System Design

Chapter 13

In this Lecture you will Learn:

- The major concerns of system design
- The main aspects of system architecture, in particular what is meant by subdividing a system into layers and partitions
- How to apply the Model-View-Controller (MVC) architecture
- Which architectures are most suitable for distributed systems
- How design standards are specified

System Architecture

- A major part of system design is defining the **System Architecture**
  - Containing potentially human, software and hardware elements and how these elements are structured and interact
- The software elements of the system embody the **Software Architecture**

System Architecture

- The architecture of the information system is first considered early in the project during the requirements capture and analysis activities.
- This first view of the system architecture is driven significantly by the use cases and then informs the continuing requirements capture and analysis activities.
- This forms a useful basis from which to develop the design architecture.

System Architecture

- The detailed software architecture of a computerized information system develops as the design process continues into class design but it is important to identify an overall system architecture within which the detail can be refined.
- High-level architectural decisions that are made during system design determine how successfully the system will meet its non-functional objectives (e.g. performance, extensibility) and thus its long-term utility for the client.
- **Reuse** is one of the much-vaunted benefits of object-orientation and poor software architecture usually reduces both the reusability of the components produced and the opportunity to reuse existing components.

System Design Activities

- Sub-systems and major components are identified
- Any inherent concurrency is identified
- Sub-systems are allocated to processors
- A data management strategy is selected
- A strategy and standards for Human-Computer Interaction (HCI) are chosen
- Code development standards are specified
- The control aspects of the application are planned
- Test plans are produced
- Priorities are set for design trade-offs
- Implementation requirements are identified (e.g. Data conversion)

Software Architecture Definition

- Buschmann et al. (1996) give the following definition of a software architecture:
  - A **Software Architecture** is a *description of the subsystems and components of a software system and the relationships between them*.
  - Sub-systems and components are typically specified in different views to show the relevant functional and non-functional properties of a software system.
  - The software architecture of a system is an artefact.
  - It is the result of the software design activity.

Sub-systems

- A sub-system typically groups together elements of the system that share some common properties
- An object-oriented sub-system encapsulates a coherent set of responsibilities in order to ensure that it has integrity and can be maintained
  - For example, the elements of one sub-system might all deal with the human-computer interface, the elements of another might all deal with data management and the elements of a third may all focus on a particular functional requirement.

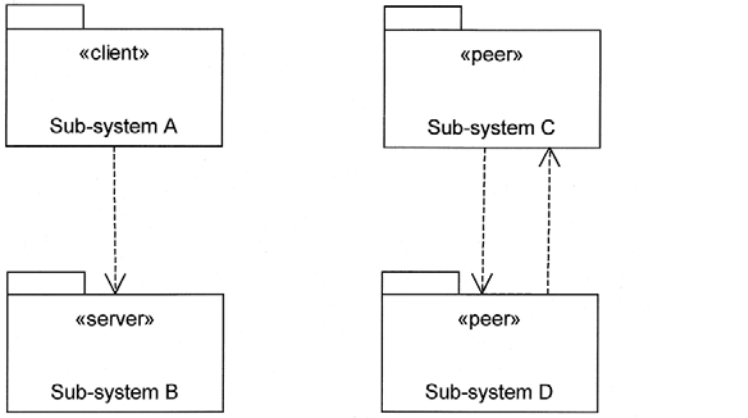
Advantages of Sub-systems

- The sub-division of an information system into sub-systems has the following advantages
  - It produces smaller units of development
  - It helps to maximize reuse at the component level
  - It helps the developers to cope with complexity
  - It improves maintainability
  - It aids portability

Styles of Communication between Sub-systems

- Each sub-system provides services for other sub-systems, and there are two different styles of communication that make this possible
- These are known as:
  - Client-Server Communication
  - Peer-to-Peer Communication

Styles of Communication between Sub-systems

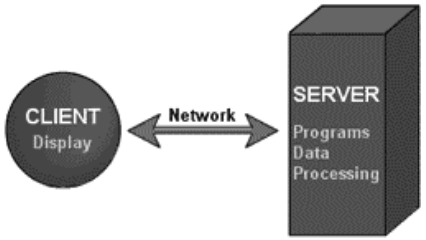


The server sub-system does not depend on the client sub-system and is not affected by changes to the client's interface.

