3. Concurrency
- Identify concurrent threads and address concurrency issues.
- Design goal: response time, performance.

- Threads
  - A thread of control is a path through a set of state diagrams on which a single object is active at a time.
  - A thread remains within a state diagram until an object sends an event to another object and waits for another event.
  - Thread splitting: Object does a nonblocking send of an event.

- Two objects are inherently concurrent if they can receive events at the same time without interacting.
- Inherently concurrent objects should be assigned to different threads of control.
- Objects with mutual exclusive activity should be folded into a single thread of control (Why?)

Implementing Concurrency
- Concurrent systems can be implemented on any system that provides
  - physical concurrency (hardware)
  - or
  - logical concurrency (software): Scheduling problem (Operating systems)

Classroom Activity – Concurrency
- Description: For your design identify the potential concurrency.
- Which objects of the object model are independent?
- What kinds of threads of control are identifiable?
- Does the system provide access to multiple users?
- Can a single request to the system be decomposed into multiple requests? Can these requests be handled in parallel?
- Process:
  - Meet as teams
  - Choose a scribe to record design goals
  - Use questions
  - You have about 5 minutes.

4. Hardware Software Mapping
- This activity addresses two questions:
  - How shall we realize the subsystems: Hardware or Software?
  - How is the object model mapped on the chosen hardware & software?
  - Mapping Objects onto Reality: Processor, Memory, Input/Output
  - Mapping Associations onto Reality: Connectivity
- Much of the difficulty of designing a system comes from meeting externally-imposed hardware and software constraints.
- Certain tasks have to be at specific locations
**Mapping the Objects**

- Processor issues:
  - Is the computation rate too demanding for a single processor?
  - Can we get a speedup by distributing tasks across several processors?
  - How many processors are required to maintain steady state load?
- Memory issues:
  - Is there enough memory to buffer bursts of requests?
- I/O issues:
  - Do you need an extra piece of hardware to handle the data generation rate?
  - Does the response time exceed the available communication bandwidth between subsystems or a task and a piece of hardware?

**Mapping the Subsystems Associations: Connectivity**

- Describe the physical connectivity of the hardware
  - Often the physical layer in ISO's OSI Reference Model
  - Which associations in the object model are mapped to physical connections?
  - Which of the client-supplier relationships in the analysis/design model correspond to physical connections?
- Describe the logical connectivity (subsystem associations)
  - Identify associations that do not directly map into physical connections:
    - How should these associations be implemented?

**Typical Informal Example of a Connectivity Drawing**

**Connectivity in Distributed Systems**

- If the architecture is distributed, we need to describe the network architecture (communication subsystem) as well.
- Questions to ask
  - What are the transmission media? (Ethernet, Wireless)
  - What is the Quality of Service (QOS)? What kind of communication protocols can be used?
  - Should the interaction asynchronous, synchronous or blocking?
  - What are the available bandwidth requirements between the subsystems?
    - Stock Price Change -> Broker
    - Icy Road Detector -> ABS System

**Drawing Hardware/Software Mappings in UML**

- System design must model static and dynamic structures:
  - Component Diagrams for static structures
    - show the structure at design time or compilation time
  - Deployment Diagram for dynamic structures
    - show the structure of the run-time system
- Note the lifetime of components
  - Some exist only at design time
  - Others exist only until compile time
  - Some exist at link or runtime

**Component Diagram**

- Component Diagram
  - A graph of components connected by dependency relationships.
  - Shows the dependencies among software components
  - source code, linkable libraries, executables
  - Dependencies are shown as dashed arrows from the client component to the supplier component.
  - The kinds of dependencies are implementation language specific.
  - A component diagram may also be used to show dependencies on a façade:
    - Use dashed arrow the corresponding UML interface.
Deployment Diagram

- Deployment diagrams are useful for showing a system design after the following decisions are made
  - Subsystem decomposition
  - Concurrency
  - Hardware/Software Mapping

- A deployment diagram is a graph of nodes connected by communication associations.
  - Nodes are shown as 3-D boxes.
  - Nodes may contain component instances.
  - Components may contain objects (indicating that the object is part of the component)

5. Data Management

- Some objects in the models need to be persistent
  - Provide clean separation points between subsystems with well-defined interfaces.
  - A persistent object can be realized with one of the following
    - Data structure
      - If the data can be volatile
    - Files
      - Cheap, simple, permanent storage
      - Low level (Read, Write)
      - Applications must add code to provide suitable level of abstraction
    - Database
      - Powerful, easy to port
      - Supports multiple writers and readers

File or Database?

