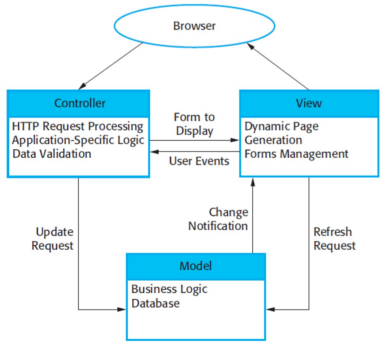
Architectural Design and OOD

Monday, September 7, 2020 8:39 PM

1. **Architectural design**
   a. **Introduction:**
      - It is concerned with understanding how a system should be organized and designing the overall structure of that system.
      - It is the critical link between design and requirements engineering, as it identifies the main structural components in a system and the relationships between them. The output of the architectural design process is an architectural model that describes how the system is organized as a set of communicating components.
      - In agile processes, it is generally accepted that an early stage of the development process should be concerned with establishing an overall system architecture. Incremental development of architectures is not usually successful as refactoring a system architecture is likely to be expensive.
      - Software architecture is important because it affects the performance, robustness, distributability, and maintainability of a system (Bosch, 2000).
      - Different architecture is used for development of software for embedded system and different architecture is used for the development of a software for distributed system.
      - **Architectural Views:** There are different opinions as to what views are required. Krutchen (1995), in his well-known 4+1 view model of software architecture, suggests that there should be four fundamental architectural views, which are related using use cases or scenarios. They are:
        - **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. It is useful for making judgments about non-functional system characteristics such as performance and availability.
        - **Development View:** shows how the software is decomposed for development i.e. it shows the breakdown of the software into components that are implemented by a single developer or development team. It is useful for software managers and programmers.
        - **Physical View:** shows the system hardware and how software components are distributed across the processors in the system. It is useful for systems engineers planning a system deployment.

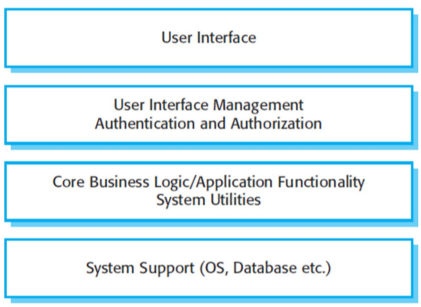
   b. **System structuring with advantages and disadvantages**

   Architectural Patterns: The idea of patterns as a way of presenting, sharing, and reusing knowledge about software systems is now widely used. The trigger for this was the publication of a book on object-oriented design patterns (Gamma et al., 1995). You can think of architectural pattern as a stylized, abstract description of good practice, which has been tried and tested in different systems and environments. For example: **Model-View-Controller (MVC)** pattern which is the basis of interaction management in many web-based systems.



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 dicks, 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.

Layered Architecture: The layered architecture pattern is another way of achieving separation and independence. Here, the system functionality is organized into separate layers, and each
layer only relies on the facilities and services offered by the layer immediately beneath it. It supports the incremental development of systems.



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.

Repository Architecture: describes how a set of interacting components can share data. The majority of systems that use large amounts of data are organized around a shared database or repository. This model is therefore suited to applications in which data is generated by one component and used by another.

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.

Client-server Architecture: the repository architecture is concerned with the static structure of a system and does not show its run-time organization for distributed system. A system that follows the client–server pattern is organized as a set of services and associated servers, and clients that access and use the services.

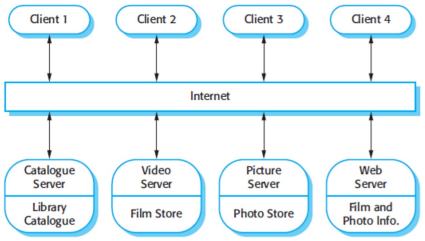
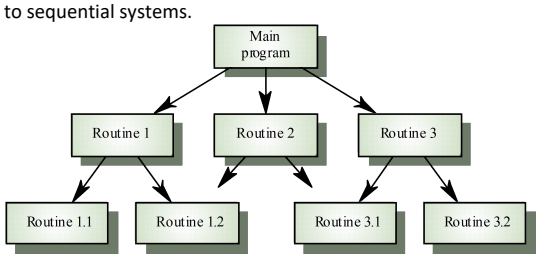


Figure 6.11 A client–server architecture for a film library

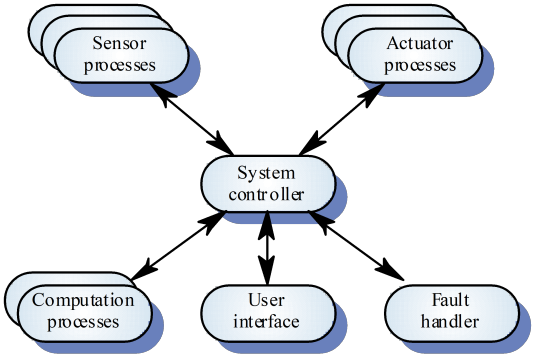
Name	Client–server
Description	In a client–server architecture, the functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.
Example	Figure 6.11 is an example of a film and video/DVD 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.