Each peer sub-system depends on the other and each is affected by changes in the other's interface.

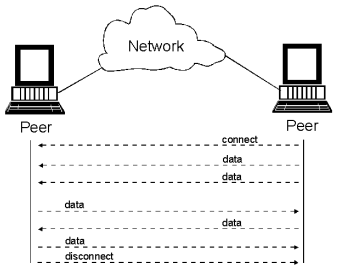
Client/Server Communication

- Client/Server communication requires the client to know the interface of the server sub-system, but the communication is only in one direction
- The client sub-system requests services from the server sub-system and not vice versa



Peer-to-Peer Communication

- Peer-to-peer communication requires each sub-system to know the interface of the other, thus coupling them more tightly
- The communication is two way since either peer sub-system may request services from the other

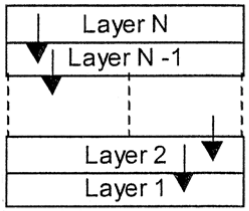


Layering and Partitioning

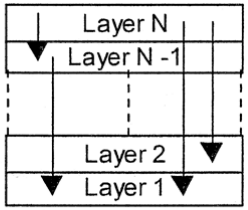
- Two general approaches to the division of a software system into sub-systems
  - **Layering –** the different sub-systems usually represent different levels of abstraction
  - **Partitioning –** each sub-system focuses on a different aspect of the functionality of the system as a whole
- Both approaches are often used together on one system, so that some of its sub-systems are divided by layering, while others are divided by partitioning.

Schematic of a Layered Architecture

- Layered architectures can be either **Open** or **Closed**



Closed architecture—messages may only be sent to the adjacent lower layer.



Open architecture—messages can be sent to any lower layer.

Open Architecture

- Produces more compact code, the services of all lower level layers can be accessed directly by any layer above them without the need for extra program code to pass messages through each intervening layer, but this breaks the encapsulation of the layers
- More difficult to maintain because each layer may communicate with all lower layers hence increasing the degree of coupling in the architecture. A change to one layer may ripple to many layers.

Closed Architecture

- Minimizes dependencies between the layers and reduces the impact of a change to the interface of any one layer
- Require more processing, as messages have to be passed through intervening layers.

Example: OSI 7 Layer Model

Layer 7: Application
Provides miscellaneous protocols for common activities.

Layer 6: Presentation
Structures information and attaches semantics.

Layer 5: Session
Provides dialogue control and synchronization facilities.

Layer 4: Transport
Breaks messages into packets and ensures delivery.

Layer 3: Network
Selects a route from sender to receiver.

Layer 2: Data Link
Detects and corrects errors in bit sequences.

Layer 1: Physical
Transmits bits: sets transmission rate (baud), bit-code, connection, etc.

