Course Lecture Schedule

Date	Topic	Book Chapter(s)
Wed 8.9.	Course introduction	
Tue 14.9.	Introduction to Software Engineering	Chapter 1
Tue 21.9.	Software Processes	Chapter 2
Mon 27.9	Agile Software Engineering	Chapter 3
Tue 5.10.	Requirements Engineering	Chapter 4
Mon 11.10.	**Architectural Design**	**Chapter 6**
Wed 20.10.	Modeling and implementation	Chapters 5 & 7
Mon 1.11.	Testing & Quality	Chapters 8 & 24
Mon 8.11.	Software Evolution & Configuration Management	Chapters 9 & 25
Mon 15.11.	Software Project Management	Chapter 22
Mon 22.11.	Software Project Planning	Chapter 23
Mon 29.11.	Global Software Engineering	
Wed 8.12.	Software Business	
Mon 13.12.	Last topics	

Maria
Susan
Sami
Hyrynsalmi

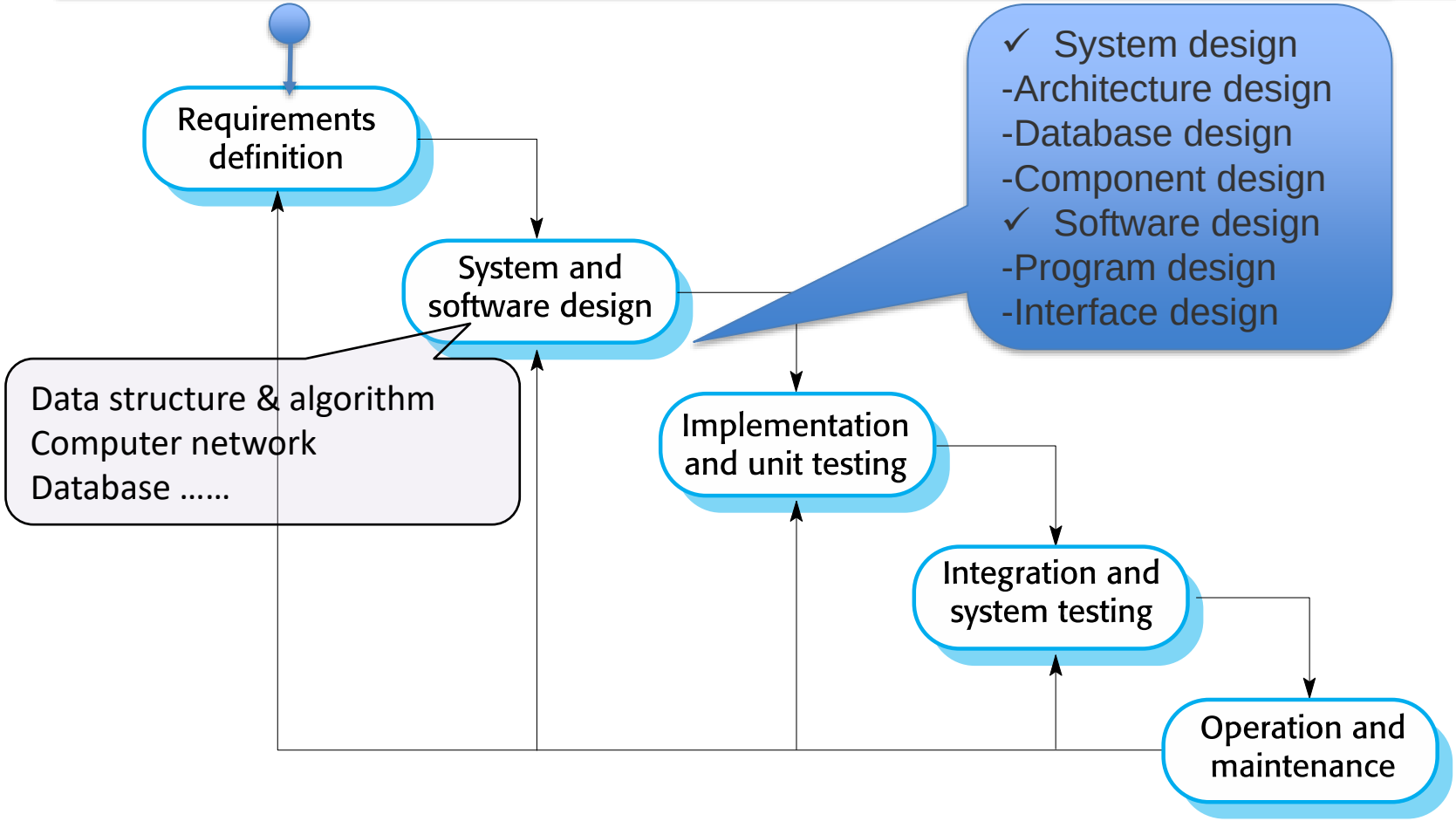
Chapter 6
Architectural Design

Topics covered

- ✧ What is architecture design
- ✧ Architectural views
- ✧ System architecture & Architectural patterns
- ✧ Application architectures

What is architectural design

Architecture design is the most technically challenging stage in waterfall model



Architectural design

- Architectural design is the critical link between design and requirements engineering, as it identifies the main structural components in a system and the relationships between them.
- For agile, it is generally accepted that an early stage of agile processes is to design an overall systems architecture.
- For IID, system architecture must be adaptive to change.

Structural models

The output of the architectural design process is an architectural model.

- Structural models of software display the organization of a system in terms of the components that make up that system and their relationships.
- Structural models may be static models, which show the structure of the system design, or dynamic models, which show the organization of the system when it is executing.

Use of architectural models

- As a way of facilitating discussion about the system design
- As a way of documenting an architecture that has been designed
- The architecture may be reusable across a range of systems.

*Example：Can you understand this program?