Abstract Model: It is used to model the interfacing of sub-systems. It organizes the system into a set of layers each of which provide a set of services. It supports the incremental development of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected.

c. **Control models:** these are concerned with the control flow between sub-systems and are distinct from the system decomposition model.
   i. **Centralized control:** one sub-system has overall responsibility for control and starts and stops other sub-systems.
      1) **Call-return model**: top-down subroutine model where control starts at the top of a subroutine hierarchy and moves downwards. It is applicable to sequential systems.

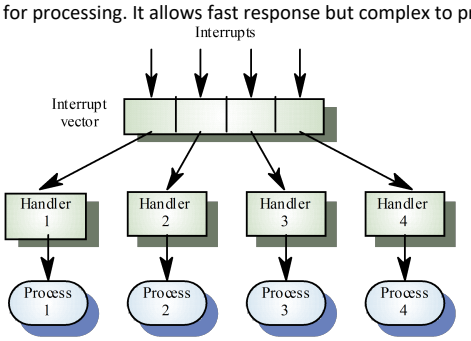


      2) **Manager model**: applicable to concurrent systems. One system component controls the stopping, starting and coordination of other system processes. Example: real-time system control



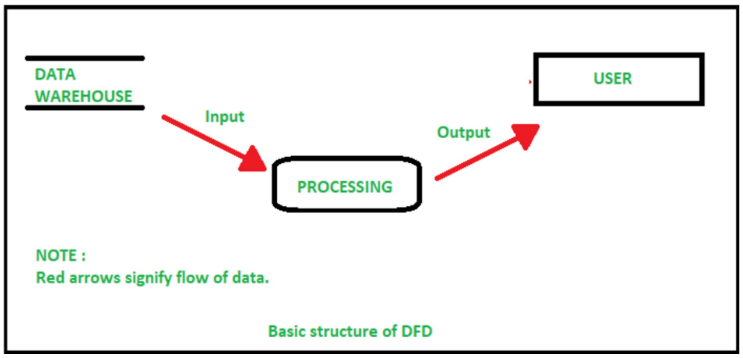
ii. **Event-based control:** each sub-system can respond to externally generated events from other sub-systems or the system's environment. Two principal event-driven models:

1) **Broadcast models:** an event is broadcast to all sub-systems. Any sub-system which can handle the event may do so. It is effective in integrating sub-systems on different computers in a network. Sub-systems register an interest in specific events. When these occur, control is transferred to the sub-system which can handle the event. Control policy is not embedded in the event and message handler. Sub-systems decide on events of interest to them.
2) **Interrupt-driven models:** used in real-time systems where interrupts are detected by an interrupt handler and passed to some other component for processing. It allows fast response but complex to program and difficult to validate.



d. **Modular decomposition:** sub-systems are decomposed into modules.

i. **Object oriented**: (class diagram) Structure the system into a set of loosely coupled objects with well-defined interfaces. It is concerned with identifying object classes, their attributes and operations. When implemented, objects are created from these classes and some control model used to coordinate object operations.
ii. **Structured**: Data-flow model.



iii. **Domain specific architecture:** architectural models which are specific to some application domain.

1) **Generic models:** they are abstractions from a number of real systems and which encapsulate the principal characteristics of these systems. Examples: compiler model, language processing system, etc. Normally bottom-up model.
2) **Reference models:** they are more abstract, idealized model, which provide a means of information about that class of system and of comparing different architectures. Example: Open Systems Interconnections (OSI) model. Normally top-down model.

2. **Object Oriented Design**

a. **Introduction:** an object-oriented system is made up of interacting objects that maintain their own local state and provide operations on that state. They are easier to change than systems developed using functional approaches. Changing the implementation of an object or adding services should not affect other system objects.
b. **Features of object oriented design**

i. **Features of OOP**

1) Abstraction
2) Encapsulation
3) Inheritance
4) Polymorphism
5) Relationships
   - *Aggregation*: indicates relationship between a whole and its parts.
   - *Association*: Here, two classes are related or connected in some way such as one class works with another to perform a task or one class acts upon other class.
   - *Generalization*: The child class is based on parent class. It indicates that two classes are similar but have some differences.

ii. **Characteristics of good object oriented design**

1) **Low coupling**: coupling is the measure of degree of independence between the modules. Low coupling is also known as loose coupling and it is regarded as a good programming practice. Tight coupling is regarded as a bad programming practice.
2) **High cohesion**: degree to which the elements of a module belong together i.e. cohesion measures the strength of relationships between pieces of functionality within a given module. High cohesion is regarded as a good programming practice whereas low cohesion is regarded as a bad programming practice.

