COSC 310: Software Engineering

Dr. Bowen Hui
University of British Columbia Okanagan
bowen.hui@ubc.ca

What is system design?

- what does it mean to develop a "design" of the system?
- why bother designing it?
- how to document a design?
- why bother with documentation?

System Modeling

- develop abstract models of system
- each model represent a different view
- typically done via graphical notation
  - e.g., DFD, UML
- some can also be formally modeled using mathematics

- most important aspect: leave out details

Different System Views

- context models (DFD, UML)
  - external perspective - model context/environment, set system boundary
- interaction models (DFD, UML)
  - model interactions between system and its environment, or between system components
- structural models (UML class diagrams, ERD)
  - model system organization, or structure of data that is processed by system
- behavioural models (example?)
  - model dynamic behaviour of system and how system responds to events

Architectural Design

- provides description of how system is organized
- influences system properties such as performance and security
- includes decisions on:
  - types of application
  - distribution of system
  - architectural styles used
  - ways to document and evaluate system

Architectural Patterns

- abstract description of good practice that have been tried and tested

- reuse knowledge about generic system architectures
  - explain when specific architecture is used
  - compare advantages and disadvantages

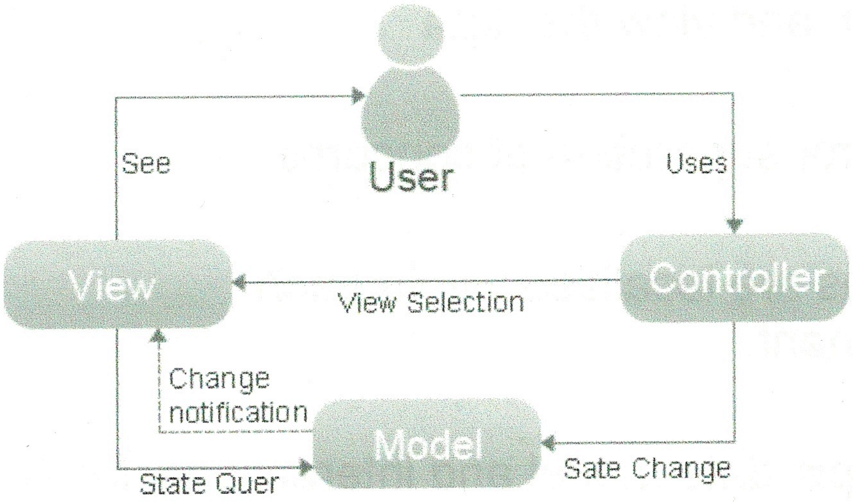
Common Architectural Patterns

- model-view-controller
- layered architecture
- repository
- client-server
- pipe and filter

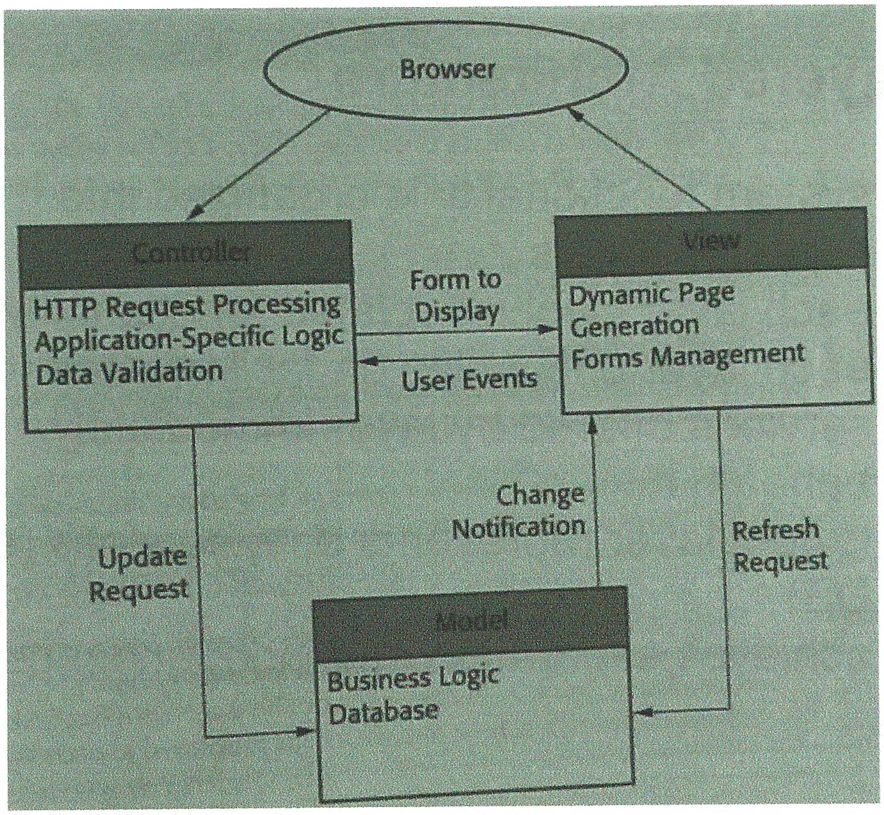
Model View Controller

- separates presentation and interaction from system data
- structured into 3 logical components:
  - **Model** - manages data and their operations
  - **View** - defines and manages how data is presented to the user
  - **Controller** - manages user interaction, passes them to the View / Model

