Software Engineering

Organization of this Lecture:

- What is Software Engineering?
- Programs vs. Software Products
- Evolution of Software Engineering
- Notable Changes In Software Development Practices
- Introduction to Life Cycle Models
- Summary

What is Software Engineering?

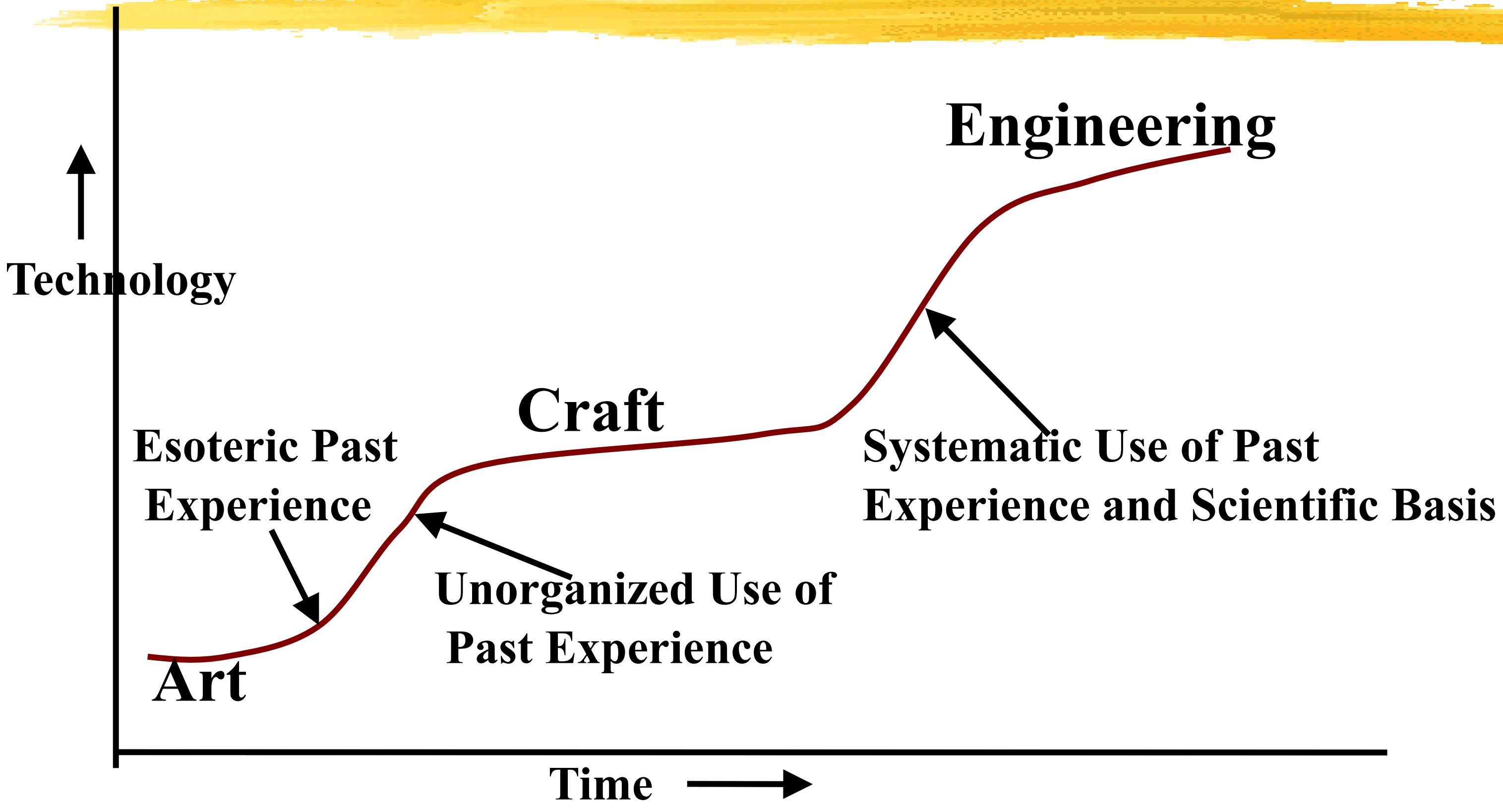
- Engineering approach to develop software.
  - Building Construction Analogy.

- Systematic collection of past experience:
  - techniques,
  - methodologies,
  - guidelines.