Example of Low Cohesion:

```
------------------
| Staff          |
------------------
| checkEmail()   |
| sendEmail()    |
| emailValidate()|
| PrintLetter()  |
------------------
```

Example of High Cohesion:

```
----------------------------
| Staff                    |
----------------------------
| -salary                  |
| -emailAddr               |
-----------------------------
| setSalary(newSalary)     |
| getSalary()              |
| setEmailAddr(newEmail)   |
| getEmailAddr()           |
----------------------------
```

3) **Depth of Inheritance tree:** the peak of inheritance tree must not be giant.
4) **Number of methods:** object's must not have too several ways

c. **Design model (UML: Unified Modeling Language))**

Shows the objects and object classes and relationships between these entities. Static models describe the static structure of the system in terms of object classes and relationships. Dynamic models describe the dynamic interactions between objects.

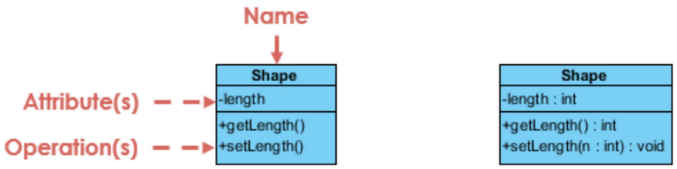
Diagrams/Models that are used in UML:

- Structural Diagrams
  - Class diagram
  - Component diagram
  - Deployment diagram
  - Object diagram
  - Package diagram, etc.
- Behavioral Diagrams
  - Use case diagram
  - Activity diagram
  - State machine diagram
  - Sequence diagram
  - Timing diagram, etc.

Models in Object Oriented Design:

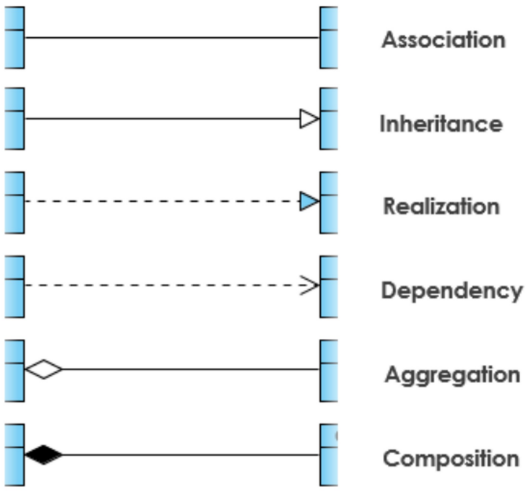
1. **Class Model:** It shows the classes present in the system. It shows the attributes and behavior associated with the objects. Class diagram is used to show the class model.

   Class Diagram: Class diagrams are the main building block of any object-oriented solution. It shows the classes in a system, attributes, and operations of each class and the relationship between each class. In a large system with many related classes, classes are grouped together to create class diagrams. Different relationships between classes are shown by different types of arrows.

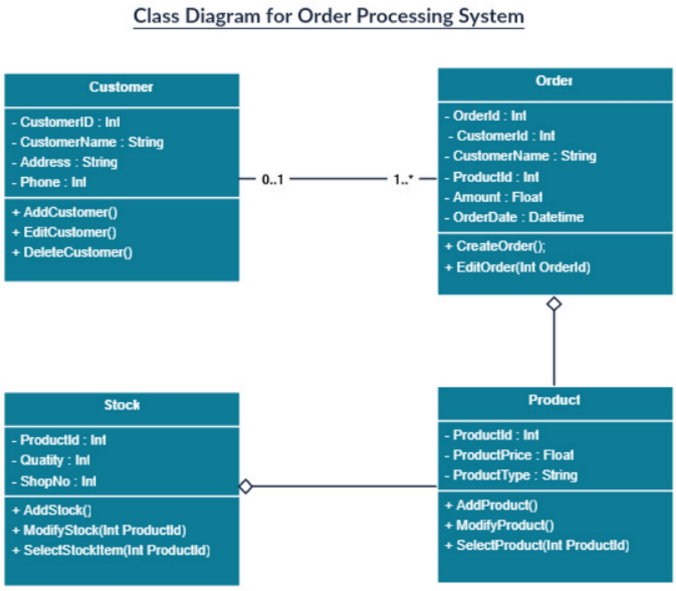


   Generally, a class diagram consists of three parts: name, attributes and methods/operations.