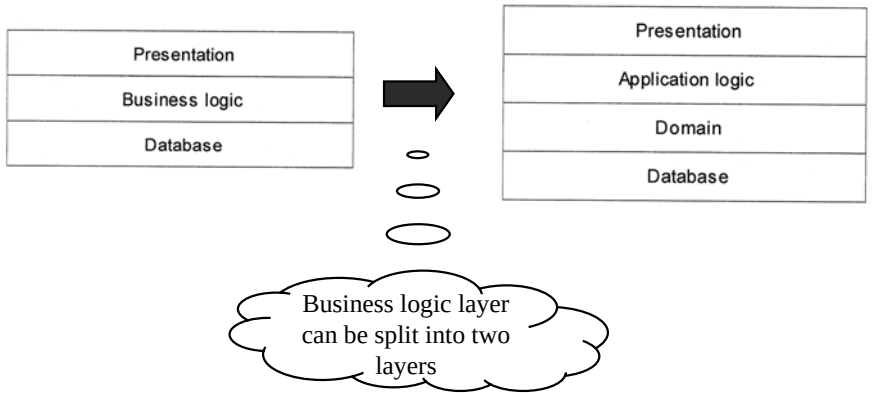
Applying a Layered Architecture

- Buschamn et. al. (1996) suggest that a series of issues need to be addressed when applying a layered architecture in an application. These include:
  - Maintaining the stability of the interfaces of each layer
  - The construction of other systems using some of the lower layers
  - Variations in the appropriate level of granularity for sub-systems
  - The further sub-division of complex layers
  - Performance reductions due to a closed layered architecture

Developing a Layered Architecture

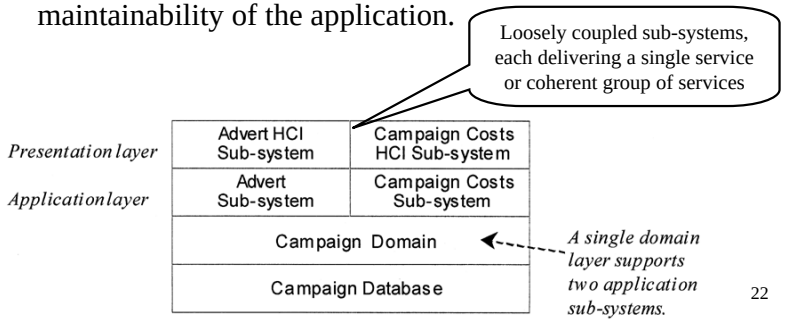
1. Define the criteria by which the application will be grouped into layers. A commonly used criterion is level of abstraction from the hardware.
2. Determine the number of layers.
3. Name the layers and assign functionality to them.
4. Specify the services for each layer.
5. Refine the layering by iterating through steps 1 to 4.
6. Specify interfaces for each layer.
7. Specify the structure of each layer. This may involve partitioning within the layer.
8. Specify the communication between adjacent layers (this assumes that a closed layer architecture is intended).
9. Reduce the coupling between adjacent layers. This effectively means that each layer should be strongly encapsulated.

Three & Four Layer Architectures



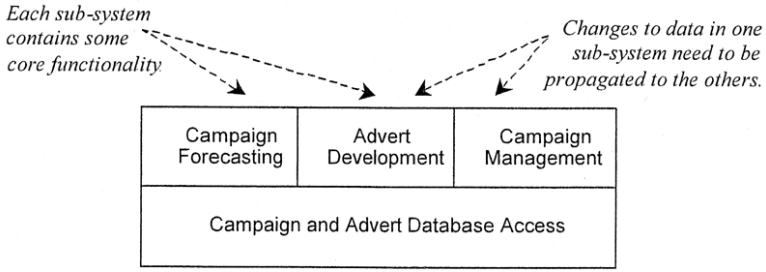
Partitioned Sub-systems

- The figure shown the Four layer architecture applied to part of the Agate campaign management system
- This would result in more compact code and perhaps improve the performance of the application. However, as mentioned above, it is likely to reduce the extensibility and maintainability of the application.



Problems with some Architectures

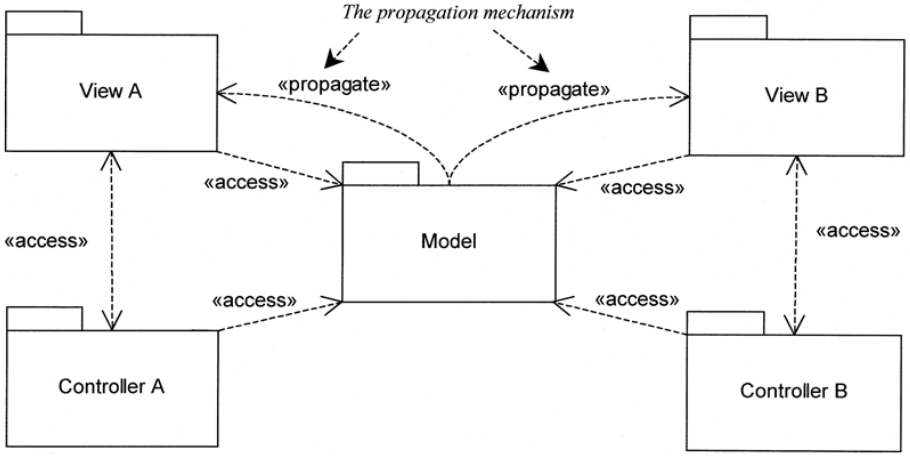
- Multiple interfaces for the same core functionality



