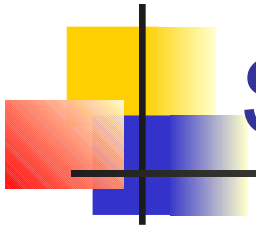
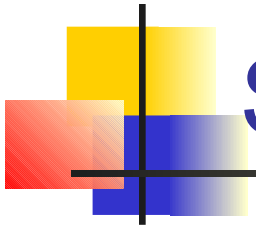
System Design



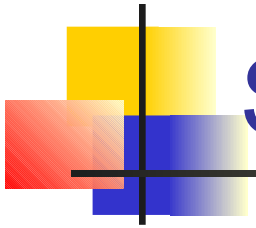
System Design

- System design is the high-level strategy for solving the problem and building a solution.
- It includes decisions about the organization of the system into subsystems, the allocation of subsystems to hardware and software components and major conceptual and policy decisions that form the framework for detailed design.



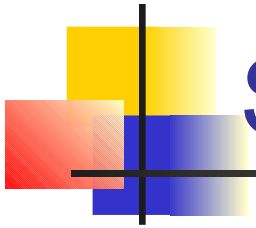
System Design

- During analysis, the focus is on what needs to be done, independent of how it is done. During design, decisions are made about how the problem will be solved, first at high level, then at increasingly detailed levels.
- It is the first design stage in which the basic approach to solving the problem is selected.
- During system design, the overall structure and style are decided.



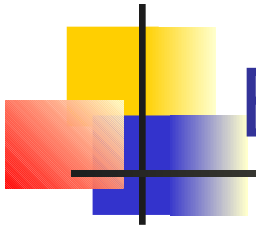
System Design

- The architecture provides the context in which more detailed decisions are made in later design stages.
- By making high-level decisions that apply to the entire system, the system designer partitions the problem into subsystems.



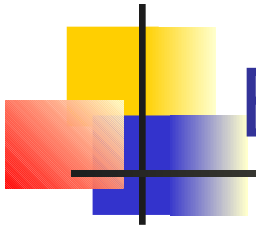
System Design

- The system designer must make the following decisions:
  - Organise the system into subsystems.
  - Identify concurrency inherent to the problem.
  - Allocate subsystems to processors and tasks.
  - Choose an approach for management of data.
  - Handle access to global resources.
  - Choose the implementation of control in s/w.
  - Handle boundary conditions.
  - Set trade-off priorities.



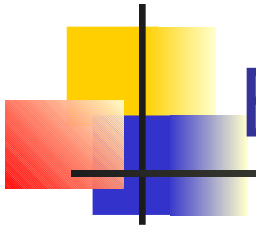
Breaking a System Into Subsystems

- Fist step in system design is to divide the system into a small number of components.
- Each subsystem encompasses aspects of the system that share some common property: similar functionality, the same physical location or execution on the same kind of hardware.



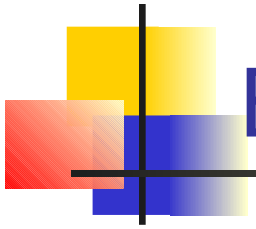
Breaking a System Into Subsystems

- A subsystem is not an object nor a function but a package of classes, associations, operations, events and constraints that are interrelated and that have a reasonably well-defined and small interface with other subsystems.
- A subsystem is usually identified by the service it provides.



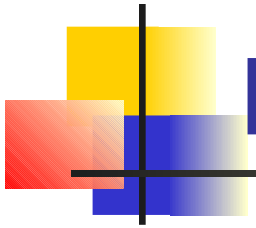
Breaking a System Into Subsystems

- A service is a group of related functions that share some common purpose, such as I/O processing, drawing pictures.
- Each subsystem has a well defined interface to the rest of the system. The interface specifies the form of all interactions and the information flow across subsystem boundaries but does not specify how the subsystem is implemented internally.



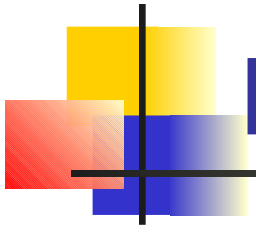
Breaking a System Into Subsystems

- Subsystems should be defined in such a manner that it has more cohesion and less coupling.
- The relationship between two subsystems can be client-supplier or peer-to-peer.
- The decomposition of systems into subsystems may be organised as a sequence of horizontal *layers* or vertical *partitions*.