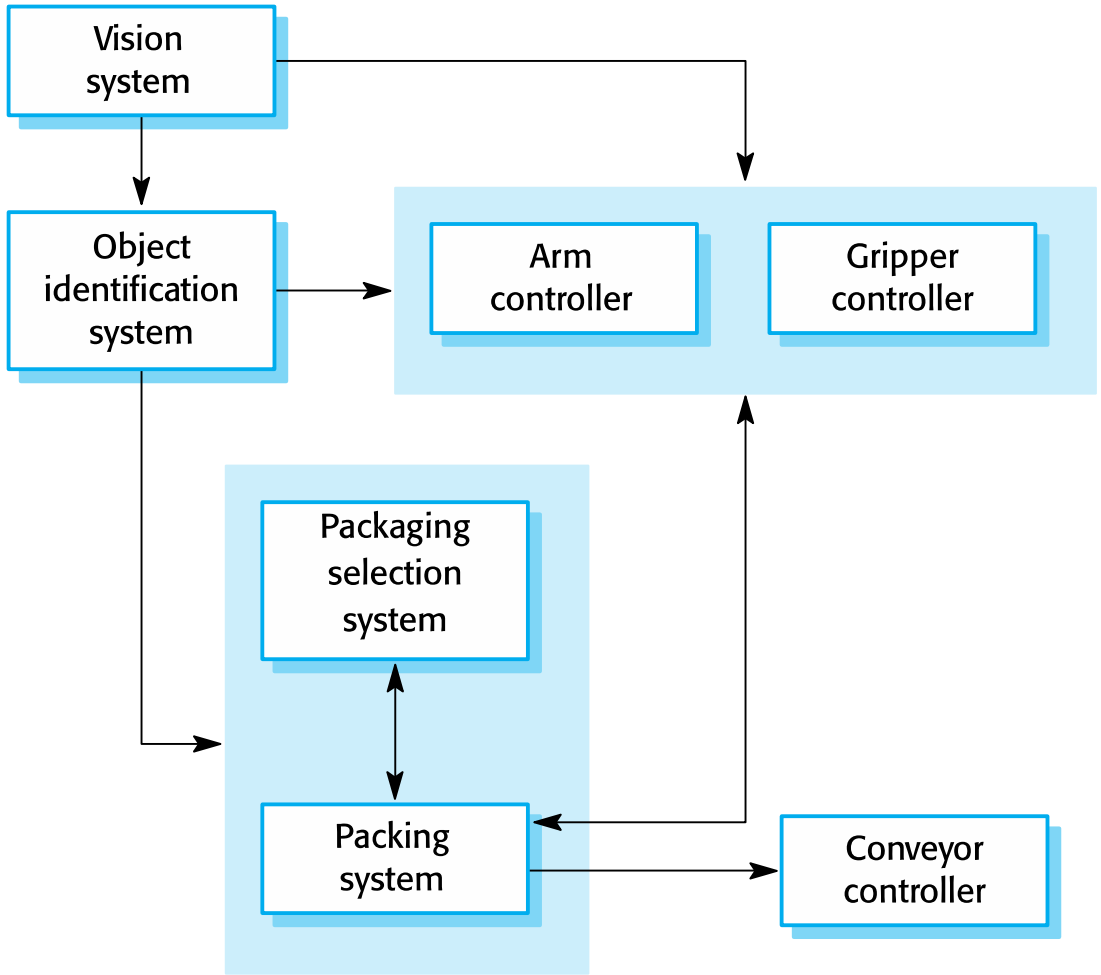

```
#include <reg52.h>
bit flagFrame = 0; //帧接收完成标志，即接收到一帧新数据
bit flagTxd = 0; //单字节发送完成标志，用来替代 TXD 中断标志位
unsigned char cntRxd = 0; //接收字节计数器
unsigned char pdata bufRxd[64]; //接收字节缓冲区
extern void UartAction(unsigned char *buf, unsigned char len); /* 串口配置函数，baud-通信波特率 */
void ConfigUART(unsigned int baud){
    SCON = 0x50; //配置串口为模式 1
    TMOD &= 0x0F; //清零 T1 的控制位
    TMOD |= 0x20; //配置 T1 为模式 2
    TH1 = 256 - (11059200/12/32)/baud; //计算 T1 重载值
    TL1 = TH1; //初值等于重载值 ET1 = 0; //禁止 T1 中断
    ES = 1; //使能串口中断
    TR1 = 1; //启动 T1
} /* 串口数据写入，即串口发送函数，buf-待发送数据的指针，len-指定的发送长度 */
void UartWrite(unsigned char *buf, unsigned char len){
    while (len--){ //循环发送所有字节 flagTxd = 0; //清零发送标志
        SBUF = *buf++; //发送一个字节数据
        while (!flagTxd); //等待该字节发送完成
    }
} /* 串口数据读取函数，buf-接收指针，len-指定的读取长度，返回值-实际读到的长度 */
unsigned char UartRead(unsigned char *buf, unsigned char len){
    unsigned char i; //指定读取长度大于实际接收到的数据长度时， //读取长度设置为实际接收到的数据长度
    if (len > cntRxd){ len = cntRxd; }
    for (i=0; i<len; i++){ //拷贝接收到的数据到接收指针上
        *buf++ = bufRxd[i];
    }
    cntRxd = 0; //接收计数器清零
    return len; //返回实际读取长度
} /* 串口接收监控，由空闲时间判定帧结束，需在定时中断中调用，ms-定时间隔 */
```

Difficult to understand:

- Complex program
- Different programming languages
- Different language Comments
- ……

```
void UartRcvMonitor(unsigned char ms){ static unsigned  char cntBak = 0;
static unsigned char idleMs = 0;
if (cntRxd > 0){ //接收计数器大于零时，监测总线空闲时间
if (cntBak != cntRxd){ //若接收计数器改变，即刚接收到数据时，清零空闲计时
cntBak = cntRxd; idleMs = 0; //更新比较值，空闲计时清零
} else if (idleMs < 30){ //总线空闲时，累积空闲时间
if (idleMs + 30 < 30){ //空闲时间达到 30ms 时，即判定为一帧接收完毕 flagFrame = 1; //设置帧接收完成标志 }}}}
else{ cntBak = 0; }} /* 串口驱动函数，监测数据帧的接收，调用间隔由你，建议与调用本函数 */
void UartDriver(){ unsigned char len; unsigned char pdata buf[40];
if (flagFrame){ //有命令到达时，读取处理该命令
flagFrame = 0;
len = UartRead(buf, sizeof(buf)); //将接收到的命令读取到缓冲区中
UartAction(buf, len); //处理数据等，调用动作执行函数 }} /* 串口中断服务函数 */
void InterruptUART()
interrupt 4{ if (RI){ //当接收到新字节 RI = 0; //将串口接收中断标志位 //接收缓冲区尚未可以时，保存接收字节，并递增计数器
if (cntRxd < sizeof(bufRxd)){ bufRxd[cntRxd++] = SBUF; }}
if (TI){ //当字节发送完毕 TI = 0; //将串发送中断标志位 flagTxd = 1; //设置字节发送完成标志 }}
```

Example: The architecture of a packing robot control system
(*Can you get the meaning of this diagram?)



Architectural abstraction

- 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.
- 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 and system characteristics

- Performance
  - Localize critical operations and minimize communications. Use large rather than fine-grain components.
- Security
  - Use a layered architecture with critical assets in the inner layers.
- Availability
  - Include redundant components and mechanisms for fault tolerance.
- Maintainability
  - Use fine-grain, replaceable components.
- Reusability

Architectural views

What aspects are concerned by software developers？

- Each architectural model only shows one view or perspective of the system.
- For both design and documentation, usually need to present multiple views of the software architecture.

Design process stages

There are a variety of different design processes that depend on the organization using the process.

- Common activities based on object-oriented methods in these processes include:
  - Define the context and the external interactions with the system;
  - Design the system architecture;
  - Identify the principal system objects;
  - Develop design models;
  - Specify object interfaces.

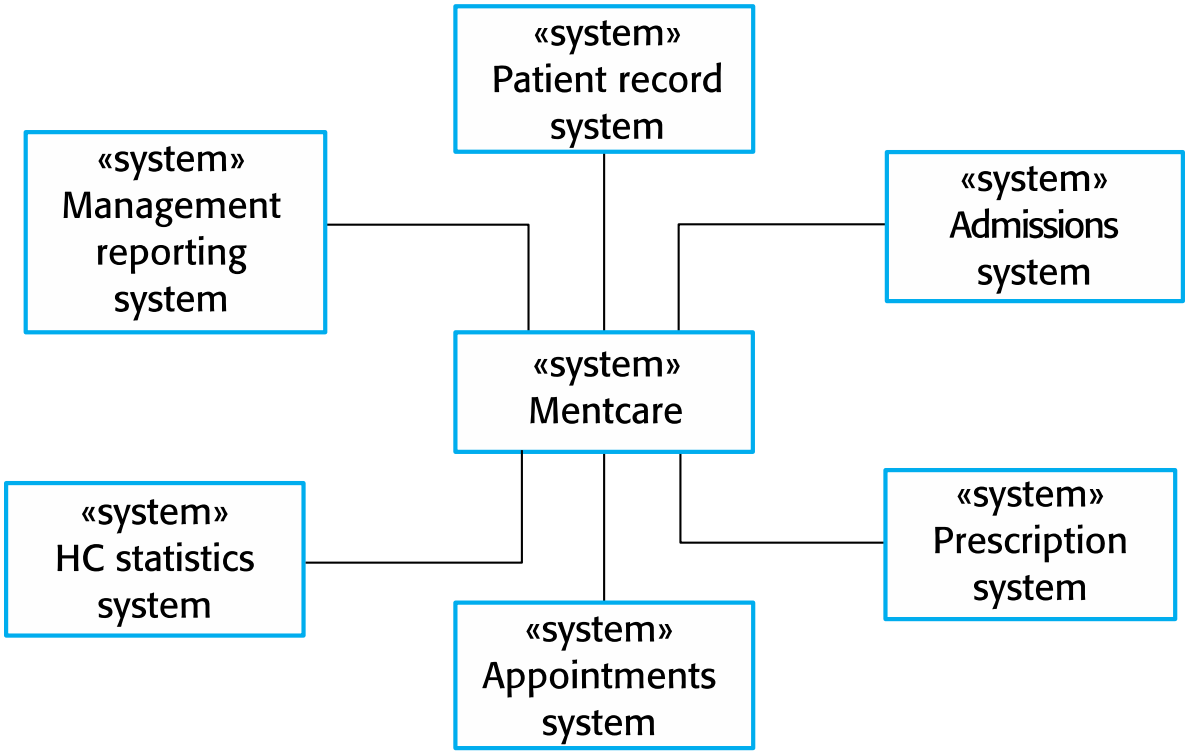
Context models

- Context models are used to illustrate the operational context of a system.
- They show the system boundaries, and its relationship with other systems.
- Social and organisational concerns may affect the decision on where to position system boundaries.