Difficulties

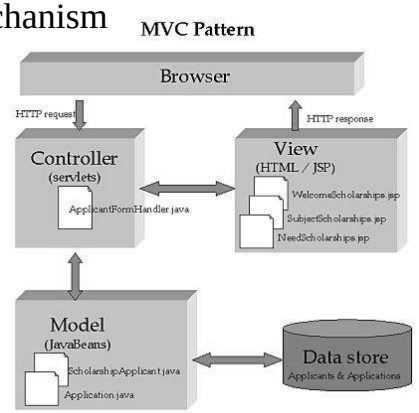
- Some of the difficulties that need to be resolved for this type of application
  - The same information should be capable of presentation in different formats in different windows
  - Changes made within one view should be reflected immediately in the other views
  - Changes in the user interface should be easy to make
  - Core functionality should be independent of the interface to enable multiple interface styles to co-exist

Basic Structure of Model-View-Controller (MVC)



Model-View-Controller Components

- The responsibilities of the components of an MVC architecture are listed below:
  - Propagation Mechanism
  - Model
  - View
  - Controller



Model-View-Controller: Model

- Provides the central functionality of the application and is aware of each of its dependent view and controller components.

Model-View-Controller: View

- Corresponds to a particular style and format of presentation of information to the user.
- The view retrieves data from the model and updates its presentations when data has been changed in one of the other views.
- The view creates its associated controller.

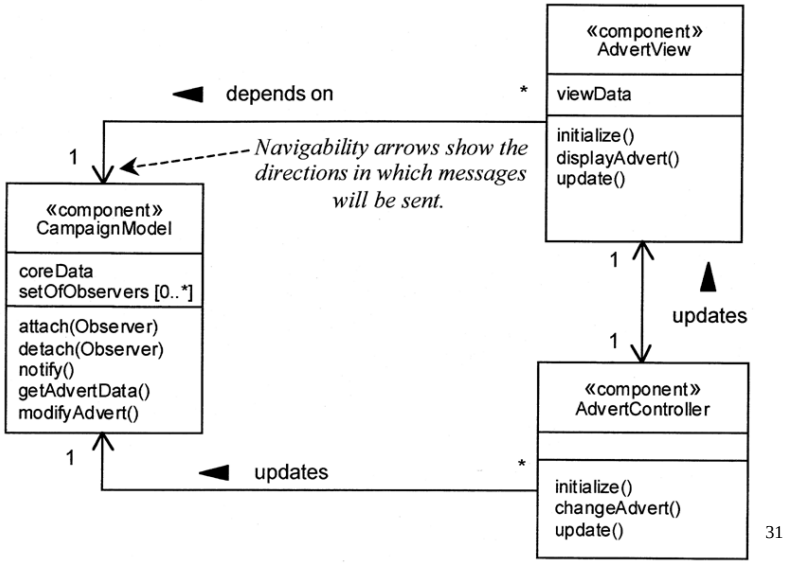
Model-View-Controller: Controller

- Accepts user input in the form of events that trigger the execution of operations within the model.
- These may cause changes to the information and in turn trigger updates in all the views ensuring that they are all up to date.

Model-View-Controller: Propagation Mechanism

- Enables the model to inform each view that the model data has changed and as a result the view must update itself.
- It is also often called the **Dependency Mechanism**.