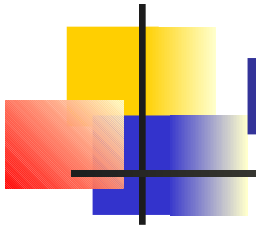
Layers

- A layered system is an ordered set of virtual worlds, each built in terms of the ones below it and providing the basis of implementation for the ones above it.
- The objects in each layer can be independent, having some interaction with other layers.
- A subsystem knows about the layers below it, but has no knowledge of the layers above it.



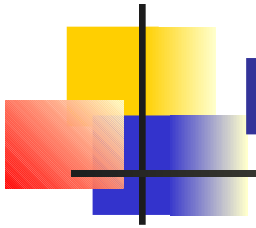
Layers

- Layered architecture come in two forms: closed and open.
- In closed architecture, each layer is built only in terms of the immediate lower layer. This reduces the dependencies between layers and allows changes to be made most easily because a layer's interface only affect the next layer.



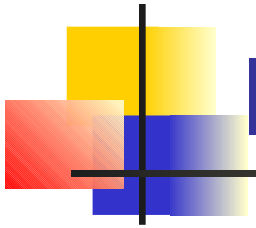
Layers

- In an open architecture, a layer can use features of any lower layer to any depth. This reduces the need to redefine operations at each level, which can result in a more efficient and compact code.
- Open architecture does not observe the principle of information hiding. Changes to a subsystem can affect any higher subsystem, so an open architecture is less robust than a closed architecture.



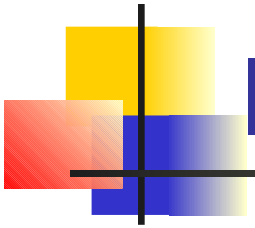
Partitions

- Partitions vertically divide a system into several independent or weakly-coupled subsystems, each providing one kind of service.
- The subsystems may have some knowledge of each other, but this knowledge is not deep, so major design dependencies are not created.



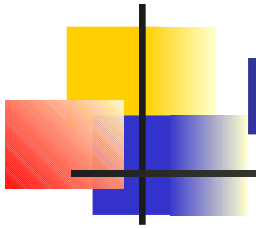
Identifying Concurrency

- In the analysis model, as in the real world and in hardware, all objects are concurrent.
- An important goal of system design is to identify which objects must be active concurrently and which objects have activity that is mutually exclusive.



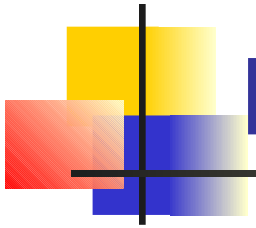
Identifying Inherent Concurrency

- The dynamic model is the guide to identifying concurrency.
- Two objects are inherently concurrent if they can receive events at the same time without interacting.
- If the events are unsynchronized, the objects cannot be folded onto a single thread of control.
- Two subsystems that are inherently concurrent need not necessarily be implemented as separate hardware units.



Defining Concurrent Tasks

- Although all objects are conceptually concurrent, in practice many objects in a system are interdependent.
- By examining the state diagrams of individual objects and the exchange of events among them, many objects can often be folded together onto a single thread of control.

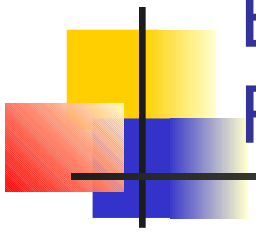


Defining Concurrent Tasks

- A thread of control is a path through a set of state diagrams on which only a single object at a time is active.
- A thread remains within a state diagram until an object sends an event to another object and waits for another event.
- The thread passes to the receiver of the event until it eventually returns to the original object.
- The thread split if the objects sends an event and continues executing.

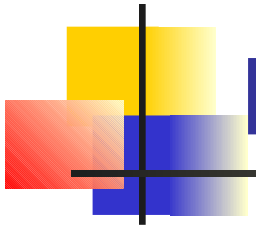
Allocating Subsystems to Processors and Tasks

- Each concurrent subsystem must be allocated to a hardware unit. The system designer must:
  - Estimate performance needs and the resources needed to satisfy them.
  - Choose hardware and software implementation for subsystem.
  - Allocate software subsystems to processors to satisfy performance needs and minimize interprocessor communication.
  - Determine the connectivity of the physical units that implement the subsystem.



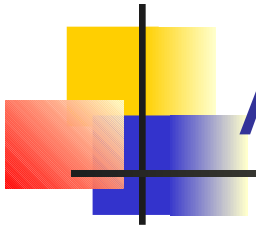
Estimating Hardware Resource Requirements

- The decision to use multiple processors or hardware functional units is based on a need for higher performance than a single CPU can provide.
- The system designer must estimate the required CPU processing power by computing the steady state load as the product of the number of transactions per second and the time required to process a transaction.



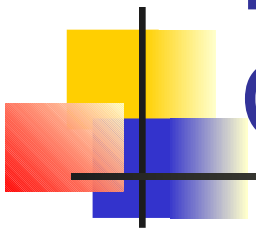
Hardware-Software Trade-offs

- Hardware can be regarded as a rigid but highly optimized form of software.
- The system designer must decide which subsystem will be implemented in hardware and which in software.
- Subsystems are implemented in hardware for two main reasons:
  - **Existing hardware provides exactly the functionality required**
  - **Higher performance is required than a general purpose CPU can provide, and more efficient hardware is available.**



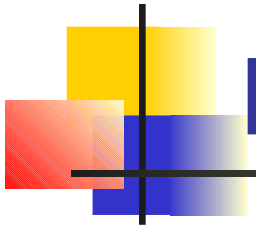
Allocating Tasks to Processors

- Tasks are assigned to processors because:
  - Certain tasks are required at specific physical location, to control hardware or to permit independent or concurrent operation.
  - The response time or information flow rate exceeds the available communication band width between a task and a piece of paper.
  - Computations rates are too great for a single processor, so tasks must be spread among several processors.



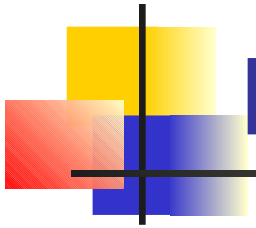
Determining Physical Connectivity

- The system designer must choose the arrangement and form of connections among the physical units.
- The following decisions must be made:
  - Choose the topology of connecting the physical units.
  - Choose the topology of repeated units.
  - Choose the form of the connection channels ad the communication protocols.



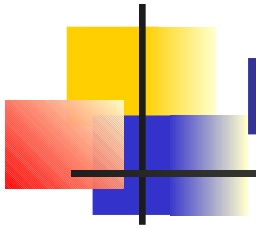
Management of Data Stores

- The following guidelines characterize the kind of data that belongs in a formal database:
  - Data that requires access at fine levels of detail by multiple users.
  - Data that can be efficiently managed with DBMS commands.
  - Data that must port across many hardware and operating system platforms.
  - Data that must be accessible by more than one application program.



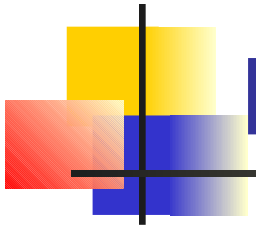
Management of Data Stores

- The following guidelines characterize the kind of data that belongs in a file and not in a relational database:
  - Data that is voluminous in quantity but difficult to structure within the confines of DBMS.
  - Data that is voluminous in quantity and of low information density.
  - “Raw” data that is summarized in the database.
  - Volatile data that is kept a short time and then discarded.



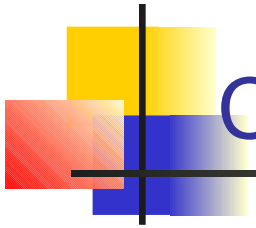
Handling Global Resources

- The system designer must identify global resources and determine mechanisms for controlling access to them. Global resources include: physical units, such as processors, communication satellites; Disk space and databases.
- If a resource is a physical object, then it can control itself by establishing a protocol for obtaining access within a concurrent system.
- If a resource is a logical entity, such as database, then there is danger of conflicting access in a shared environment.



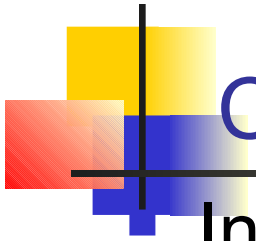
Handling Global Resources

- Each global resource must be owned by a “guardian object” that control access to it.
- All access to the resource must pass through the guardian object.
- In a time critical application, the cost of passing all access to a resource through a guardian object is sometimes too high, and clients must access the resource directly by making use of locks.
- A lock is a logical object associated with some defined subset of a resource that gives the lock holder the right to access the resource directly.