System boundaries

- System boundaries are established to define what is inside and what is outside the system.
  - They show other systems that are used or depend on the system being developed.
- The position of the system boundary has a profound effect on the system requirements.
  - That can increase / decrease the influence or workload of different parts of an organization.

Example: The context of the Mentcare system

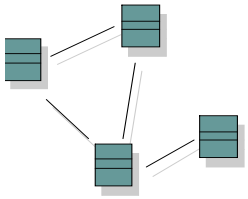


- Off-the-shelf systems: It can be cheaper and faster than developing a system.
- Pre-defined system: new system should support effective interaction with the existing system.

Architecture design & Modeling

- The main outcomes of architecture design are some kind of system architectural models.
- System modeling has now come to mean representing a system using some kind of graphical notations, which is now almost always based on notations in the Unified Modeling Language (UML).

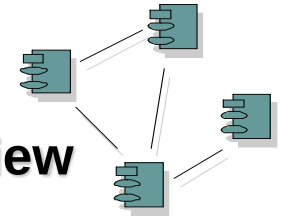
4 + 1 view model of software architecture



Logical View

Analysts/Designers

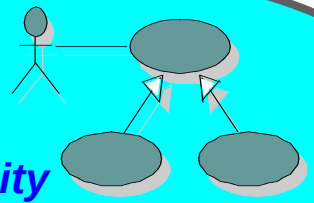
Structure



Implementation View

Programmers

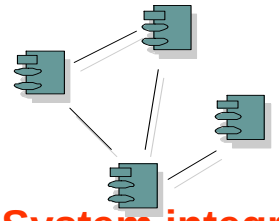
Software management



End-user

Functionality

Use-Case View



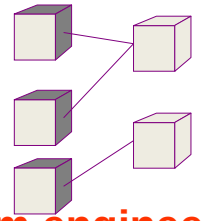
Process View

System integrators

Performance

Scalability

Throughput



Deployment View

System engineering

System topology

Delivery, installation

communication

Interaction models

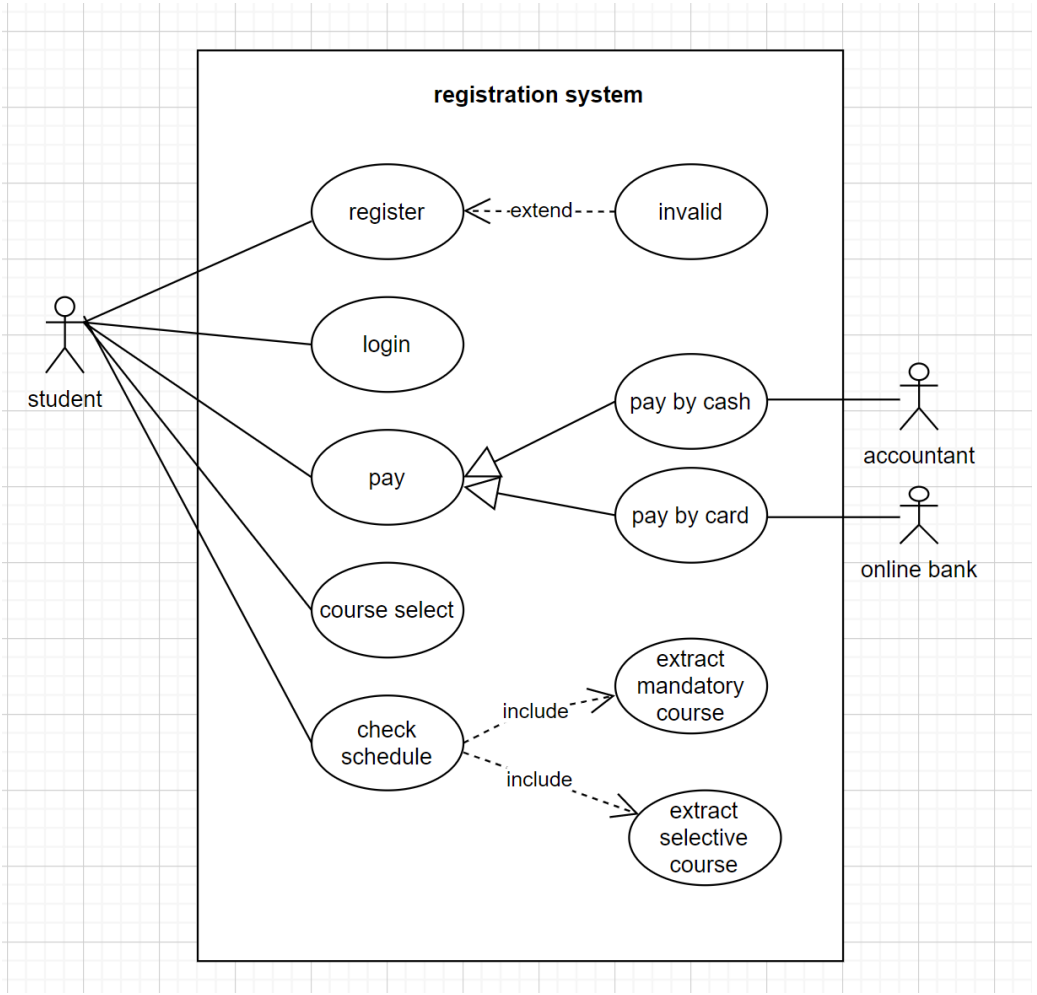
- Modeling user interaction is important as it helps to identify user requirements.
- Modeling system-to-system interaction highlights the communication problems that may arise.
- Modeling component interaction helps us understand if a proposed system structure is likely to deliver the required system performance and dependability.
- Use case diagrams and sequence diagrams may be used for interaction modelling.

Use case modeling

- Use cases were developed originally to support requirements elicitation, and now are usually used to model user-system interactions and system-system interactions.
- Each use case represents a discrete task that involves external interaction with a system.
- Actors in a use case may be people or other systems.
- Provide an overview of an interaction, and need to add a more detailed textual description.

Key points of Use-case Diagram:

- Boundary of the system
- Roles
  - different kind of users
  - outside systems
- Use-case
  - name after a verb
  - represents a function
- Relationships
  - use
  - include
  - extend
  - generalize
- Use-case description



Description of login use-case:

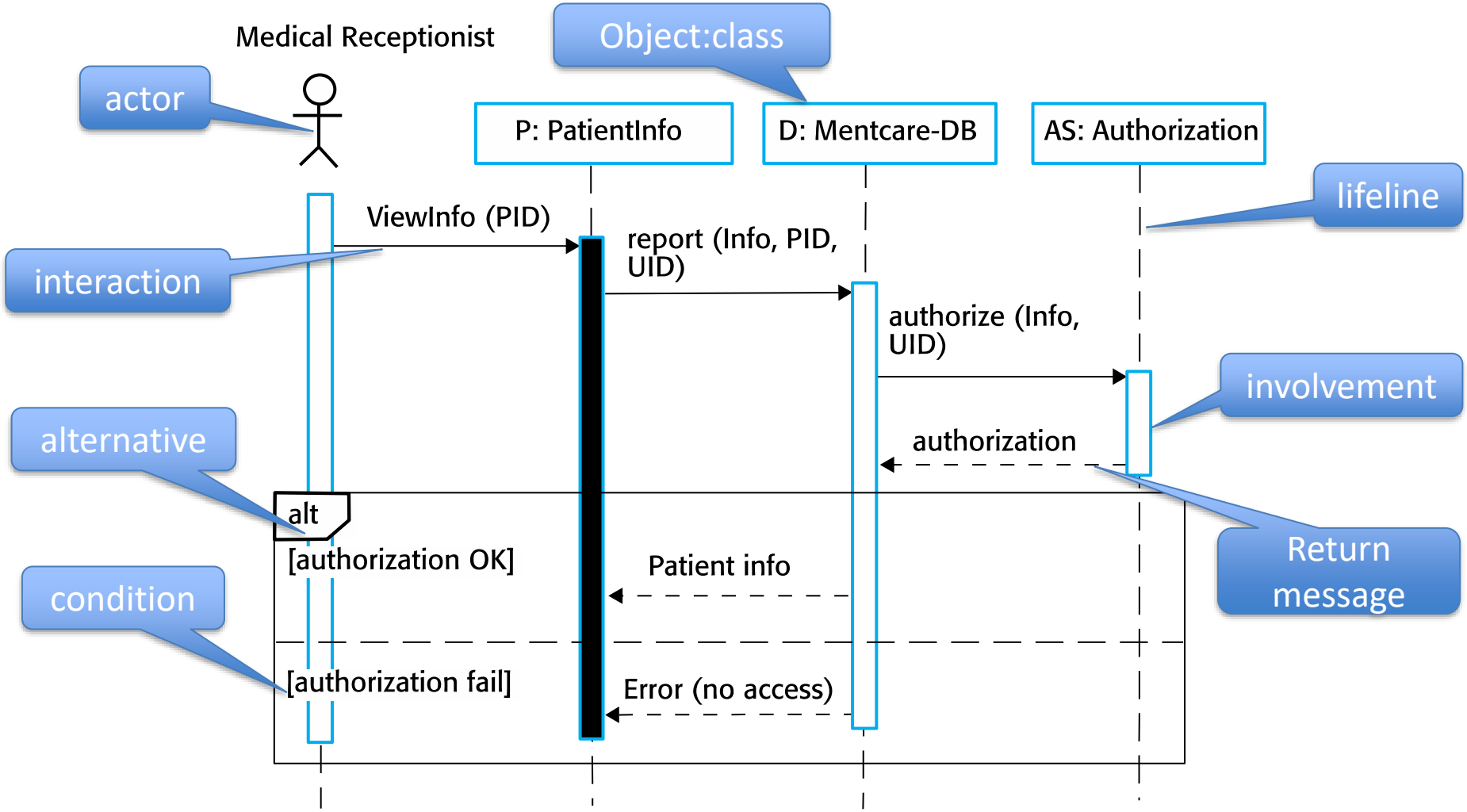
Use Case Name	Login
Use case Description	A user login to System to access the functionality of the system.
Actors	Parents, Students, Teacher, Admin
Pre-Condition	System must be connected to the network.
Post -Condition	After a successful login a notification mail is sent to the User mail id

Main Scenarios	Serial No	Steps
Actors/Users	1	Enter username Enter Password
	2	Validate Username and Password
	3	Allow access to System
Extensions	1a	Invalid Username System shows an error message
	2b	Invalid Password System shows an error message
	3c	Invalid Password for 4 times Application closed

Sequence diagrams

- Sequence diagrams are part of the UML and are used to model the interactions between the actors and the objects within a system.
  - Objects are arranged horizontally across the top;
  - Time is represented vertically so models are read from top to bottom;
  - Interactions are represented by labelled arrows, different styles of arrow represent different types of interaction;
  - A thin rectangle in an object lifeline represents the time when the object in action.

Example: Sequence diagram for View patient information



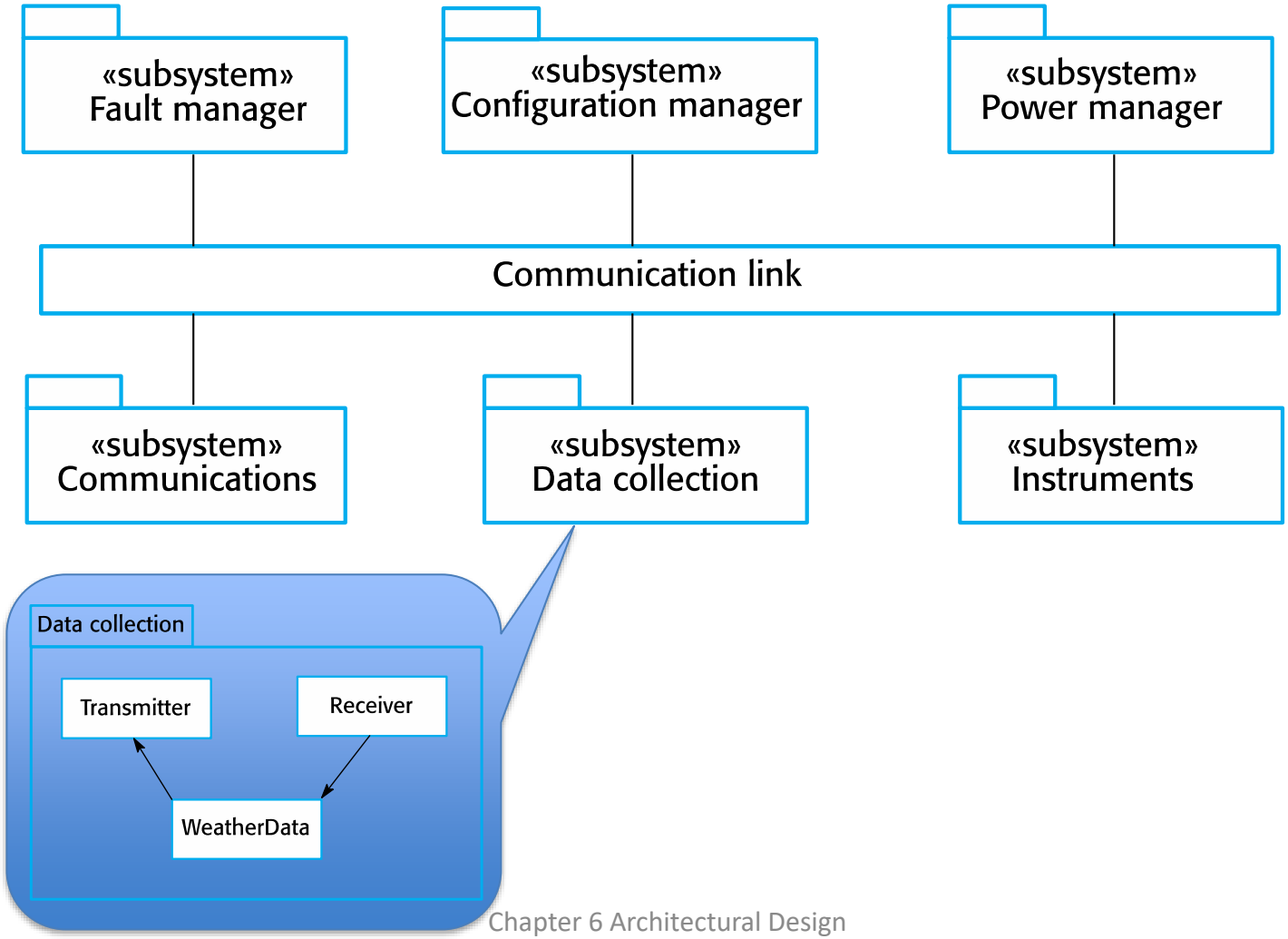
System Architecture & Architectural patterns

Representing architectural views

High-level system architectures are used to explain the system architecture to stakeholders and to inform architectural decision making.

- ✧ UML architecture models
  - Package diagram
  - Deployment diagram
  - Component diagram
- ✧ Architectural patterns

Example: High-level architecture of the weather station



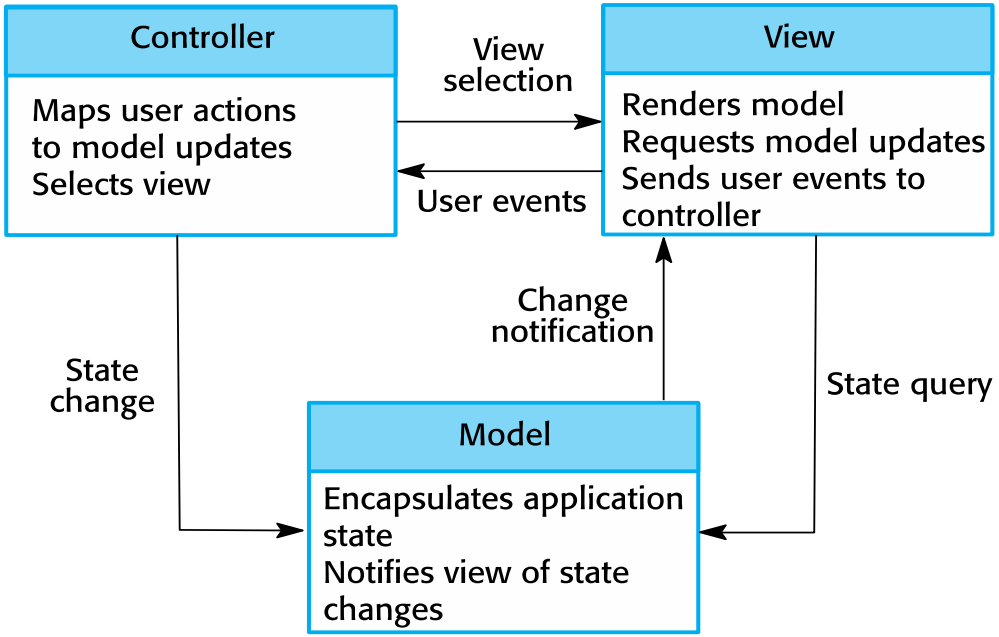
Architectural patterns

- Patterns are a means of representing, sharing and reusing knowledge.
- An architectural pattern is a stylized description of good design practice, which has been tried and tested in different environments.
- Patterns should include information about when they are and when the are not useful.
- Patterns may be represented using tabular and graphical descriptions.

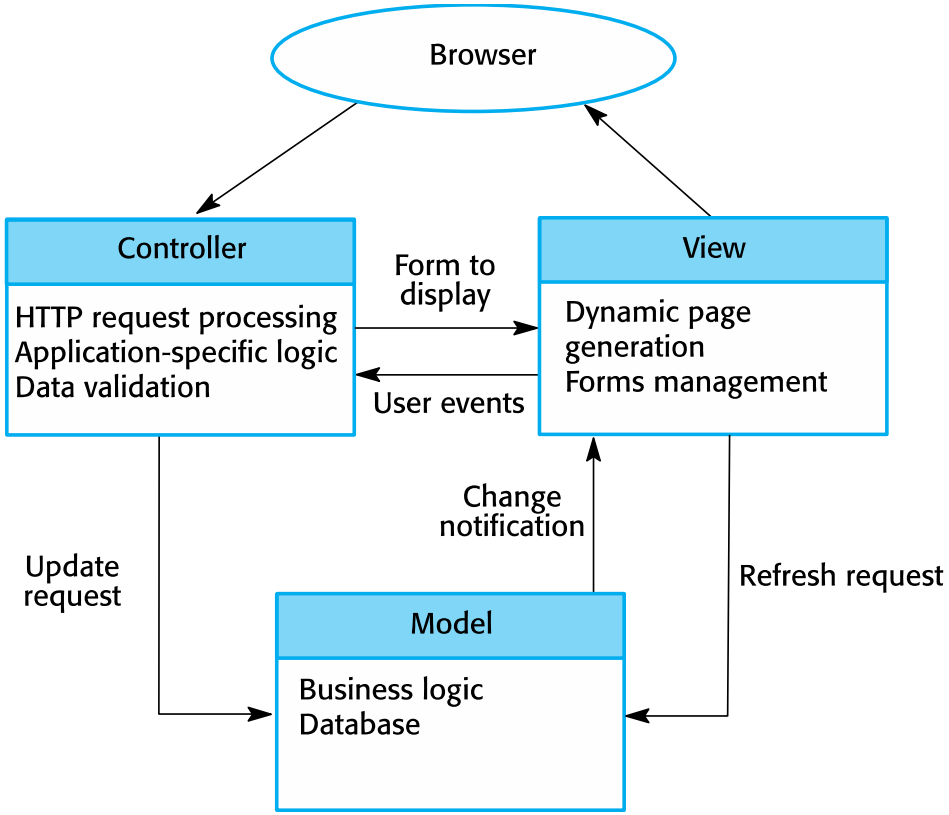
The Model-View-Controller (MVC) pattern

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



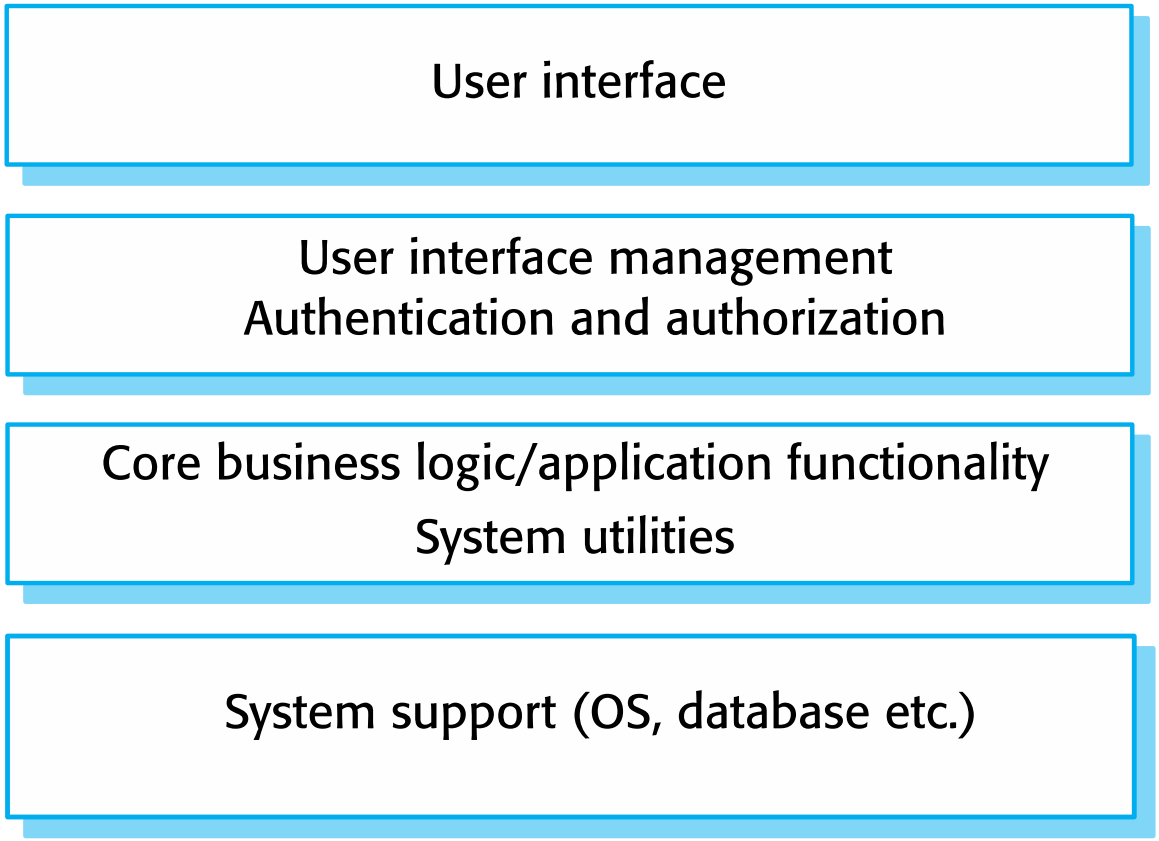
Layered architecture

- Used to model the interfacing of sub-systems.
- Organizes 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.

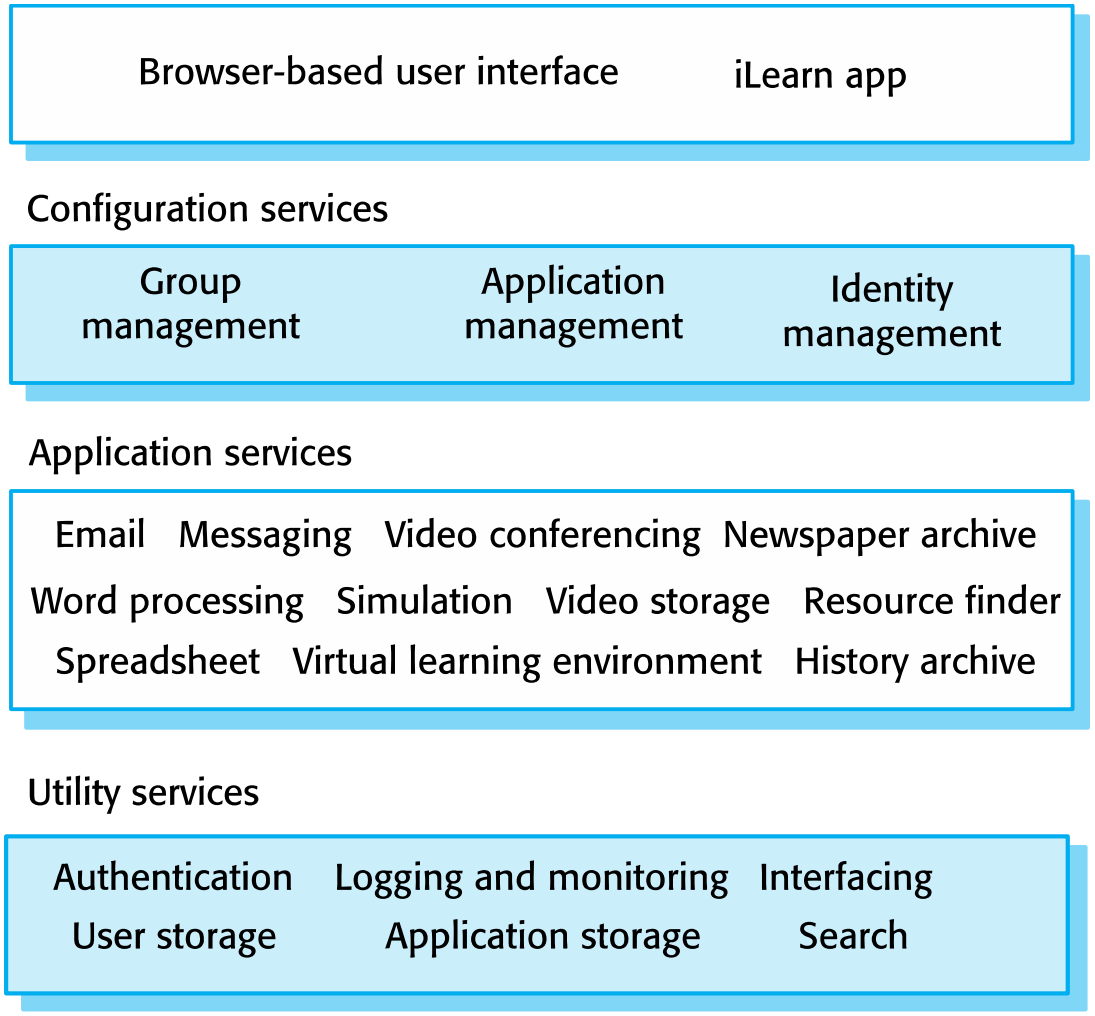
The Layered architecture pattern

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



Example: iLearn system architecture using layered architecture



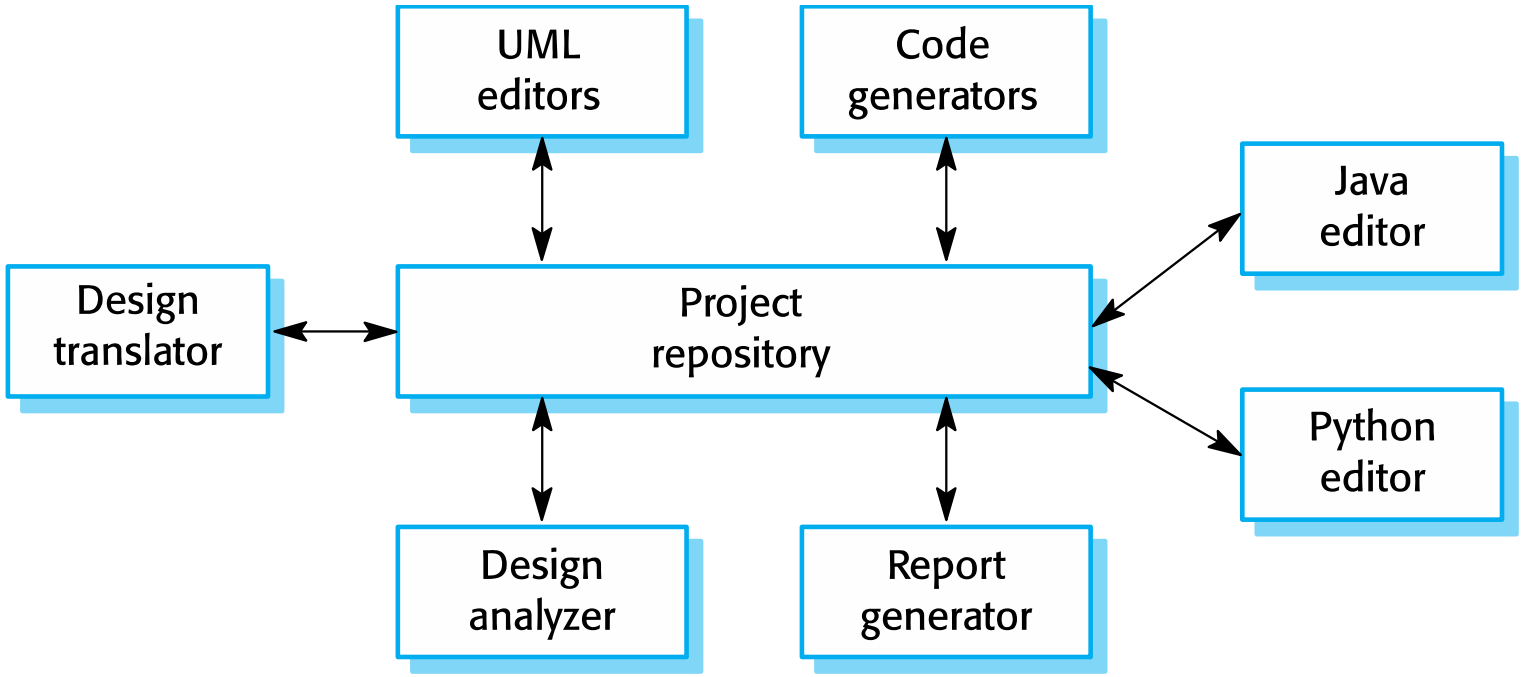
Repository architecture

- When sub-systems must exchange data, the sharing may be done in two ways:
  - Shared data is held in a central database or repository and may be accessed by all sub-systems;
  - Each sub-system maintains its own database and passes data explicitly to other sub-systems.

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 6.9 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