Visual View of MVC



Example: Website using MVC



MVC Pros and Cons

- used when there are multiple ways to interact and view the data

- allows for separation of concerns

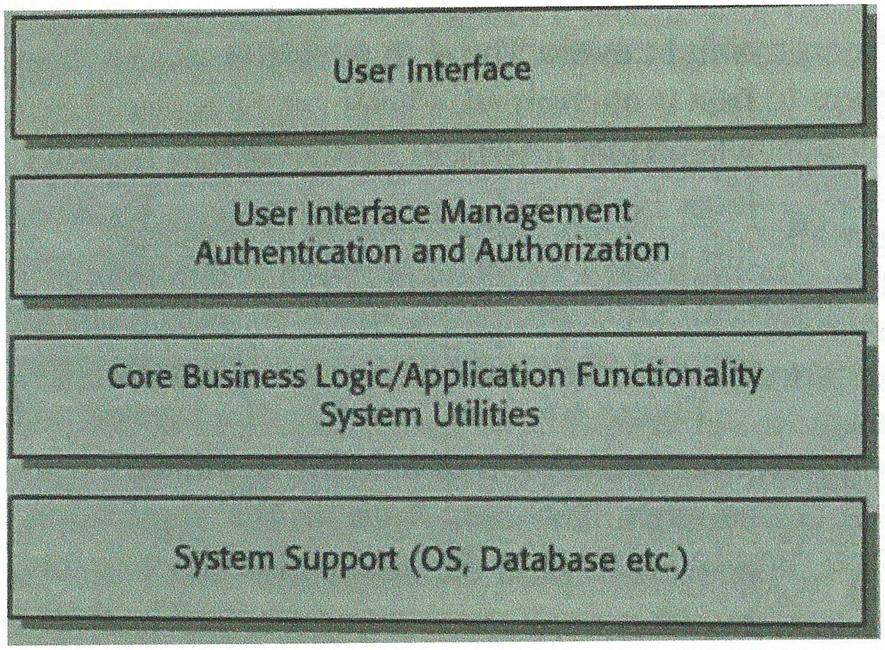
- changes are localized within each component

- for simple data model and interactions, MVC can involve additional complexity

Layered Architecture

- system functionality is organized into separate layers
- each layer involves related functionality
- each layer only relies on facilities and services offered by layer immediately beneath it
- lowest layer represents core services (e.g., DB)
- achieves separation and independence

Visual View of Layered Architecture



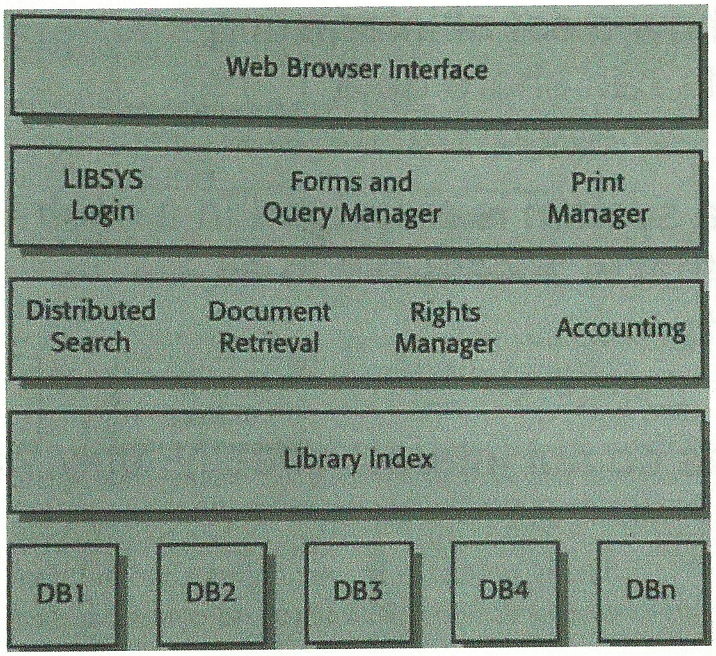
User Interface

User Interface Management
Authentication and Authorization

Core Business Logic/Application Functionality
System Utilities

System Support (OS, Database etc.)

Example:
System for sharing copyright documents held in different libraries



Web Browser Interface

LIBSYS Login | Forms and Query Manager | Print Manager

Distributed Search | Document Retrieval | Rights Manager | Accounting

Library Index

DB1 | DB2 | DB3 | DB4 | DBn

Layered Architecture: Pros and Cons

- supports incremental development of systems
- allows layers to be completely replaced
- redundant facilities can be provided to increase dependability
- clean separation of layers is often difficult in practice
  - upper layers may have to interact directly with lower layers
- performance may be slower due to layered requests
- used when:
  - building new facilities on top of existing systems
  - multiple teams working on separate layers
  - there's a requirement for multi-level security

Repository Architecture

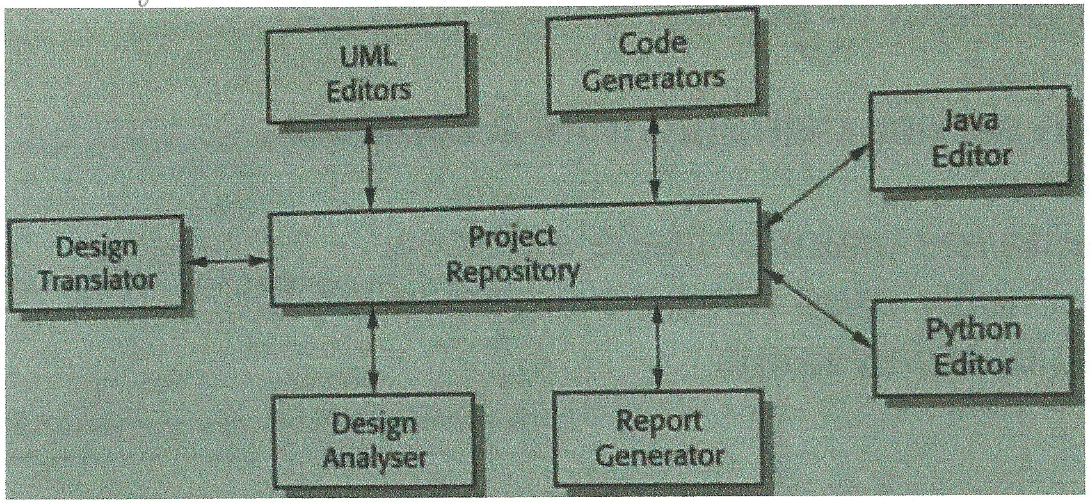
- describes how a set of interacting components can share data

- all data in a system is managed in a central repository that is accessible to all components

- components do not interact directly, only via repository

Example:
IDE using a Repository pattern

↑meaning?