Responsibilities of MVC Components



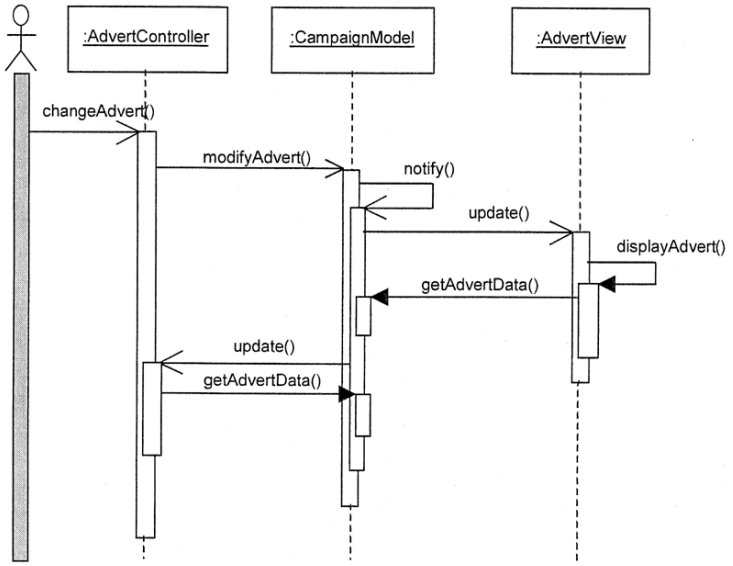
Explanation

- The operation `update()` in the `AdvertView` and `AdvertController` components triggers these components to request data from the `CampaignModel` component[1]. This model component has no knowledge of the way that each view and controller component will use its services. It need only know that all view and controller components must be informed whenever there is a change of state (a modification either of object attributes or of their links).
- The `attach()` and `detach()` services in the `CampaignModel` component enable views and controllers to be added to the `setOfObservers`. This contains a list of all components that must be informed of any change to the model core data. In practice there would be separate views, each with its own controller, to support the requirements of the campaign manager and the director.
  - Note: *[1] In this example the CampaignModel will hold details of campaigns and their adverts.*

MVC Component Interaction

- In the MVC when a change to the model data is detected by the model it informs each registered view/controller that some change has been made but does not provide any detail of the change (in the standard use of the architecture at least).
- It is then up to each view/controller pair to request from the model the latest version of data they require so that they can update themselves.

MVC Component Interaction



Class Exercise

- What are the main differences between the MVC architecture and the layered and partitioned architecture?

Architectures for the Distributed System

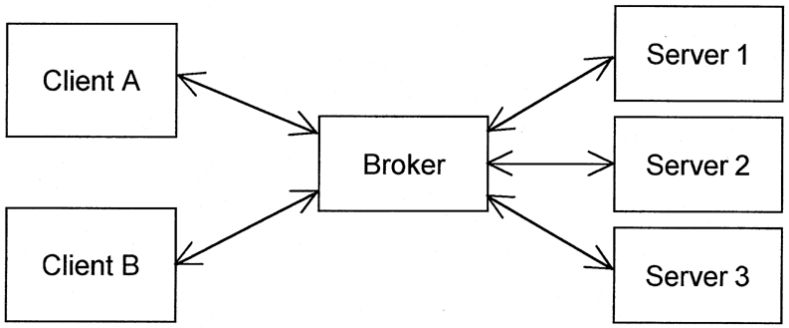
- Distributed information systems are becoming more common as communications technology improves and becomes more reliable.
- An information system might be distributed over computers at the same location or different locations.
- An architecture that is suitable for distributed information system need also to be flexible so that it can cope with change
- A distributed information system may be support by software products such as distributed database management system or object request brokers

Communication between Sub-systems

- A broker decouples sub-systems by acting as an intermediate messaging-passing component through which all messages are passed.
- As a result a sub-system is aware of the broker and not directly in communication with the other sub-systems.
- This makes it easier to move the sub-systems to distributed computers.