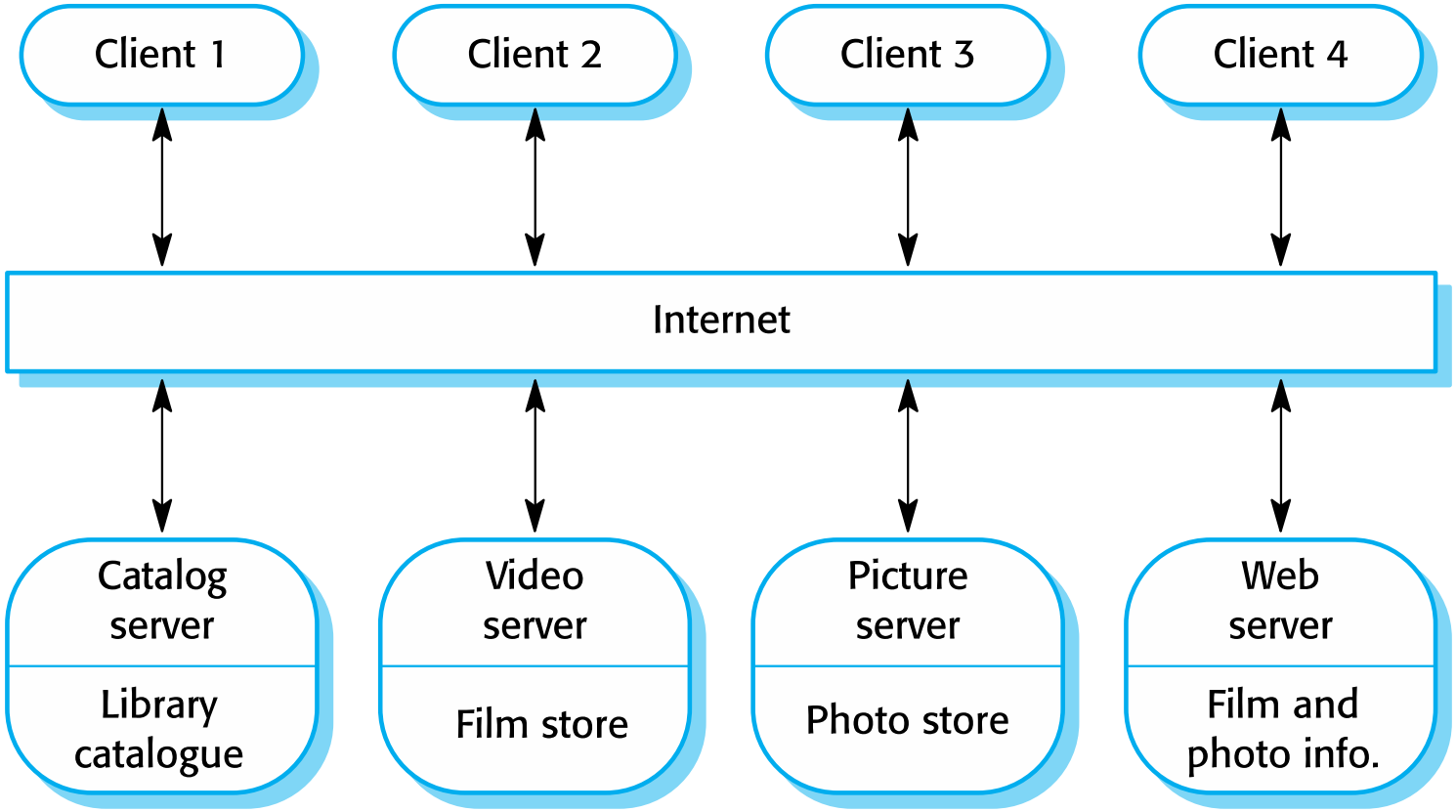
Client-server architecture

- 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.

Example: A client–server architecture for a film library



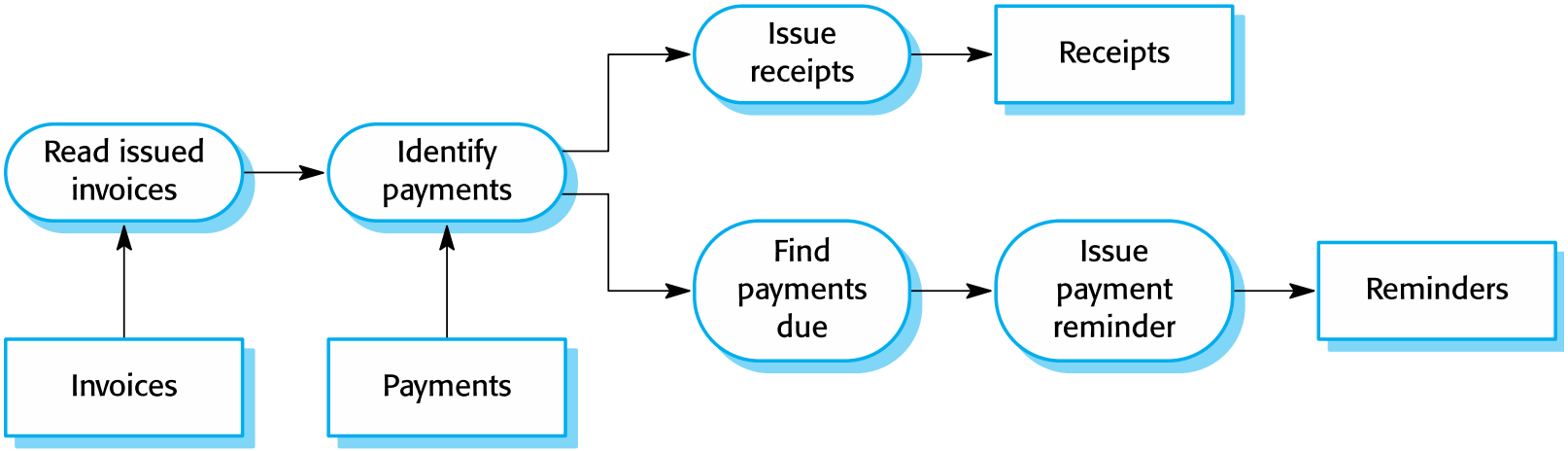
Pipe and filter architecture

- Functional transformations process their inputs to produce outputs.
- When transformations are sequential, this is a batch sequential model which is extensively used in data processing systems.
- Not really suitable for interactive systems.

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 6.13 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 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.

Example: pipe and filter architecture used in a payments system



Application architectures

Application architectures

- An application architecture can be expressed based on different architecture models or patterns.
- A generic application architecture is an architecture for a type of software system that may be configured and adapted to create a system that meets specific requirements.

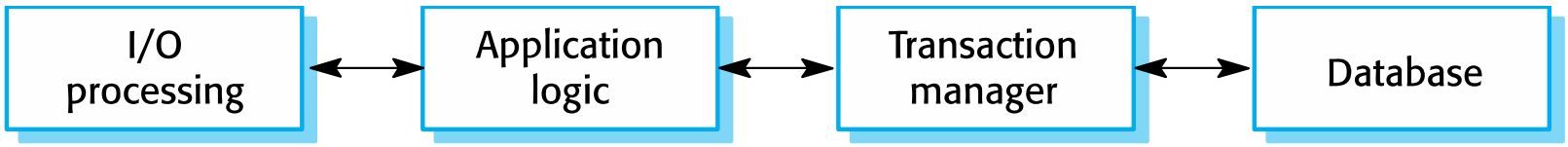
Application type examples

- Two very widely used generic application architectures are transaction processing systems and language processing systems.
- Transaction processing systems
  - Data-centered applications that process user requests and update information in a system database.
  - Such as E-commerce systems, Reservation systems.
- Language processing systems
  - Applications where the users' intentions are specified in a formal language that is processed and interpreted by the system.
  - Such as Compilers, Command interpreters.