Repository Architecture:
Pros and Cons

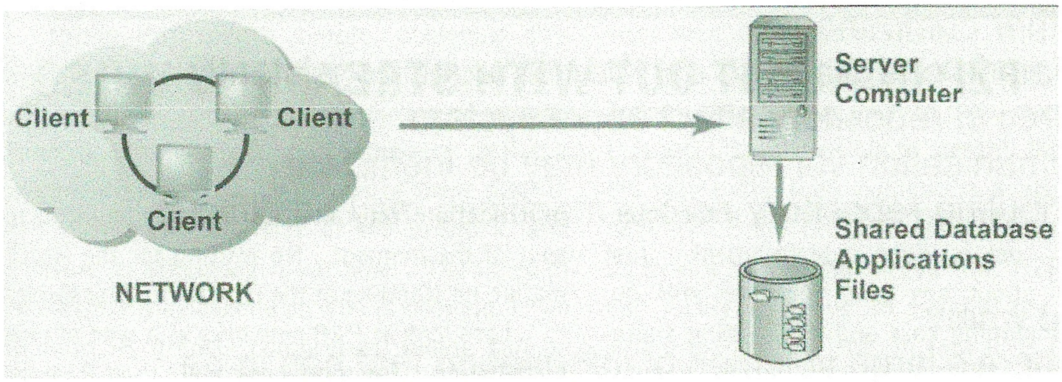
- components can be designed/implemented independent of other components
- all data in one place so can be managed consistently
- failures in repository affect entire system
- communication via repository may be inefficient
- distributing repository across machines may be difficult

- used when:
  - there's large volumes of information that has to be stored for a long time
  - data that needs to be shared

Client-Server Architecture

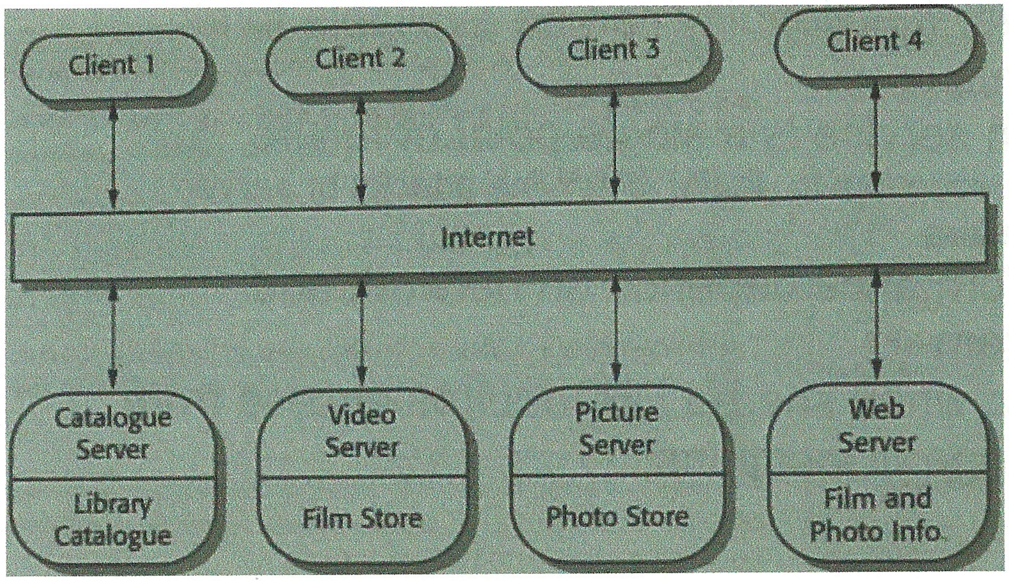
- functionality of system is organized into services
- each service delivered from a separate server
- clients are users of these services
- commonly used run-time organization for distributed systems

Visual View of Client-Server Architecture



- clients are connected in a network together with one/more servers

Example:
Film Library using Client-Server pattern



Client-Server Details

- strictly speaking, clients and servers are *software*
  - both may exist on the same physical machine
- client's tasks:
  - sends requests to servers
  - display results from server to user
- server's responsibilities
  - satisfies requests
  - consult other sources
  - fail to satisfy requests
- note: servers cannot initiate dialog with clients

Client-Server Pros and Cons

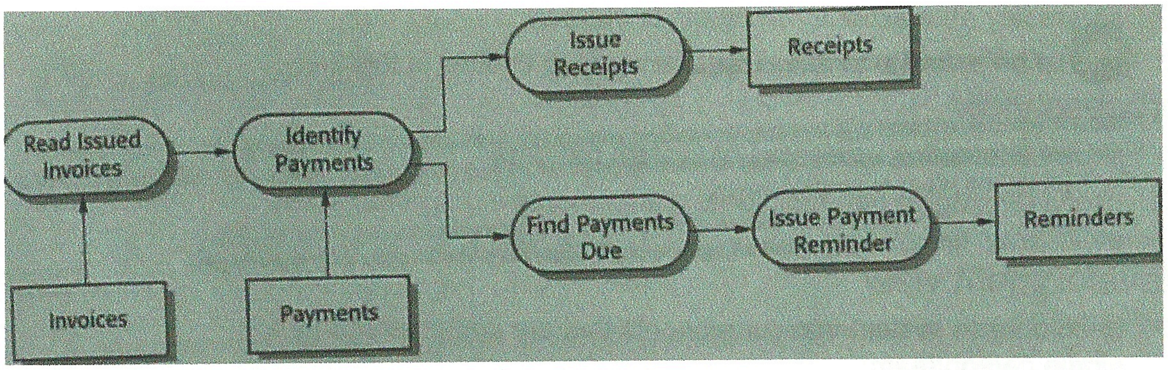
- servers can be distributed across a network
- server functionality can be available to all clients
- each service is a single point of failure
  - susceptible to denial of service attacks or server failure
- performance depends on network and system
- used when shared DB requires access from a range of locations

Pipe and Filter Architecture

what is the meaning of this?

- models run-time organization of system
- each component is a transformation of data
- input data flows through these transforms until converted to output
- transformations may execute sequentially or in parallel
- data can be single item or in batch

Example:
Invoice processing



Pipe and Filter Pros and Cons

- easy to understand
- supports transformation reuse
- workflow style matches structure of many business processes
- evolution by transformations is straightforward
- format of data transfer needs standardization
- each transformation needs to parse and unparse data format
- used in data processing applications

Views for Documenting Design

- logical view
  - shows key abstractions as objects or classes
  - should be possible to relate requirements to entities in this view
- process view
  - shows interacting processes at system run-time
  - useful for judging non-functional requirements e.g., performance, availability
- development view
  - decomposes system into components that are implemented by a single person/team
  - useful for managers and programmers
- physical view
  - shows hardware and distribution of software across processors
  - useful for planning and deployment

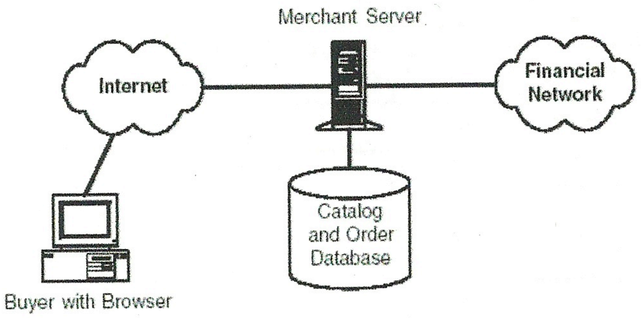
Physical vs. Logical Views

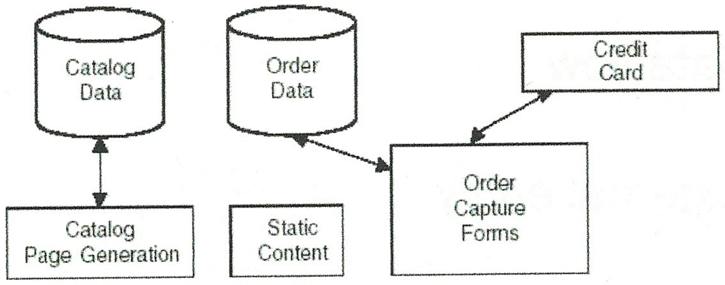
- purpose is to show high level system specification

- **physical design** = mapping of logical design to physical components
  - e.g. actual servers and communications

- **logical design** = description of flow of information and major processes and relationships involved
  - major system components and their inputs/outputs

Physical vs. Logical Views



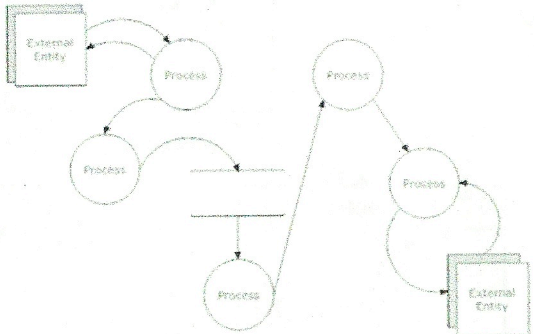


hard to understand...

Need formal syntax

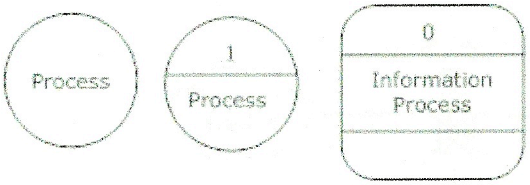
Data Flow Diagram

- models the flow of data in a system
- identifies system processes, their inputs and outputs
- focus on system functions
- cannot represent system objects
- example:

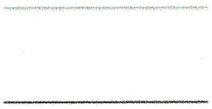


Notation

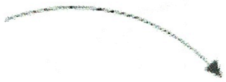
- process



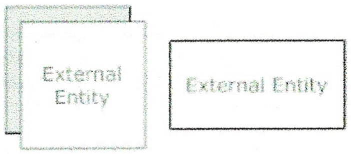
- datastore/file



- data flow

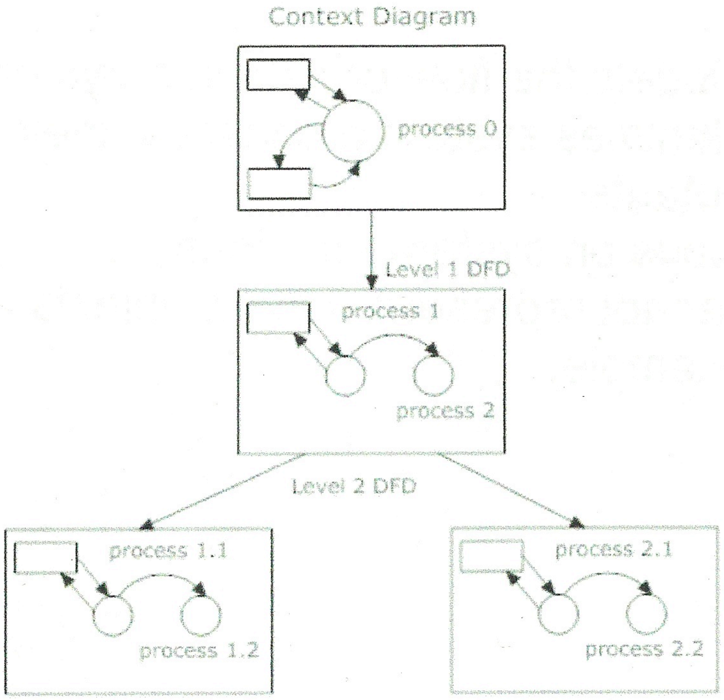


- external entity



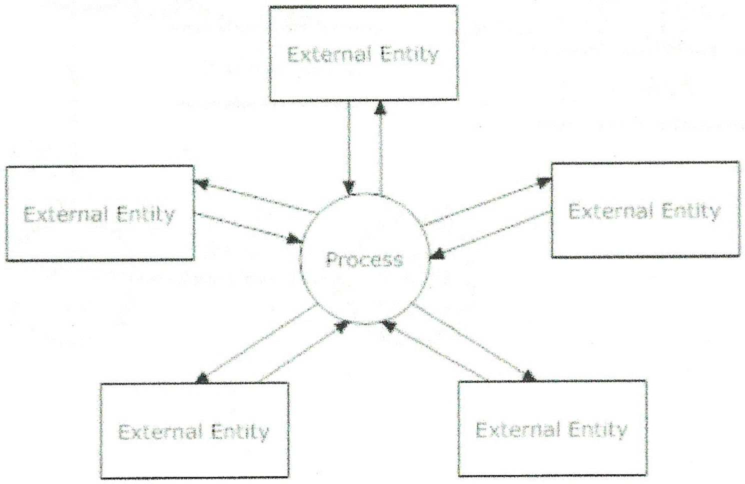
DFD Layers

- each layer expands a process from the previous layer



Level 0: Context Diagram

- contains one process only
- describes system's external interactions



NOTATIONS
are
reversed?

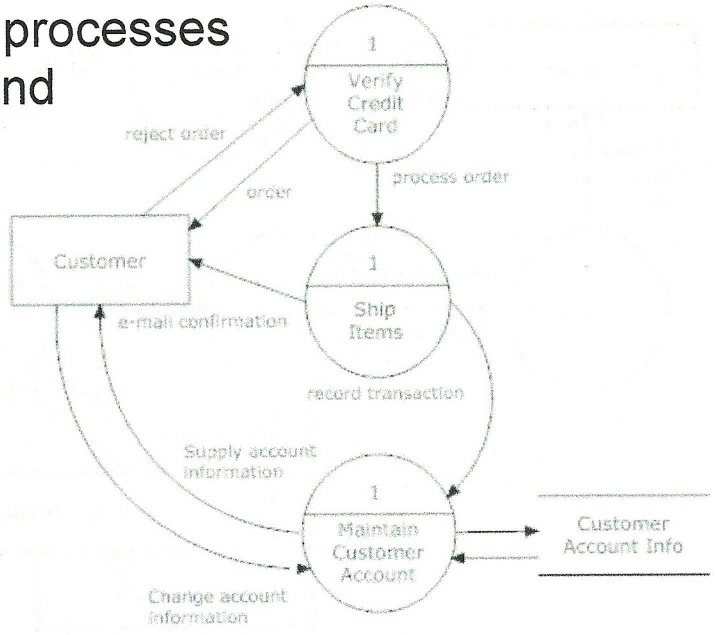
Ellipses
should be
external
entities
& processes
are rectangular

WHICH ONE
TO USE?
STICK TO SSADM, eh?

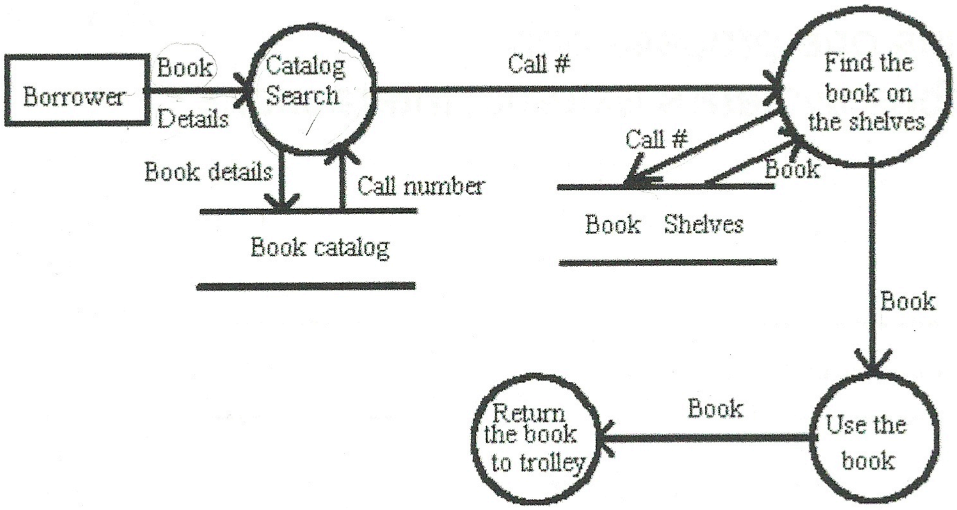
DFD Level 1

- illustrates main processes
- levels can expand

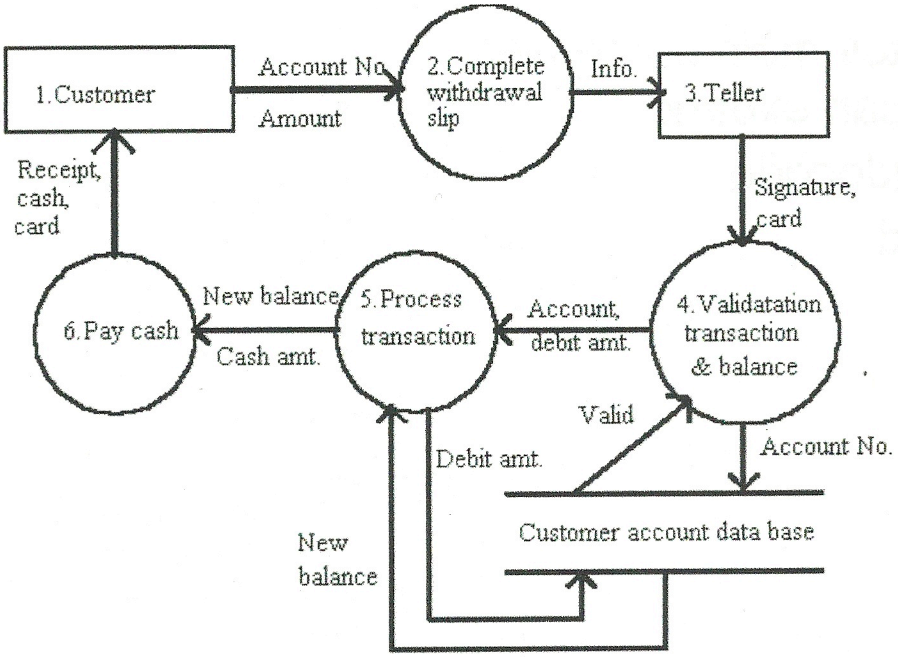
until pseudocode
is reached



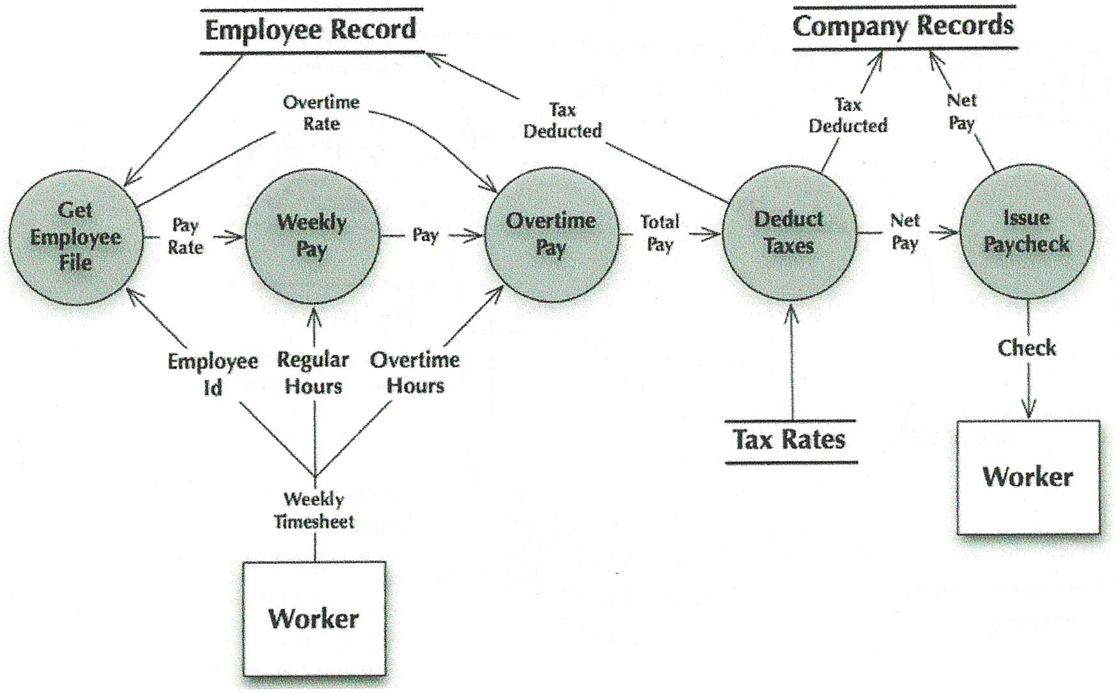
Example:
Library loans



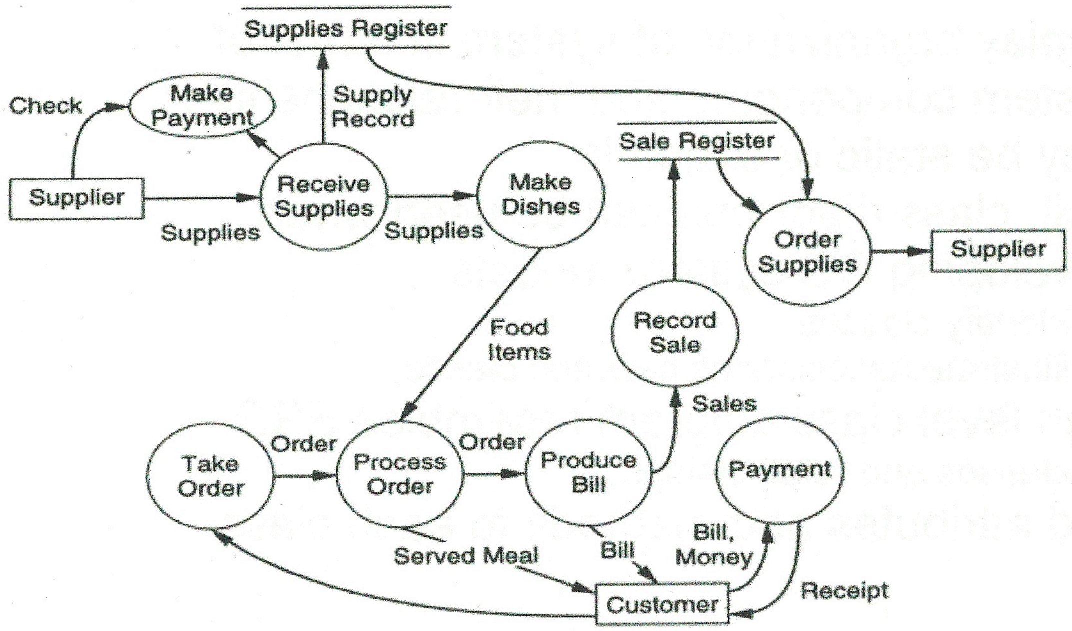
Example:
Cash Withdrawal from Teller



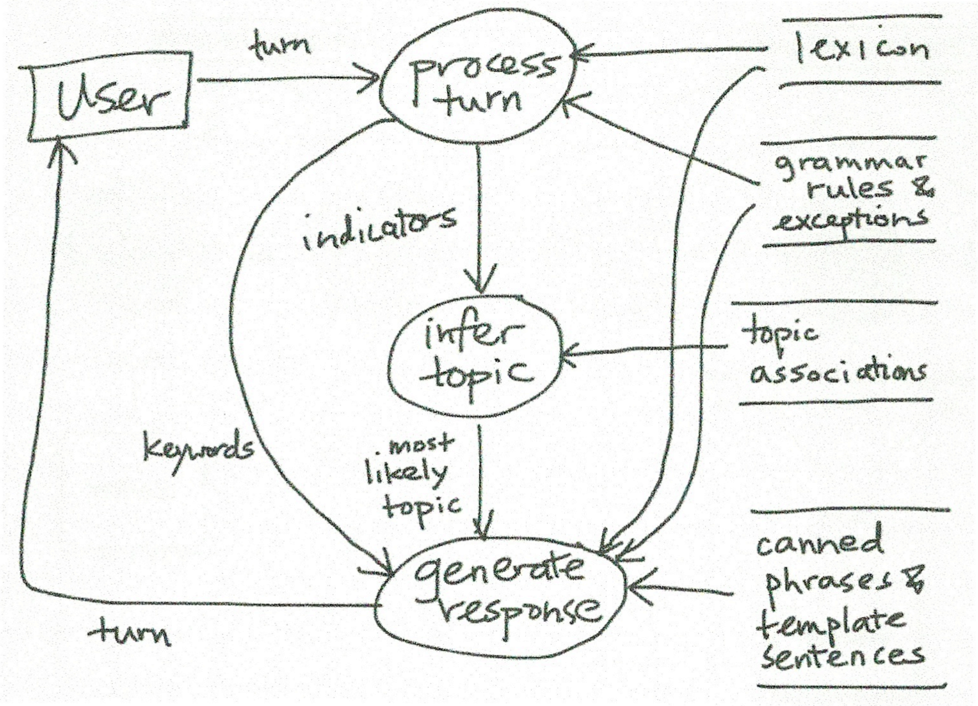
Example:
Employee Payroll



Example:
Inventory Management



Example:
Project - AI option

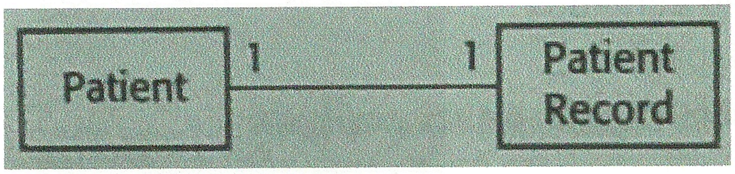


Structural Models (Sommerville Ch5)

- display organization of system in terms of system components and their relationships
- may be static or dynamic
- UML class diagrams can be used when developing OO system models
  - identify classes
  - illustrate associations between classes
- high level class diagram resembles ERD
  - classes and relationships
- add attributes and methods to each class

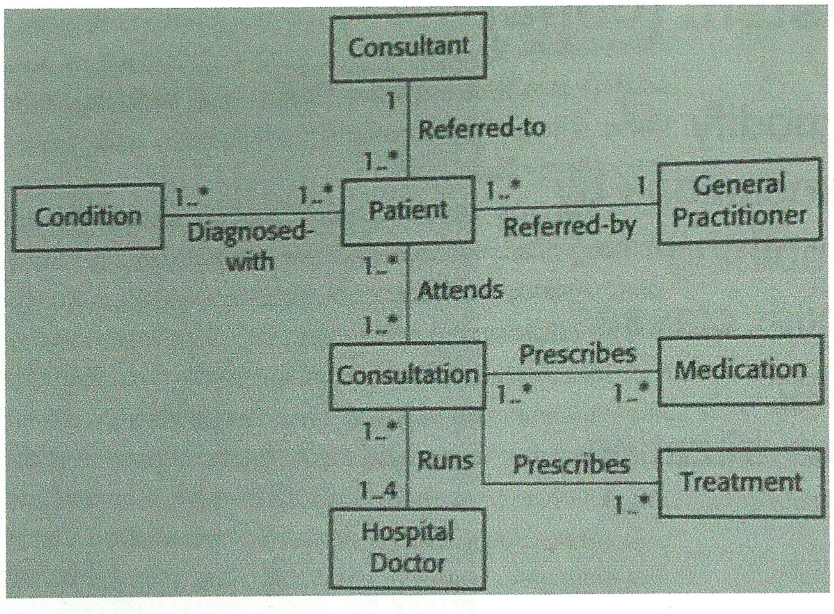
Example:
Patient Medical System

- start off with basic classes and their association



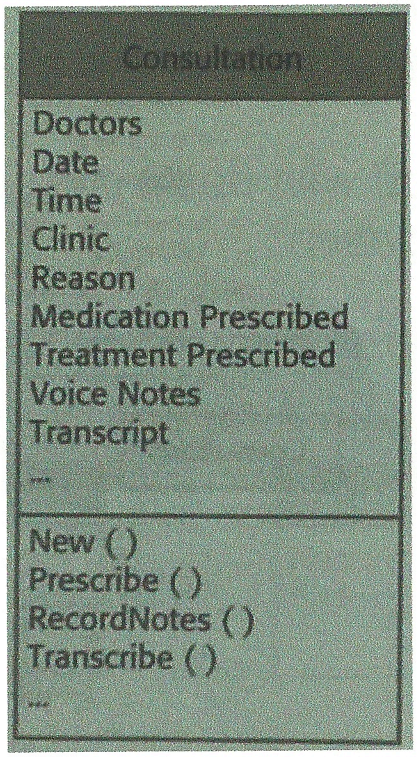
Example:
Patient Medical System (cont.)

- expand to include other classes and their associations



Example: Patient Medical System (cont.)

- for each class:
  - add detailed attributes (and their types)
  - add detailed methods



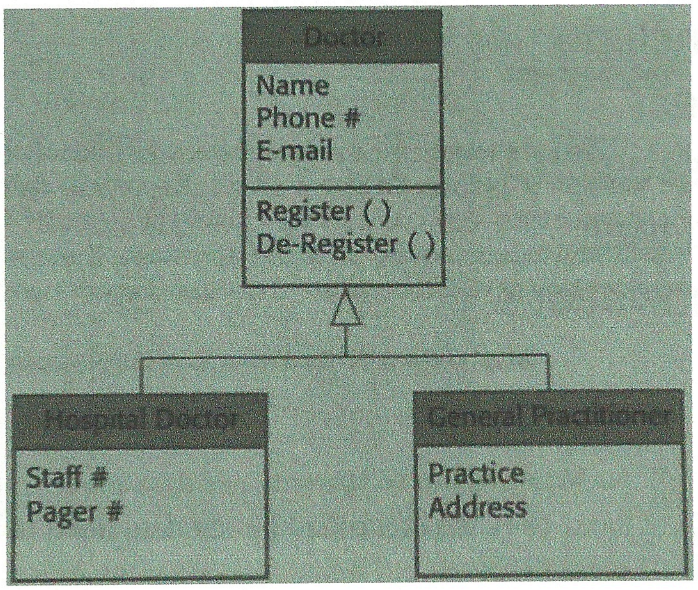
Consultation

Doctors
Date
Time
Clinic
Reason
Medication Prescribed
Treatment Prescribed
Voice Notes
Transcript
...

New ( )
Prescribe ( )
RecordNotes ( )
Transcribe ( )
...

Example: Patient Medical System (cont.)

- modify associations by combining similar classes using generalization



Doctor

Name
Phone #
E-mail

Register ( )
De-Register ( )

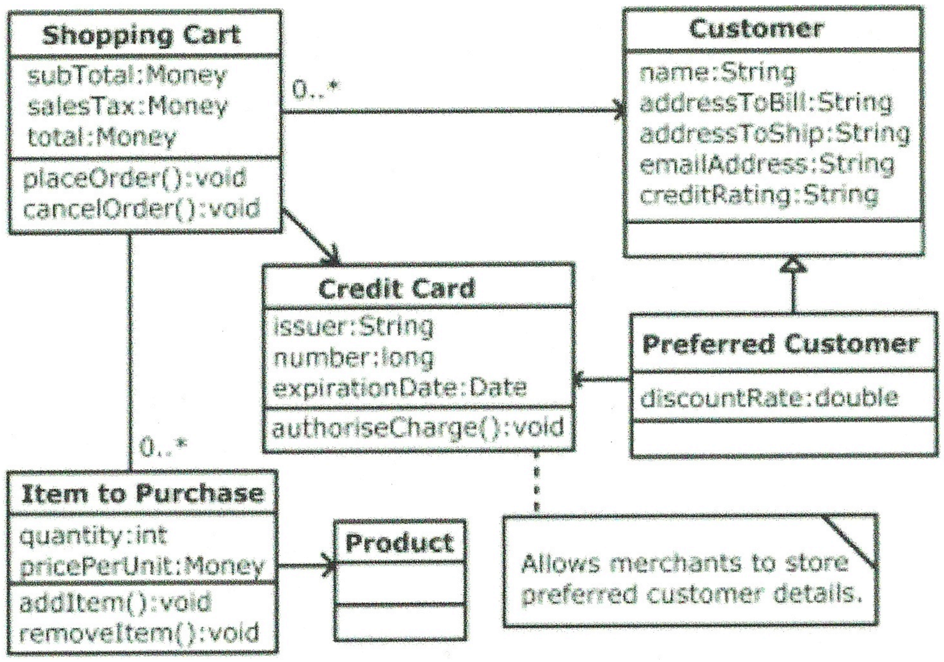
Hospital Doctor

Staff #
Pager #

General Practitioner

Practice
Address

Example:
Online Payment System



References

- http://www.smartdraw.com/resources/tutorials/data-flow-diagrams/#/resources/tutorials/Data-Flow-Diagram-Notations
- http://www.schools.ash.org.au/olshc/infotech/dataflow.htm
- Somerville's text Ch 5 and Ch 6
- http://www.alasdairking.me.uk/research/King2004-PresentingUMLDiagrams.htm