Engineering Practice

- Heavy use of past experience:
  - Past experience is systematically arranged.
- Theoretical basis and quantitative techniques provided.
- Many are just thumb rules.
- Tradeoff between alternatives
- Pragmatic approach to cost-effectiveness

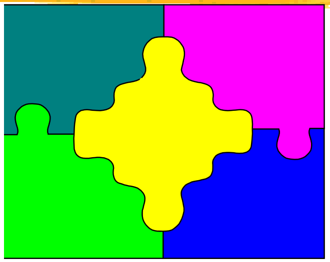
Technology Development Pattern



Why Study Software Engineering? (1)

- To acquire skills to develop large programs.
  - Exponential growth in complexity and difficulty level with size.
  - The ad hoc approach breaks down when size of software increases

Why Study Software Engineering? (2)



- Ability to solve complex programming problems:
  - How to break large projects into smaller and manageable parts?
- Learn techniques of:
  - specification, design, interface development, testing, project management, etc.

Why Study Software Engineering? (3)

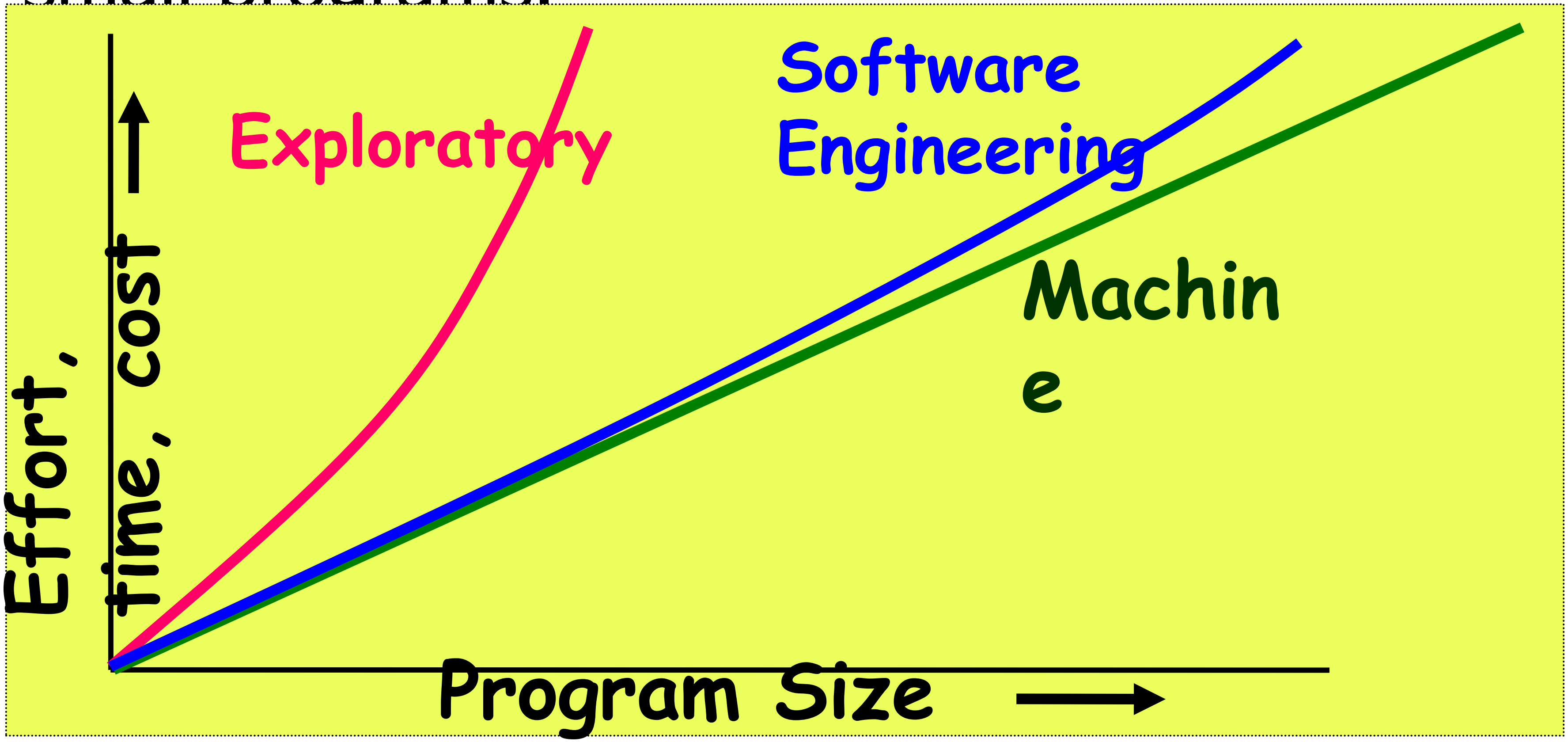
- To acquire skills to be a better programmer:
  - Higher Productivity
  - Better Quality Programs

Exploratory Development

- Early programmers used an exploratory (also called build and fix) style.
  - A `dirty' program is quickly developed.
  - The bugs are fixed as and when they are noticed.
  - Similar to a first year student develops programs.

What is Wrong with the Exploratory Style?

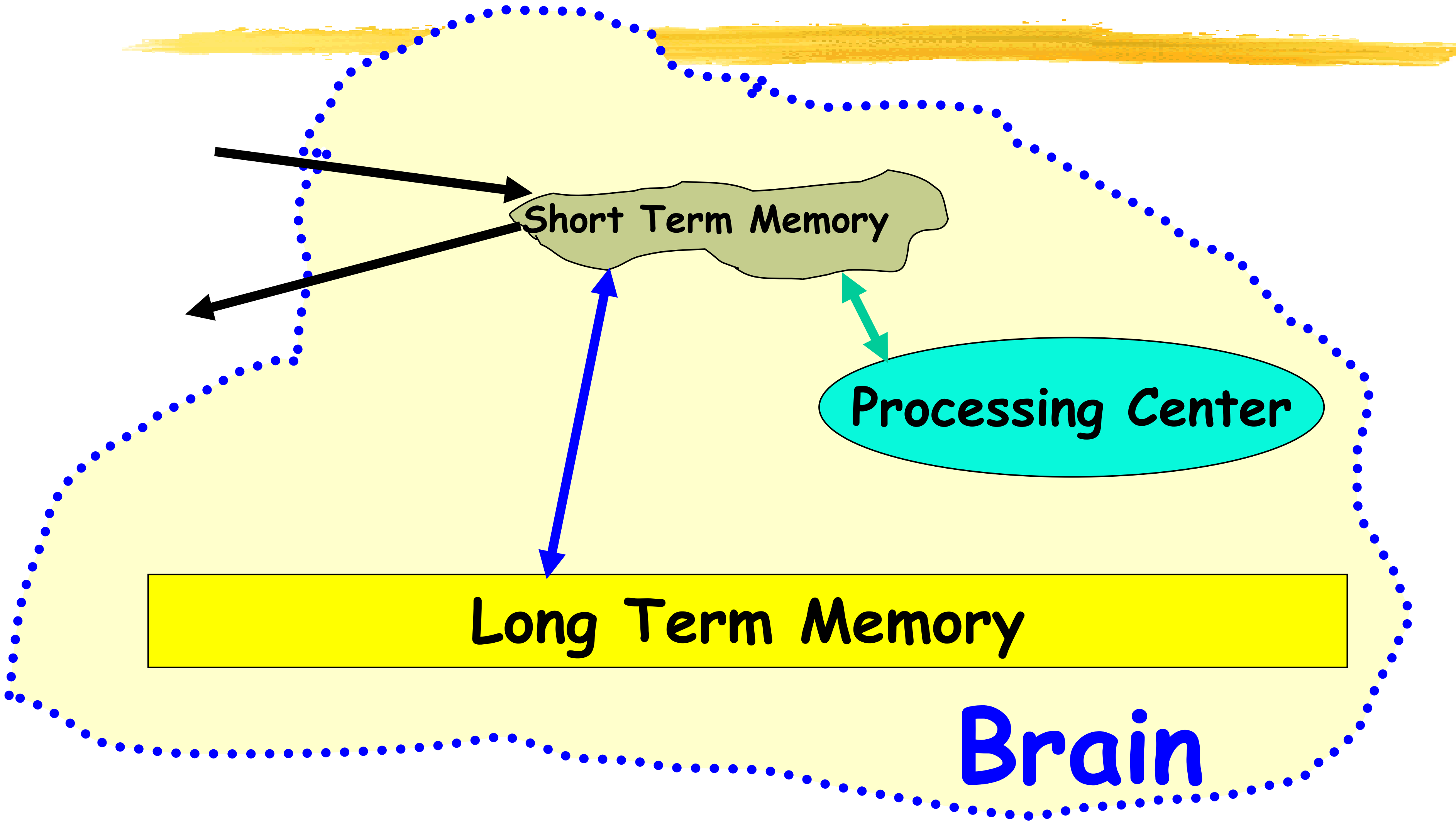
- Can successfully be used for developing rather small programs.



An Interpretation Based on Human Cognition Mechanism

- Human memory can be thought to be made up of two distinct parts [Miller 56]:
  - Short term memory and
  - Long term memory.

Schematic Representation of Brain



Short Term Memory

- An item stored in the short term memory can get lost:
  - Either due to decay with time or
  - Displacement by newer information.
- This restricts the time for which an item is stored in short term memory:
  - To few tens of seconds.
  - However, an item can be retained longer in the short term memory by recycling.

Evidence of Short Term Memory

- Short term memory is evident:
  - In many of our day-to-day experiences.
- Suppose, you look up a number from the telephone directory and start dialling it.
  - If you find the number is busy, you can dial the number again after a few seconds without having to look up the directory.
- But, after several days:
  - You may not remember the number at all
  - Would need to consult the directory again.

The Magical Number 7

- If a person deals with seven or less number of items:
  - These would be easily be accommodated in the short term memory.
  - So, he can easily understand it.
- As the number of new information increases beyond seven:
  - It becomes exceedingly difficult to understand it.

Implication in Program Development

- A small program having just a few variables:
  - Is within easy grasp of an individual.
- As the number of independent variables in the program increases:
  - It quickly exceeds the grasping power of an individual:
    - Requires an unduly large effort to master the problem.

Implication in Program Development

- Instead of a human, if a machine could be writing (generating) a program,
  - The slope of the curve would be linear.
- But, why does the effort-size curve become almost linear when software engineering principles are deployed?
  - Software engineering principles extensively use techniques specifically targeted to overcome the human cognitive limitations.

Software Engineering to Overcome Human Cognitive Limitations

- Two important principles are profusely used:
  - Abstraction
  - Decomposition

What is Abstraction?

- Simplify a problem by omitting unnecessary details.
  - Focus attention on only one aspect of the problem and ignore irrelevant details.
  - Also called model building.

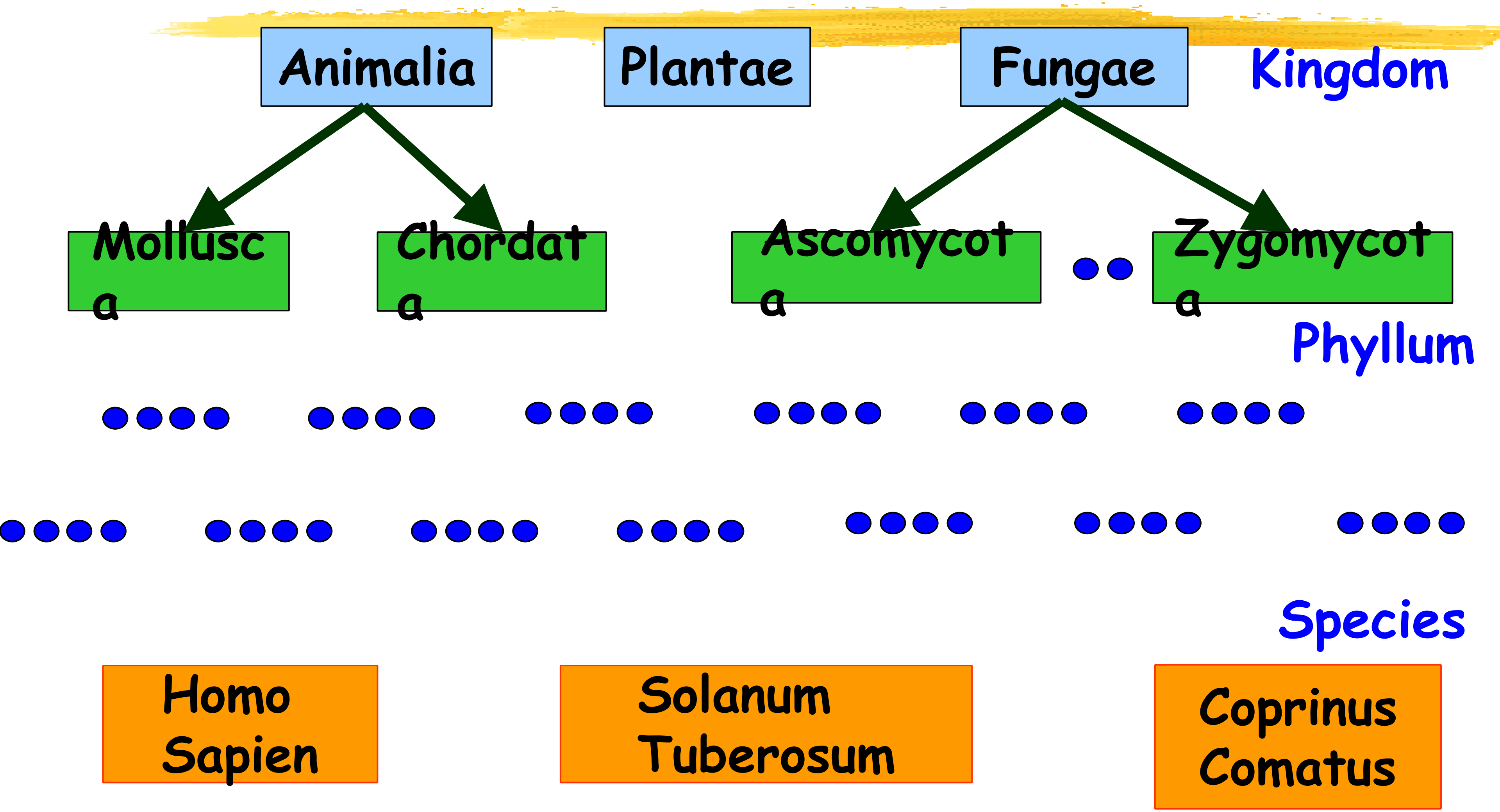
Abstraction

- For complex problems:
  - A single level of abstraction is inadequate.
  - A hierarchy of abstractions needs to be constructed.
- Hierarchy of models:
  - A model in one layer is an an abstraction of the lower layer model.
  - An implementation of the model at the higher layer.

Abstraction Example

- Suppose you are asked to understand the life forms that inhabit the earth.
- If you start examining each living organism:
  - You will almost never complete it.
  - Also, get thoroughly confused.
- You can build an abstraction hierarchy.

Living Organisms



Decomposition

- Decompose a problem into many small independent parts.
  - The small parts are then taken up one by one and solved separately.
  - The idea is that each small part would be easy to grasp and can be easily solved.
  - The full problem is solved when all the parts are solved.

Decomposition

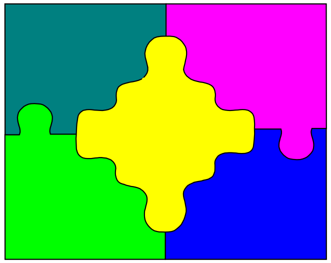
- A popular use of decomposition principle:
  - Try to break a bunch of sticks tied together versus breaking them individually.
- Any arbitrary decomposition of a problem may not help.
  - The decomposed parts must be more or less independent of each other.

Decomposition Example

- Example use of decomposition principle:
  - You understand a book better when the contents are organized into independent chapters.
  - Compared to when everything is mixed up.

Why Study Software Engineering? (Coming back)

- To acquire skills to develop large programs.
  - Handling exponential growth in complexity with size.
  - Systematic techniques based on abstraction (modelling) and decomposition.



Software Crisis

- Software products:
  - fail to meet user requirements.
  - frequently crash.
  - expensive.
  - difficult to alter, debug, and enhance.
  - often delivered late.
  - use resources non-optimally.

Factors contributing to the software crisis

- Larger problems,
- Lack of adequate training in software engineering,
- Increasing skill shortage,
- Low productivity improvements.

Programs versus Software Products

• Usually small in size	• Large
• Author himself is sole user	• Large number of users
• Single developer	• Team of developers
• Lacks proper user interface	• Well-designed interface
• Lacks proper documentation	• Well documented & user-manual prepared
• Ad hoc development.	• Systematic development

Emergence of Software Engineering

- Early Computer Programming (1950s):
  - Programs were being written in assembly language.
  - Programs were limited to about a few hundreds of lines of assembly code.

Early Computer Programming (50s)

- Every programmer developed his own style of writing programs:
  - according to his intuition (exploratory programming).

High-Level Language Programming (Early 60s)

- High-level languages such as FORTRAN, ALGOL, and COBOL were introduced:
  - This reduced software development efforts greatly.

High-Level Language Programming (Early 60s)

- Software development style was still exploratory.
  - Typical program sizes were limited to a few thousands of lines of source code.

Control Flow-Based Design
(late 60s)

- Size and complexity of programs increased further:
  - exploratory programming style proved to be insufficient.
- Programmers found:
  - very difficult to write cost-effective and correct programs.

Control Flow-Based Design
(late 60s)

- Programmers found:
  - programs written by others very difficult to understand and maintain.
- To cope up with this problem, experienced programmers advised: ``Pay particular attention to the design of the program's control structure.''

Control Flow-Based Design (late 60s)

- A program's control structure indicates:
  - the sequence in which the program's instructions are executed.
- To help design programs having good control structure:
  - flow charting technique was developed.

Control Flow-Based Design (late 60s)

- Using flow charting technique:
  - one can represent and design a program's control structure.
  - Usually one understands a program:
    - by mentally simulating the program's execution sequence.

Control Flow-Based Design

(Late 60s)

- A program having a messy flow chart representation:
  - difficult to understand and debug.

Control Flow-Based Design (Late 60s)

- It was found:
  - GO TO statements makes control structure of a program messy
  - GO TO statements alter the flow of control arbitrarily.
  - The need to restrict use of GO TO statements was recognized.

Control Flow-Based Design (Late 60s)

- Many programmers had extensively used assembly languages.
  - JUMP instructions are frequently used for program branching in assembly languages,
  - programmers considered use of GO TO statements inevitable.

Control-flow Based Design (Late 60s)

- At that time, Dijkstra published his article:
  - "Goto Statement Considered Harmful" Comm. of ACM, 1969.
- Many programmers were unhappy to read his article.

Control Flow-Based Design (Late 60s)

- They published several counter articles:
  - highlighting the advantages and inevitability of GO TO statements.

Control Flow-Based Design (Late 60s)

- But, soon it was conclusively proved:
  - only three programming constructs are sufficient to express any programming logic:
    - sequence (e.g. a=0;b=5;)
    - selection (e.g.if(c=true) k=5 else m=5;)
    - iteration (e.g. while(k>0) k=j-k;)

Control-flow Based Design (Late 60s)

- Everyone accepted:
  - it is possible to solve any programming problem without using GO TO statements.
  - This formed the basis of Structured Programming methodology.

Structured Programming

- A program is called structured
  - when it uses only the following types of constructs:
    - sequence,
    - selection,
    - iteration

Structured programs

- Structured programs are:
  - Easier to read and understand,
  - easier to maintain,
  - require less effort and time for development.

Structured Programming

- Research experience shows:
  - programmers commit less number of errors
    - while using structured if-then-else and do-while statements
    - compared to test-and-branch constructs.

Data Structure-Oriented Design (Early 70s)

- Soon it was discovered:
  - it is important to pay more attention to the design of data structures of a program
    - than to the design of its control structure.

Data Structure-Oriented Design (Early 70s)

- Techniques which emphasize designing the data structure:
  - derive program structure from it:
    - are called `data structure-oriented design techniques.`

Data Structure Oriented Design (Early 70s)

- Example of data structure-oriented design technique:
  - Jackson's Structured Programming(JSP) methodology
    - Developed in 1970s.

Data Structure Oriented Design (Early 70s)

- JSP technique:
  - program code structure should correspond to the data structure.

Data Structure Oriented Design (Early 70s)

- In JSP methodology:
  - a program's data structures are first designed using notations for
    - sequence, selection, and iteration.
  - Then data structure design is used :
    - to derive the program structure.

Data Structure Oriented Design (Early 70s)

- Several other data structure-oriented Methodologies also exist:
  - e.g., Warnier-Orr Methodology.

Data Flow-Oriented Design (Late 70s)

- Data flow-oriented techniques advocate:
  - the data items input to a system must first be identified,
  - processing required on the data items to produce the required outputs should be determined.