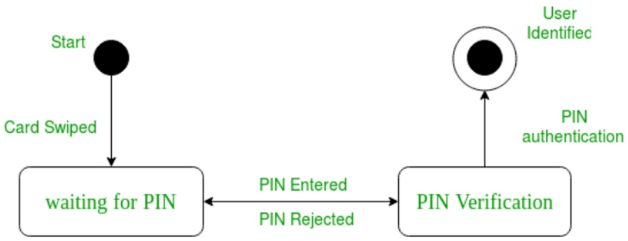
   Relationships between classes:



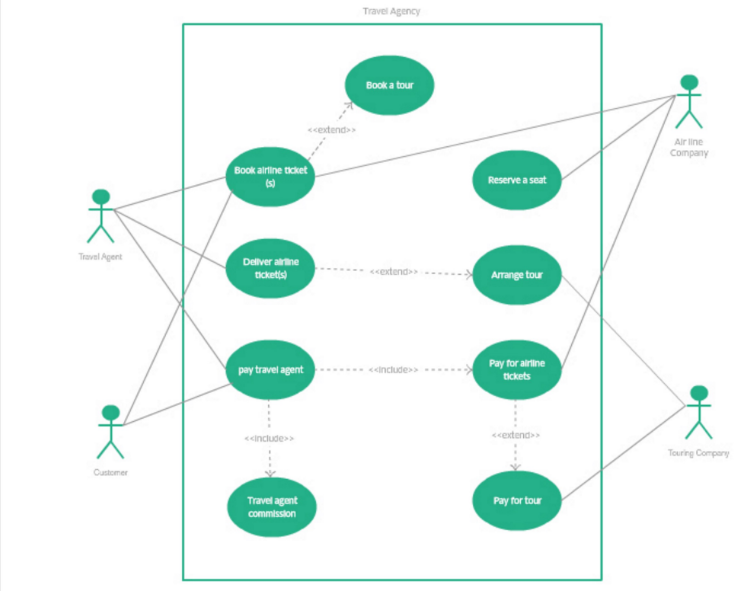
   Example of class diagram:



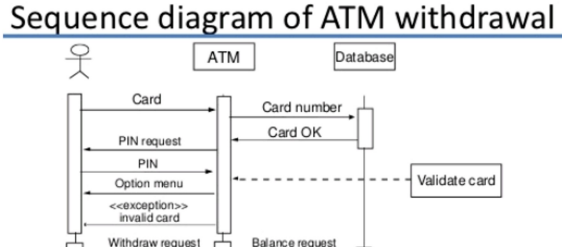
2. **State Model:** State model describes those aspects of objects concerned with time and the sequencing of operations – events that mark changes, states that define the context for events, and the organization of events and states. Actions and events in a state diagram become operations on objects in the class model. State diagram describes the state model. (geeksforgeeks.org)

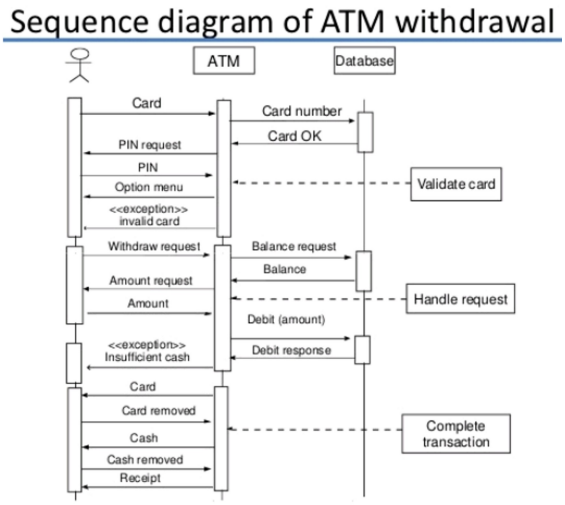


3. **Interaction Model:** Interaction model is used to show the various interactions between objects, how the objects collaborate to achieve the behavior of the system as a whole.
Use case Diagram: Use case diagrams give a graphic overview of the actors involved in a system, different functions needed by those actors and how these different functions interact. It only summarizes some of the relationships between use cases, actors, and systems. It does not show the order in which steps are performed to achieve the goals of each use case. Example:



Sequence diagram: Sequence diagrams in UML show how objects interact with each other and the order those interactions occur. It's important to note that they show the interactions for a particular scenario. Example:





Activity diagram: Activity diagrams represent workflows in a graphical way. They can be used to describe the business workflow or the operational workflow of any component in a system. Sometimes activity diagrams are used as an alternative to State machine diagrams. Example: