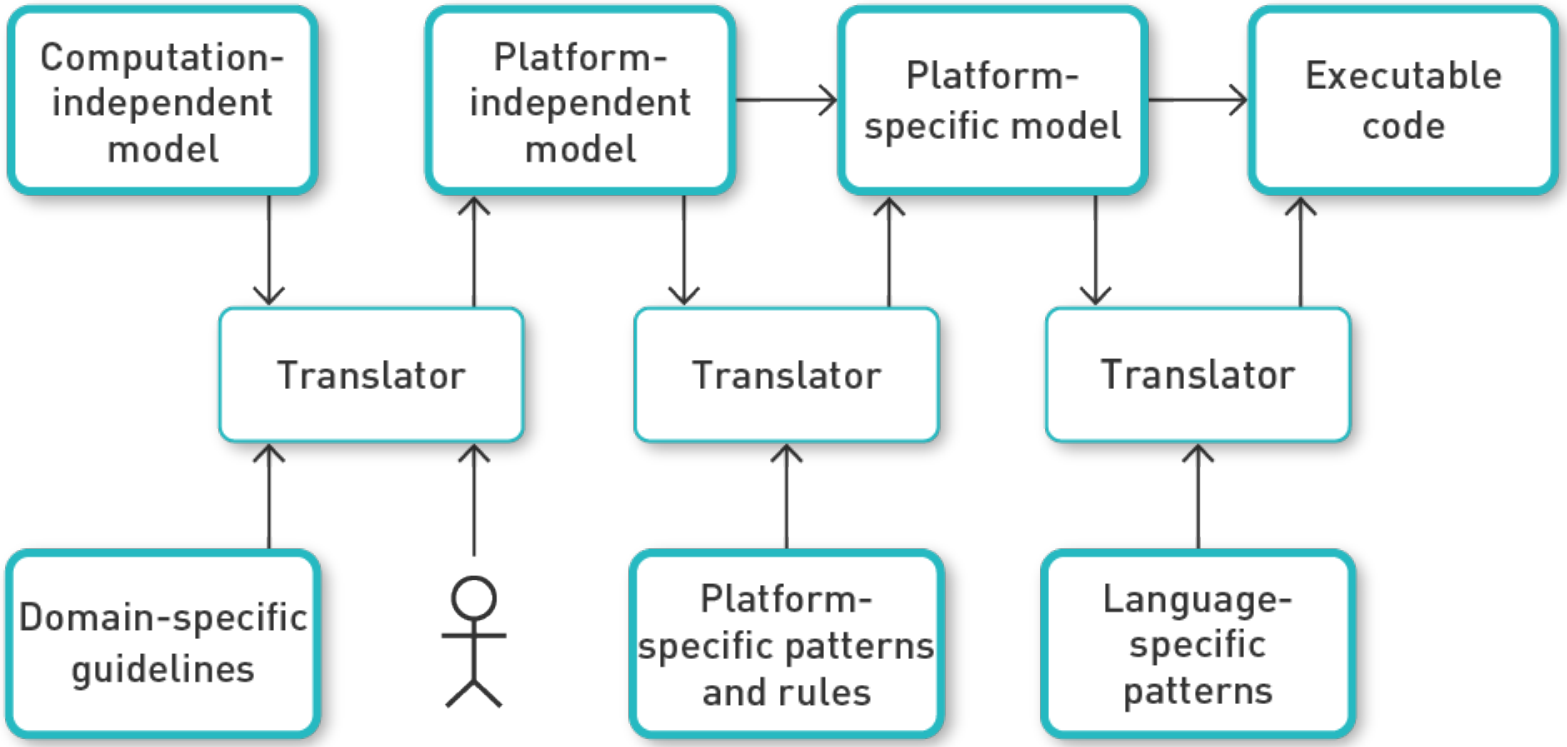
Usage of Model-Driven Architecture

- MDA is still at an early stage of development, and it is unclear whether or not it will have a significant effect on software engineering practice.
  - Pros
    - Allows systems to be considered at higher levels of abstraction, as mentioned earlier.
    - Generating code automatically means that it is cheaper to adapt systems to new platforms (cross-platform).
  - Cons
    - Models for abstraction not necessarily right for implementation.
    - Savings from generating code may be outweighed by the costs of developing translators for new platforms.

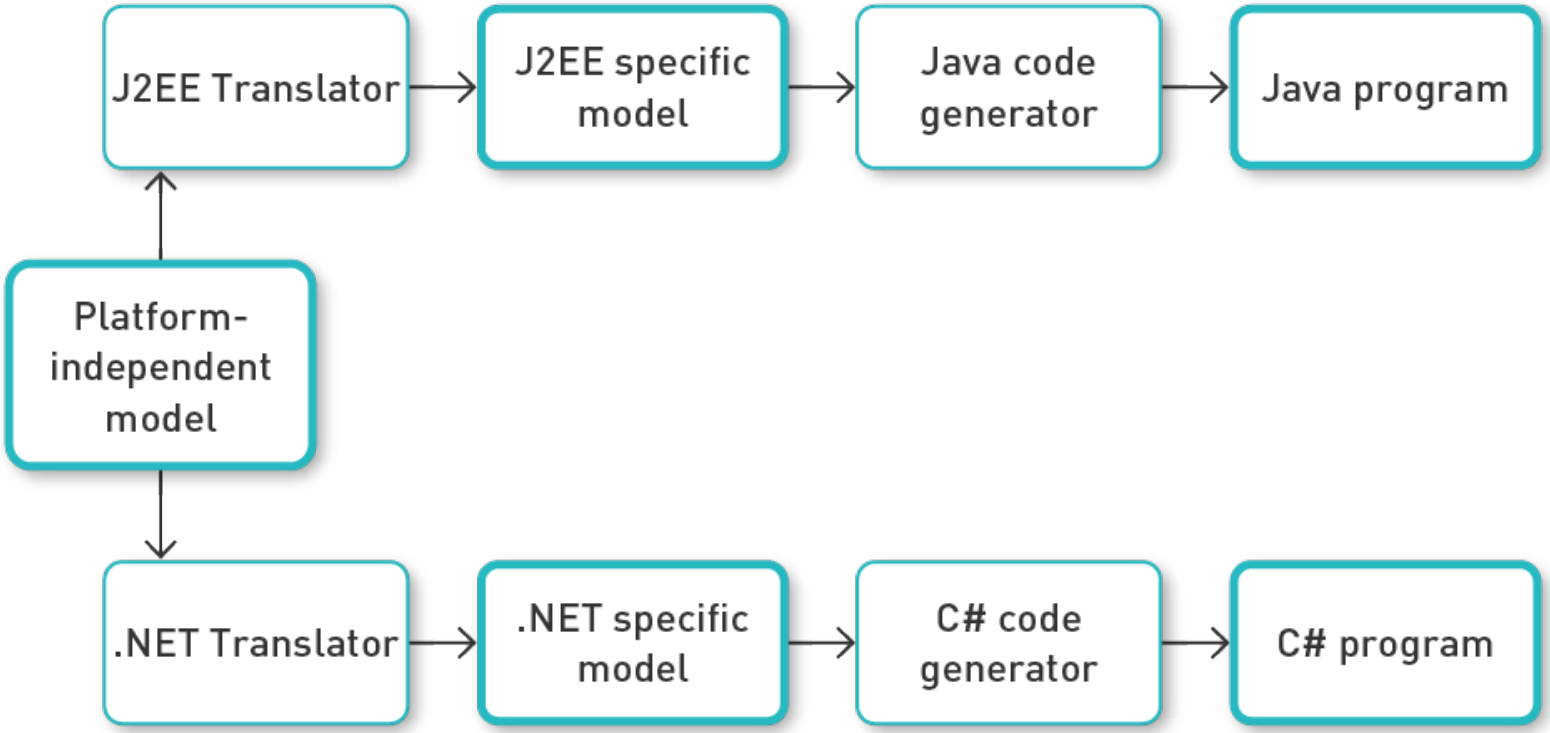
Types of Models

- A computation-independent model (CIM)
  - These model the important domain abstractions used in a system.
  - CIMs are sometimes called domain models.
- A platform-independent model (PIM)
  - These model the operation of the system without reference to its implementation.
  - The PIM is usually described using UML models that show the static system structure and how it responds to external/internal events.
- Platform-specific models (PSM)
  - These are transformations of the platform-independent model with a separate PSM for each application platform.