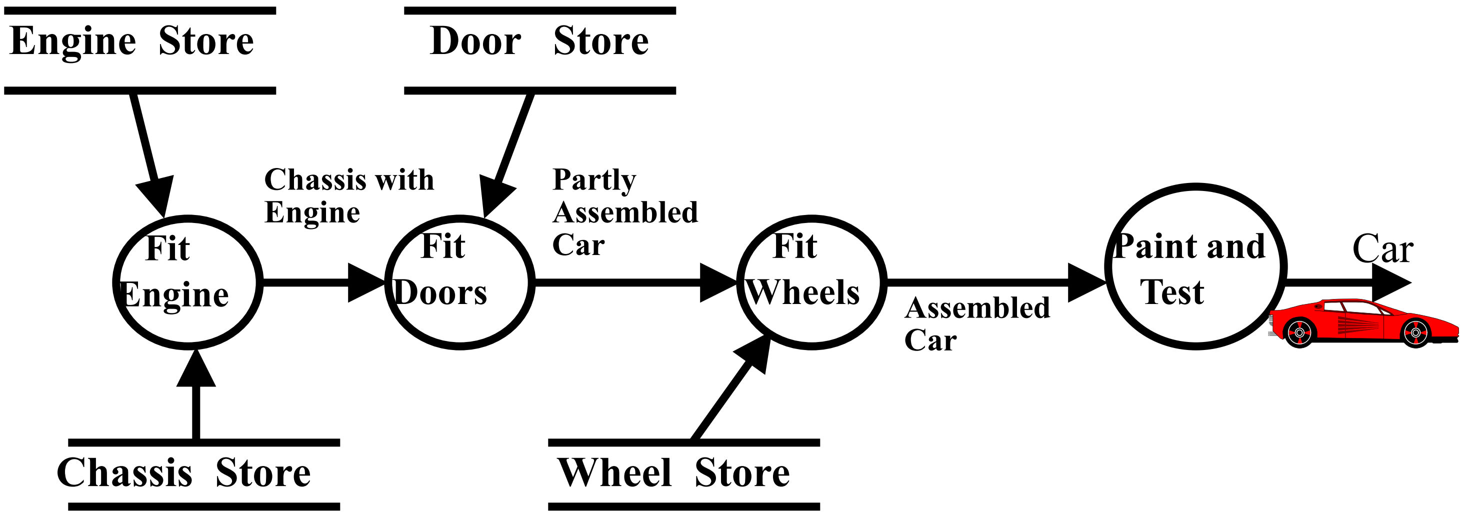
Data Flow-Oriented Design (Late 70s)

- Data flow technique identifies:
  - different processing stations (functions) in a system
  - the items (data) that flow between processing stations.

Data Flow-Oriented Design (Late 70s)

- Data flow technique is a generic technique:
  - can be used to model the working of any system
    - not just software systems.
- A major advantage of the data flow technique is its simplicity.

Data Flow Model of a Car Assembly Unit



Object-Oriented Design (80s)

- Object-oriented technique:
  - an intuitively appealing design approach:
  - natural objects (such as employees, pay-roll-register, etc.) occurring in a problem are first identified.

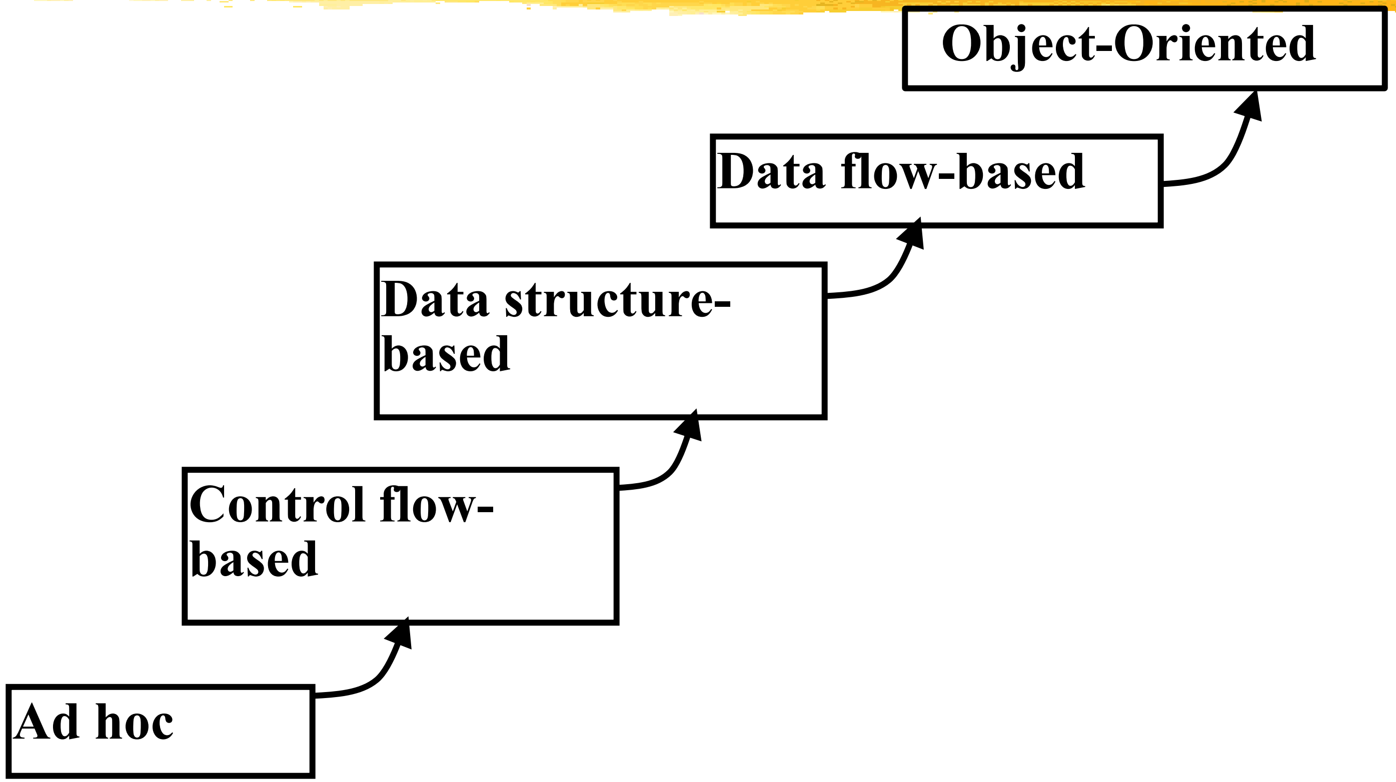
Object-Oriented Design (80s)