- When should you choose a file?
  - Are the data voluminous (bit maps)?
  - Do you have lots of raw data (core dump, event trace)?
  - Do you need to keep the data only for a short time?
  - Is the information density low (archival files, history logs)?

- When should you choose a database?
  - Do the data require access at fine levels of details by multiple users?
  - Must the data be ported across multiple platforms (heterogeneous systems)?
  - Do multiple application programs access the data?
  - Does the data management require a lot of infrastructure?
Database Management System

- Contains mechanisms for describing data, managing persistent storage and for providing a backup mechanism.
- Provides concurrent access to the stored data.
- Contains information about the data (“meta-data”), also called data schema.

Issues To Consider When Selecting a Database

- **Storage space**
  - Database require about triple the storage space of actual data
- **Response time**
  - Mode databases are I/O or communication bound (distributed databases). Response time is also affected by CPU time, locking contention and delays from frequent screen displays
- **Locking modes**
  - Pessimistic locking: Lock before accessing object and release when object access is complete
  - Optimistic locking: Reads and writes may freely occur (high concurrency!) When activity has been completed, database checks if contention has occurred. If yes, all work has been lost.
- **Administration**
  - Large databases require specially trained support staff to set up security policies, manage the disk space, prepare backups, monitor performance, adjust tuning.

Object-Oriented Databases

- Support all fundamental object modeling concepts
  - Classes, Attributes, Methods, Associations, Inheritance
- Mapping an object model to an OO-database
  - Determine which objects are persistent.
  - Perform normal requirement analysis and object design
  - Create single attribute indices to reduce performance bottlenecks
  - Do the mapping (specific to commercially available product).
    - Example:
      - In ObjectStore, implement classes and associations by preparing C++ declarations for each class and each association in the object model

Relational Databases

- Based on relational algebra
- Data is presented as 2-dimensional tables. Tables have a specific number of columns and and arbitrary numbers of rows
  - Primary key: Combination of attributes that uniquely identify a row in a table. Each table should have only one primary key
  - Foreign key: Reference to a primary key in another table
- SQL is the standard language defining and manipulating tables.
- Leading commercial databases support constraints.
  - Referential integrity, for example, means that references to entries in other tables actually exist.

Classroom Activity – Data Management

- Description: Design the data management for your system.
  - Should the data be distributed?
  - Should the database be extensible?
  - How often is the database accessed?
  - What is the expected request (query) rate? In the worst case?
  - What is the size of typical and worst case requests?
  - Do the data need to be archived?
  - Does the system design try to hide the location of the databases (location transparency)?
  - Is there a need for a single interface to access the data?
  - What is the query format?
  - Should the database be relational or object-oriented?
- Process:
  - Meet as teams
  - Choose a scribe to record design goals
  - Use questions
  - You have about 10 minutes.

6. Global Resource Handling

- Discusses access control
- Describes access rights for different classes of actors
- Describes how object guard against unauthorized access
Defining Access Control

- In multi-user systems different actors have access to different functionality and data.
- During analysis we model these different accesses by associating different use cases with different actors.
- During system design we model these different accesses by examining the object model by determining which objects are shared among actors.
- Depending on the security requirements of the system, we also define how actors are authenticated to the system and how selected data in the system should be encrypted.

Access Matrix

- We model access on classes with an access matrix.
  - The rows of the matrix represent the actors of the system.
  - The columns represent classes whose access we want to control.
- Access Right: An entry in the access matrix. It lists the operations that can be executed on instances of the class by the actor.

Access Matrix Implementations

- Global access table: Represents explicitly every cell in the matrix as a (actor,class,operation) tuple.
  - Determining if an actor has access to a specific object requires looking up the corresponding tuple. If no such tuple is found, access is denied.
- Access control list associates a list of (actor,operation) pairs with each class to be accessed.
  - Every time an object is accessed, its access list is checked for the corresponding actor and operation.
  - Example: guest list for a party.
- A capability associates a (class,operation) pair with an actor.
  - A capability provides an actor to gain control access to an object of the class described in the capability.
  - Example: An invitation card for a party.
- Which is the right implementation?

Global Resource Questions

- Does the system need authentication?
  - If yes, what is the authentication scheme?
  - User name and password? Access control list
  - Tickets? Capability-based
- What is the user interface for authentication?
- Does the system need a network-wide name server?
  - How is a service known to the rest of the system?
  - At runtime? At compile time?
  - By port?
  - By name?

7. Decide on Software Control

Choose implicit control (non-procedural, declarative languages)
- Rule-based systems
- Logic programming

Choose explicit control (procedural languages): Centralized or decentralized

Centralized control: Procedure-driven or event-driven
- Procedure-driven control
  - Control resides within program code. Example: Main program calling procedures of subsystems.
  - Simple, easy to build, hard to maintain (high recompilation costs)
- Event-driven control
  - Control resides within a dispatcher calling functions via callbacks.
  - Very flexible, good for the design of graphical user interfaces, easy to extend

Event-Driven Control Example: MVC

- Model-View-Controller Paradigm (Adele Goldberg, Smalltalk 80)
Software Control (continued)

- Decentralized control
  - Control resides in several independent objects.
  - Possible speedup by mapping the objects on different processors, increased communication overhead.
  - Example: Message based system.

Centralized vs. Decentralized Designs

- Should you use a centralized or decentralized design?
  - Take the sequence diagrams and control objects from the analysis model
  - Check the participation of the control objects in the sequence diagrams
    - If sequence diagram looks more like a fork: Centralized design
    - The sequence diagram looks more like a stair: Decentralized design

- Centralized Design
  - One control object or subsystem ("spider") controls everything
    - Pro: Change in the control structure is very easy
    - Con: The single control object is a possible performance bottleneck

- Decentralized Design
  - Not a single object is in control, control is distributed. That means, there is more than one control object
    - Con: The responsibility is spread out
    - Pro: Fits nicely into object-oriented development

Classroom Activity – Control

- Description: Select the type of software control for your system and justify your selection.
  - Procedural
  - Event-driven
  - Threads
- Process:
  - Meet as teams
  - Choose a scribe to record design goals
  - You have about 10 minutes.

8. Boundary Conditions

- Most of the system design effort is concerned with steady-state behavior.
- However, the system design phase must also address the initiation and finalization of the system. This is addressed by a set of new uses cases called administration use cases
  - Initialization
    - Describes how the system is brought from an non initialized state to steady-state ("startup use cases").
  - Termination
    - Describes what resources are cleaned up and which systems are notified upon termination ("termination use cases").
  - Failure
    - Many possible causes: Bugs, errors, external problems (power supply).
    - Good system design foresees fatal failures ("failure use cases").

Example: Administrative Use cases for MyTrip

- Administration use cases for MyTrip (UML use case diagram).
  - An additional subsystems that was found during system design is the server. For this new subsystem we need to define use cases.
  - ManageServer includes all the functions necessary to start up and shutdown the server.
### Modeling Boundary Conditions

- Boundary conditions are best modeled as use cases with actors and objects.
- Actor: often the system administrator
- Interesting use cases:
  - Start up of a subsystem
  - Start up of the full system
  - Termination of a subsystem
  - Error in a subsystem or component
- Task:
  - Model the startup of the ARENA system as a set of use cases.

### Classroom Activity – Partitioning

- **Description:** Partition your system into subsystems using the ideas of coupling and cohesion.
  - Initialization
    - How does the system start up?
      - What data need to be accessed at startup time?
      - What services have to be registered?
    - How does the user interface do at startup time?
      - How does it present itself to the user?
  - Termination
    - Are single subsystems allowed to terminate?
    - Are other subsystems notified if a single subsystem terminates?
    - How are local updates communicated to the database?
  - **Failure**
    - How does the system behave when a node or communication link fails? Are there backup communication links?
    - How does the system recover from failure? Is this different from initialization?
- **Process:**
  - Meet as teams. Use questions. You have about 10 minutes.

### Summary

Activities of system design:
- Concurrency identification
- Hardware/Software mapping
- Persistent data management
- Global resource handling
- Software control selection
- Boundary conditions

Each of these activities revises the subsystem decomposition to address a specific issue. Once these activities are completed, the interface of the subsystems can be defined.