MDA Transformations



Multiple Platform-Specific Models



Agile Methods and MDA

- The developers of MDA claim that it is intended to support an iterative approach to development and so can be used within agile methods.
  - However, the notion of extensive up-front modeling contradicts the fundamental ideas in the agile manifesto, and hence few agile developers feel comfortable with it.

- If transformations can be completely automated and a complete program generated from a PIM, then, in principle, MDA could be used in an agile development process as no separate coding would be required.

Adoption of MDA

- A range of factors has limited the adoption of MDA.
  - Specialized tool support is required to convert models from one level to another.
  - There is limited tool availability and organizations may require tool adaptation and customization to their environment.
  - For the long-lifetime systems developed using MDA, companies are reluctant to develop their own tools or rely on small companies that may go out of business.
  - The arguments for platform independence are valid only for large, long-lifetime systems.
  - The widespread adoption of agile methods over the same period that MDA was evolving has diverted attention away from model-driven approaches.

Home

Getting started

Documentation

» Tutorials and Articles

» Javadoc

Downloads

Support

Community

» Newsgroups

» Submit A Bug

» About This Project

Eclipse Modeling Framework (EMF)

The EMF project is a modeling framework and code generation facility for building tools and other applications based on a structured data model. From a model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the model, along with a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor.

EMF (core) is a common standard for data models, many technologies and frameworks are based on. This includes **server solutions**, **persistence frameworks**, **UI frameworks** and **support for transformations**. Please have a look at the **modeling project for an overview of EMF technologies**.

EMF (Core)

EMF consists of three fundamental pieces:

» **EMF** - The core EMF framework includes a **meta model (Ecore)** for describing models and runtime support for the models including change notification, persistence support with default XMI serialization, and a very efficient reflective API for manipulating EMF objects generically.

papyrus

Documentation
Tutorials, Examples, Videos, Online Reference

Getting started
Papyrus installation guide
Update Site

Getting Involved
CVS, Workspace Setup, Wiki, Committers

Support
Bug Tracker, Newsgroup
Professional Support

About Papyrus

Papyrus is aiming at providing an integrated and user-consumable environment for editing any kind of EMF model and particularly supporting UML and related modeling languages such as SysML and MARTE. Papyrus provides diagram editors for EMF-based modeling languages amongst them UML 2 and SysML and the glue required for integrating these editors (GMF-based or not) with other MBD and MDSD tools.

Papyrus also offers a very advanced support of UML profiles that enables users to define editors for DSLs based on the UML 2 standard. The main feature of Papyrus regarding this latter point is a set of very powerful customization mechanisms which can be leveraged to create user-defined Papyrus perspectives and give it the same look and feel as a "pure" DSL editor.

UML2.5.0

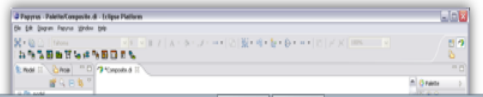
Papyrus is graphical editing tool for **UML2** as defined by **OMG**. Papyrus targets to implement 100% of the OMG specification!

DSL

Papyrus provides a very advanced support for UML profiles enabling support for "pure" DSL. Every part of Papyrus may be customized: model explorer, diagram editors, property editors, etc.

SysML

Papyrus provides also a complete support to **SysML** in order to enable model-based system engineering. It includes an implementation of the SysML static profile and the specific graphical editors required for SysML.



ONGOING WORK...

- The plan for Papyrus 1.2.x is not available yet
- **Previous plan (1.1.0/Mars)**

CURRENT STATUS

Papyrus team has released the version 1.1.0. It can be installed on Eclipse Mars 4.5. You can install it using the instructions **here**.

You can read **here** what is new and updated in Papyrus 1.1.0.

PAPYRUS CINEMA

Following this link, you will find the **Papyrus cinema corner**.

SYRACUSE UNIVERSITY
ENGINEERING & COMPUTER SCIENCE