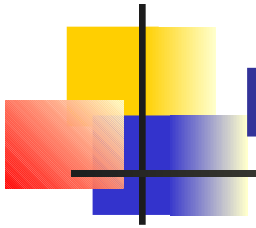
Choosing Software Control Implementation

- There are two kinds of control flows in a software system: external control and internal control.
- External control is the flow of externally visible events among the objects in the system.
- There are three kinds of control for external events: procedure-driven, event driven sequential and concurrent.



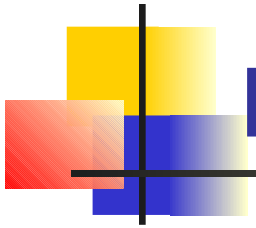
Choosing Software Control Implementation

- Internal control is the flow of control within a process. It exist only in the implementation and therefore is not inherently concurrent or sequential.
- Unlike external events, internal transfers of control, such as procedure calls or inter-task call are under the direction of the program and can be structured for convenience.
- Three kinds of control flow are common: procedure call, quasi-concurrent inter-tasks call and concurrent inter-tasks call.



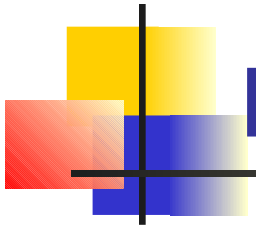
External Control: Procedure-driven Systems

- In a procedure-driven sequential system, control resides within the program code.
- The major advantage of procedure-driven control is that it is easy to implement with convention languages, the disadvantage is that it requires the concurrency inherent in objects to be mapped into a sequential flow of control.



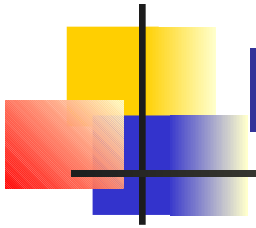
External Control: Event-driven Systems

- In an event-driven sequential system, control resides within a dispatcher or monitor provided by the language, subsystem or operating system.
- Event driven systems permit more flexible patterns of control than procedure-driven systems.
- The mapping from events to program construct is much simpler and more powerful.



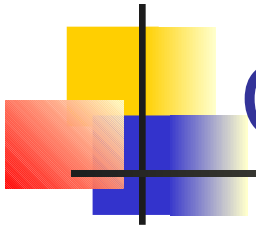
External Control: Concurrent Systems

- In a concurrent system, control resides concurrent in several independent objects, each a separate task. Events are implemented directly as one way message between objects.
- One task can wait for input, but other tasks continue execution.
- The OS usually supplies a queuing mechanism for events so that events are not lost if a task is executing when they arrive.



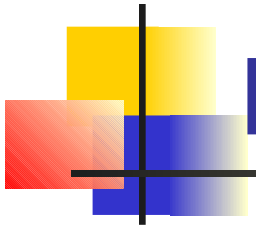
Internal Control

- External interactions inherently involve waiting for events because different objects are independent and cannot force other objects to respond: internal operations are generated by objects as part of the implementation algorithm, so their response patterns are predictable.
- Most internal operations can therefore be thought of as procedure calls, in which the caller issues a request and waits for the response.



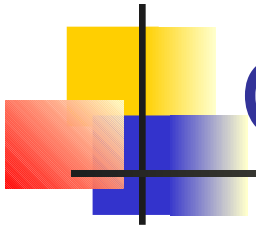
Common Architectural Frameworks

- There are several prototypical architectural frameworks that are common in existing systems. Each of these is well-suited to a certain kind of system. The kind of system include:
  - Batch processing.
  - Continuous transformation.
  - Interactive interface.
  - Dynamic simulation.
  - Real-time system.
  - Transaction manager.



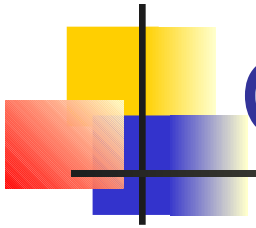
Batch Transformation

- A batch transformation is a sequential input-to-output transformation, in which inputs are supplied at the start and the goal is to compute an answer.
- The steps in designing a batch processing transformation are:
  - Break the overall transformation into stages, each stage performing one part of the transformation.
  - Define intermediate object classes for the data flows between each pair of successive stages.
  - Expand each stage in turn until the operations are straightforward to implement.
  - Restructure the final pipeline for optimization.