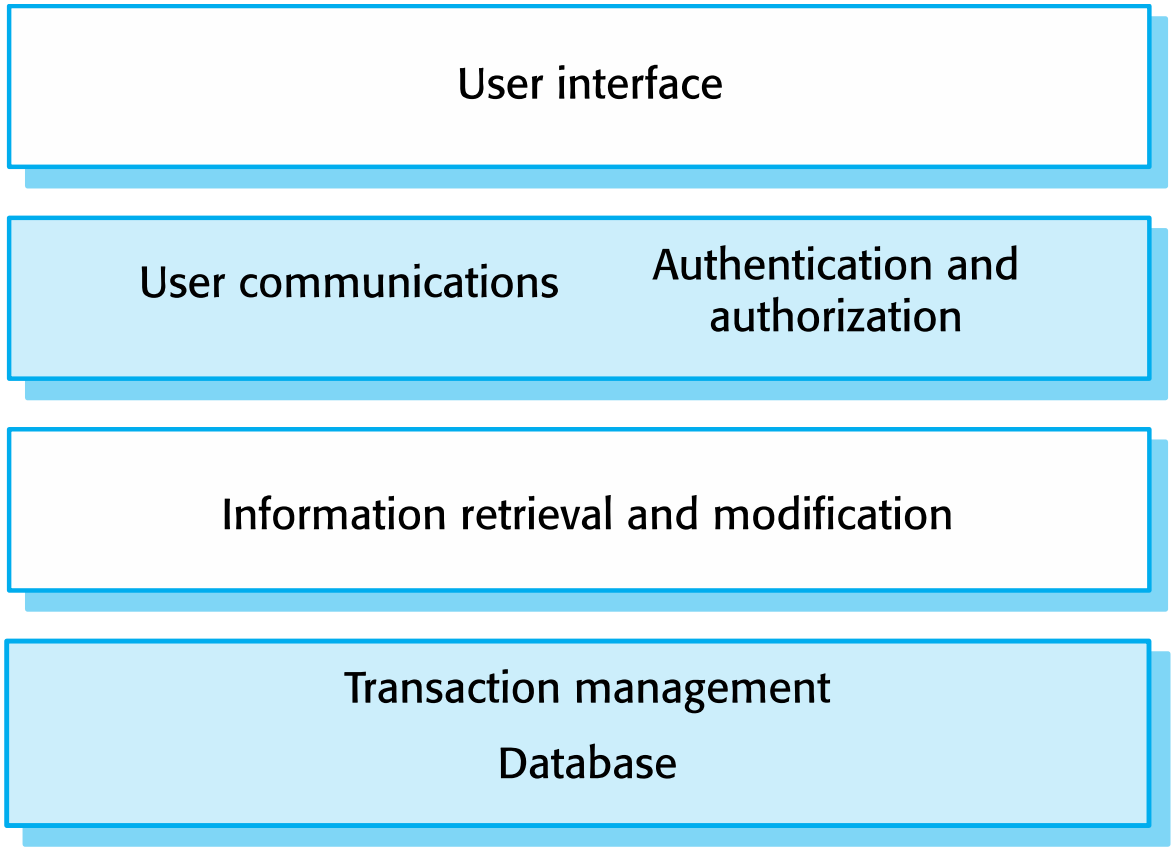
Transaction processing systems

- Process user requests for information from a database or requests to update the database.
- From a user perspective a transaction is: Any coherent sequence of operations that satisfies a goal.
- Users make asynchronous requests for service which are then processed by a transaction manager.

The structure of transaction processing applications



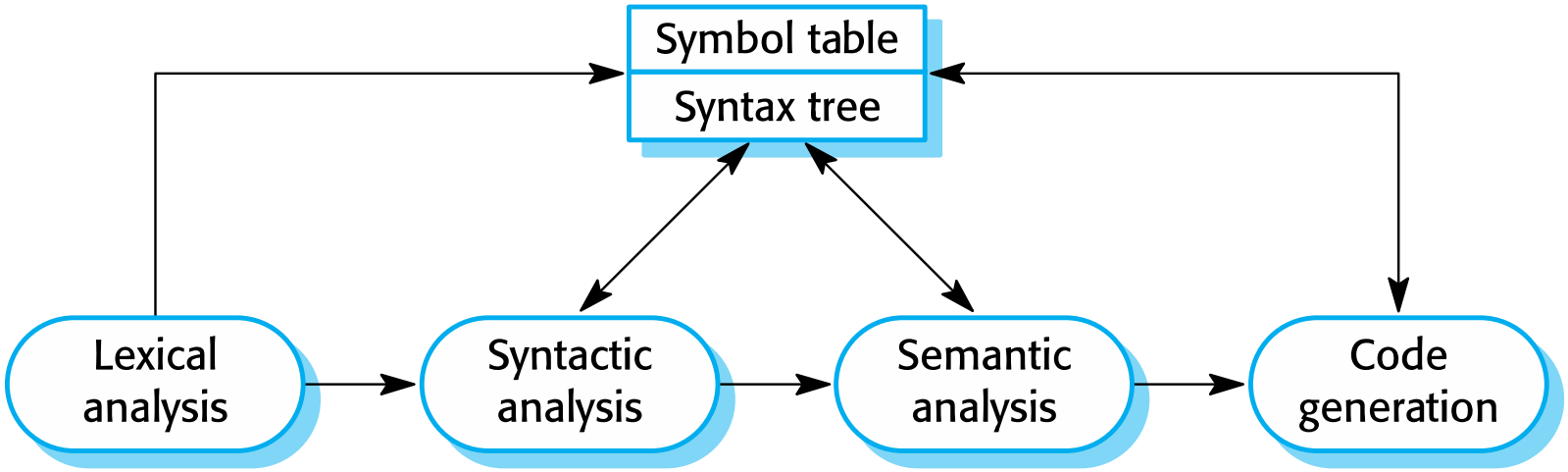
Layered information system architecture



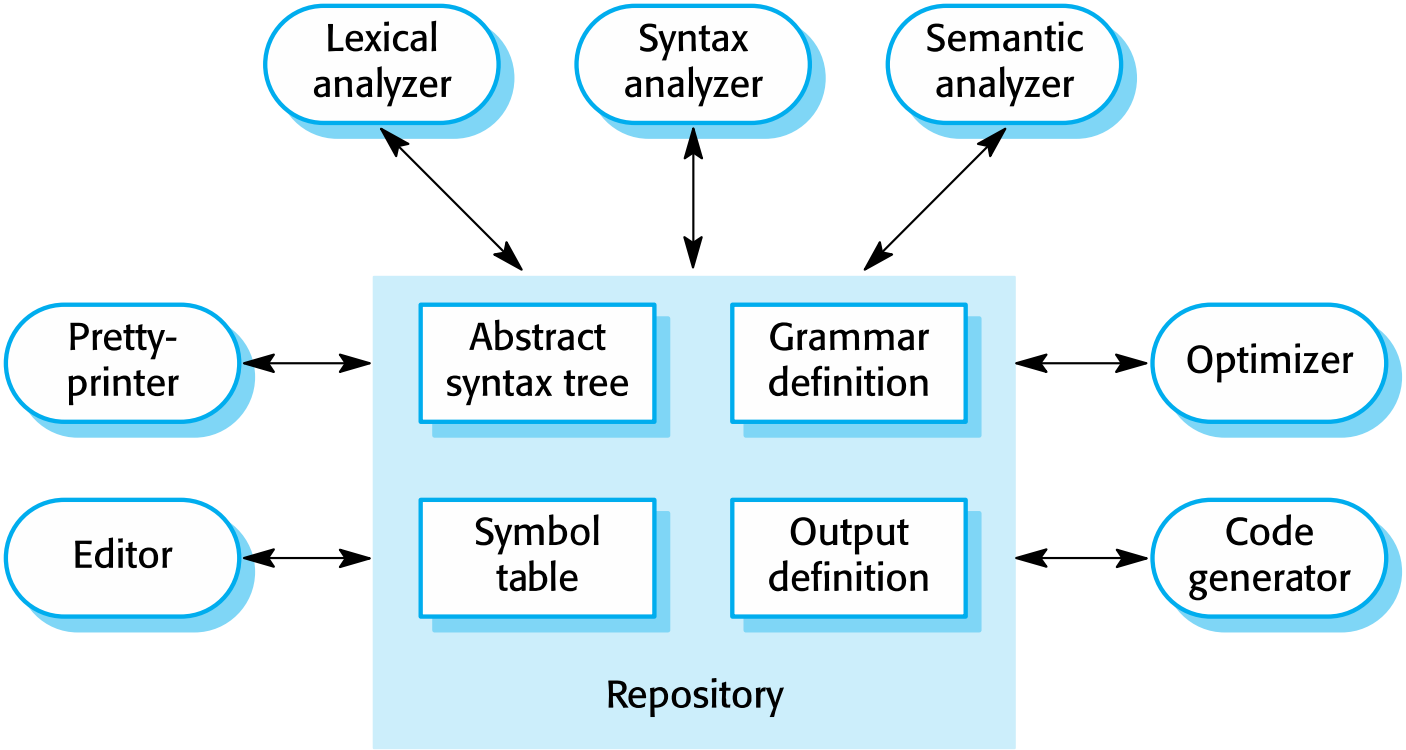
Language processing systems

- Accept a natural or artificial language as input and generate some other representation of that language.
- May include an interpreter to act on the instructions in the language that is being processed.

A pipe and filter compiler architecture



A repository architecture for a language processing system



lecture6：Software Architecture design

- **Reading for this week:**
  - 'Agility and Architecture: Can They Coexist?' by Abrahamsson, P., Babar, M.A., and Kruchten
- - Chapter 6: Architecture Design from the coursebook (Software Engineering by Ian Sommerville)
- **Watch the video:**
- https://www.youtube.com/watch?v=nTtQwGoUUNc（UML 2 Deployment Diagrams，8min）
- **Assignments for this week:**
  - Quiz 6: 2 attempts, 30 minutes time
  - Essay5: The Role and Importance of Software Architecture in Agile Development.