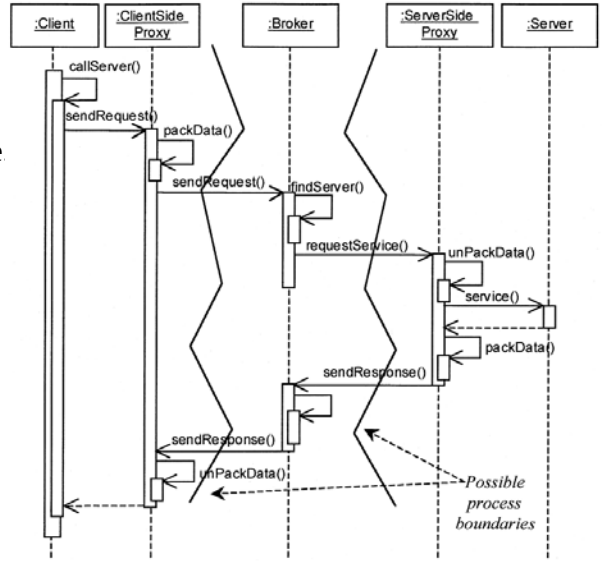
Schematic of Simplified Broker Architecture

- A general Broker architecture for distributed system is described by Buschmann et al. (1996)
- A simplified version of the broker architectures is shown:



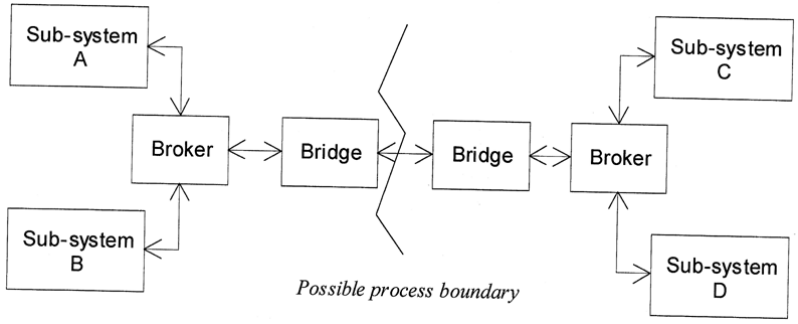
Broker Architecture for Local Server

- The sequence diagram for Client-Server communication using the Broker Architecture
- The diagram is drawn with asynchronous message type but the actual implementation may involve both synchronous and asynchronous message type.



Schematic of Broker Architecture using Bridge Components

- The figure shows a schematics broker architecture that use bridge components to communicate between two remote processors.
- Each bridge converts services requests into a network specific protocol so that the message can be transmitted.



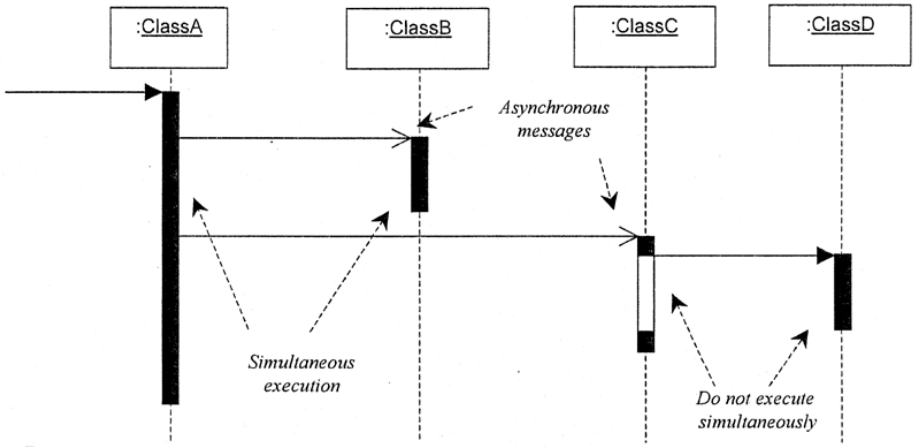
Concurrent

- User requirements may dictate concurrent behavior that cannot be supported using facilities such as multi-tasking but requires the use of custom-built scheduler component.
- For example concurrent behavior may be required because:
  - A use case indicates that different events require concurrent response.
  - A statechart highlights the need for concurrent substates.
  - An interaction diagram may show a single thread of control splitting into two concurrent threads.