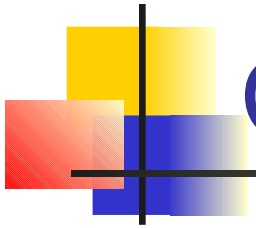
Continuous Transformation

- A continuous transformation is a system in which the outputs actively depend on changing inputs and must be periodically updated.
- since a complete re-computation is impossible for every input value change, an architecture for a continuous transformation must facilitate incremental computation. The transformation can be implemented as a pipeline of functions.
- The effect of each incremental change in an input value must be propagated through pipeline.



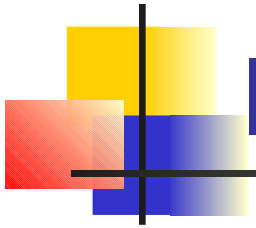
Continuous Transformation

- Synchronization of values within the pipeline may be important for high-performance systems.
- In such cases, operations are performed at well-defined times and the flow path of operations must be carefully balanced so that values arrive at the right place at the right time.



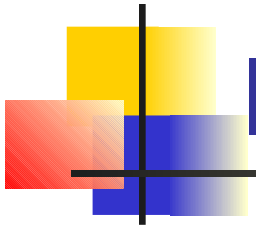
Continuous Transformation

- The steps in designing a pipeline for a continuous transformation are:
  - Draw a data flow diagram for the system
  - Define intermediate objects between each pair of successive stages, as in the batch transformation.
  - Differentiate each operation to obtain incremental changes to each stage.
  - Add additional intermediate objects for optimization.



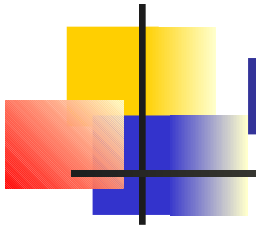
Interactive Interface

- An interactive interface is a system that is dominated by interactions between the system and external agents, such as humans, devices or other programs.
- It usually includes part of an entire application which can often be handled independently from the computational part of the application.
- The major concerns of an interactive interface are the communications protocol between the system and the external agents, the syntax of possible interactions, the presentation of output.



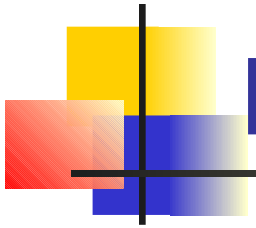
Interactive Interface

- Interactive interfaces are dominated by the dynamic model. The steps in designing an interactive interface are:
  - Isolate the objects that form the interface from the objects that define the semantics of the application.
  - Use predefined objects to interact with external agents.
  - Use the dynamic model as the structure of the program. Interactive interfaces are best implemented using concurrent control or event-driven control.
  - Isolate physical events from logical events.
  - Fully specify the application functions that are invoked by the interface. Make sure that the information to implement them is present.



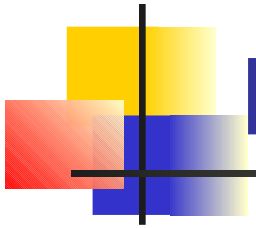
Dynamic Simulation

- A dynamic simulation models or tracks objects in the real world.
- Traditional methodologies built on data flow diagrams are poor at representing these problems because simulations involve many distinct objects that constantly update themselves, rather than a single large transformation.



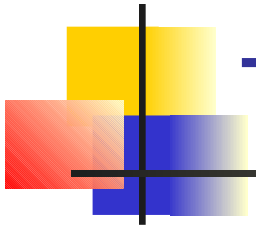
Dynamic Simulation

- The steps in designing a dynamic simulation are:
  - Identify actors, active real-world objects, from the object model
  - Identify discrete events.
  - Identify continuous dependencies.
  - Generally a simulation is driven by a timing loop at a fine time scale.
- The hardest problem with simulation is usually providing adequate performance.



Real time System

- A real time system is an interactive system for which time constraints on actions are particularly tight or in which the slightest timing failure cannot be tolerated.
- For critical actions, the system must be guaranteed to respond within an absolute interval of time.
- Typical application include process control, data acquisition, communication devices etc.



Transaction Manager

- A transaction manager is a database system whose main function is to store and access information. The information comes from the application domain.
- The steps in designing a transaction manager are
  - Map the object model directly into a database.
  - Determine the units of concurrency.
  - Determine the unit of transaction.
  - Design concurrency control for transactions.