Simulating Concurrency in the Execution of a System

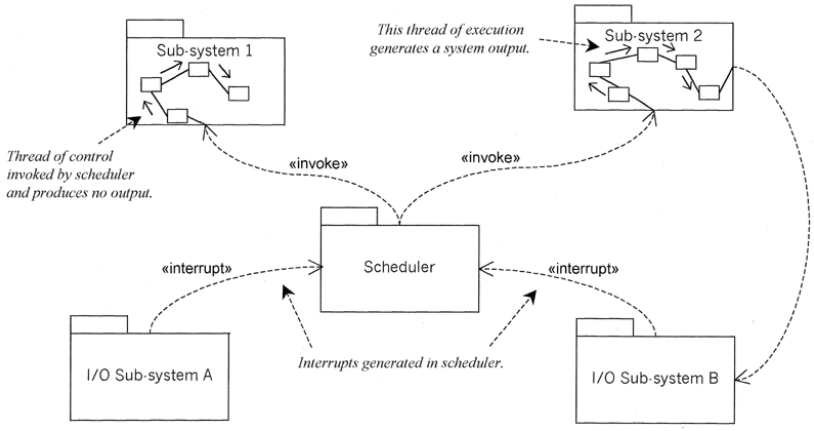
- Concurrency may be simulated by
  - Utilizing multi-tasking capabilities in the operating system.
  - Using a software development environment that supports multi-threading.
  - Using a scheduler to serialize concurrent threads.
  - Using a multi-processor environment.

Concurrent Activity in an Interaction Diagram



Scheduler Handling Concurrency

- The figure illustrates a possible relationship between a scheduler and other parts of a system



Processor Allocation

- Allocation of a system to multiple processors
  - Application should be divided into sub-systems
  - Estimate processing requirements for sub-systems
  - Determine access criteria and location requirements
  - Identify concurrency requirements
  - Each sub-system should be allocated to an operating platform
  - Communication requirements between sub-systems should be determined
  - The communications infrastructure should be specified

Data Management Issues

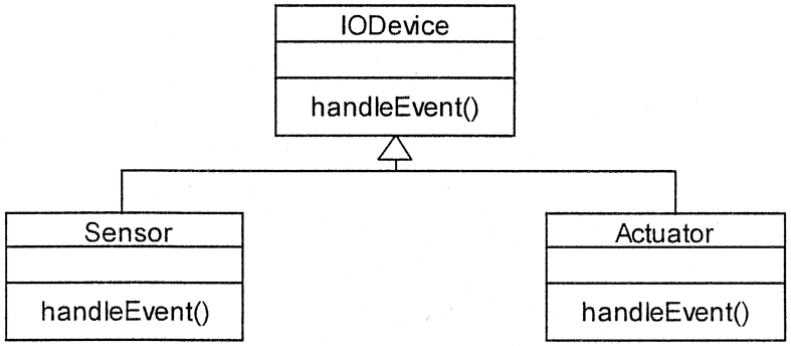
- A DBMS provides various facilities that are useful across a wide range of different applications. These include:
  - Different views of the data by different users
  - Control of multi-user access
  - Distribution of the data over different platforms
  - Security
  - Enforcement of integrity constraints
  - Access to data by various applications
  - Data recovery
  - Portability across platforms
  - Data access via query languages
  - Query optimization

Development Standards

- HCI Guidelines
- Input/Output Device Guidelines
- Construction Guidelines

I/O Device Hierarchy

- I/O Hierarchy providing consistency for device handling



Prioritizing Design Trade-offs

- Designer is often faced with design objectives that are mutually incompatible
- It is helpful if guidelines are prepared for prioritizing design objectives
- If design choice is unclear users should be consulted

Design for Implementation

- Initialization and implementation issues should be considered
- There may be data transfer or data conversion requirements or both