- Relationships among objects:
  - such as composition, reference, and inheritance are determined.
- Each object essentially acts as
  - a data hiding (or data abstraction) entity.

Object-Oriented Design (80s)

- Object-Oriented Techniques have gained wide acceptance:
  - Simplicity
  - Reuse possibilities
  - Lower development time and cost
  - More robust code
  - Easy maintenance

Evolution of Design Techniques



Evolution of Other Software Engineering Techniques

- The improvements to the software design methodologies
  - are indeed very conspicuous.
- In additions to the software design techniques:
  - several other techniques evolved.

Evolution of Other Software Engineering Techniques

- life cycle models,
- specification techniques,
- project management techniques,
- testing techniques,
- debugging techniques,
- quality assurance techniques,
- software measurement techniques,
- CASE tools, etc.

Differences between the exploratory style and modern software development practices

- Use of Life Cycle Models
- Software is developed through several well-defined stages:
  - requirements analysis and specification,
  - design,
  - coding,
  - testing, etc.

Differences between the exploratory style and modern software development practices

- Emphasis has shifted
  - from error correction to error prevention.
- Modern practices emphasize:
  - detection of errors as close to their point of introduction as possible.

Differences between the exploratory style and modern software development practices (CONT.)

- In exploratory style,
  - errors are detected only during testing,
- Now,
  - focus is on detecting as many errors as possible in each phase of development.

Differences between the exploratory style and modern software development practices (CONT.)

- In exploratory style,
  - coding is synonymous with program development.
- Now,
  - coding is considered only a small part of program development effort.

Differences between the exploratory style and modern software development practices (CONT.)

- A lot of effort and attention is now being paid to:
  - requirements specification.
- Also, now there is a distinct design phase:
  - standard design techniques are being used.

Differences between the exploratory style and modern software development practices (CONT.)

- During all stages of development process:
  - Periodic reviews are being carried out
- Software testing has become systematic:
  - standard testing techniques are available.

Differences between the exploratory style and modern software development practices (CONT.)

- There is better visibility of design and code:
  - visibility means production of good quality, consistent and standard documents.
  - In the past, very little attention was being given to producing good quality and consistent documents.
  - We will see later that increased visibility makes software project management easier.

Differences between the exploratory style and modern software development practices (CONT.)

- Because of good documentation:
  - fault diagnosis and maintenance are smoother now.
- Several metrics are being used:
  - help in software project management, quality assurance, etc.

Differences between the exploratory style and modern software development practices (CONT.)

- Projects are being thoroughly planned:
  - estimation,
  - scheduling,
  - monitoring mechanisms.

Summary

- Software engineering is:
  - systematic collection of decades of programming experience
  - together with the innovations made by researchers.

Summary

- A fundamental necessity while developing any large software product:
  - adoption of a life cycle model.

Summary

- Adherence to a software life cycle model:
  - helps to do various development activities in a systematic and disciplined manner.
  - also makes it easier to manage a software development effort.

Reference

- R. Mall, "Fundamentals of Software Engineering," Prentice-Hall of India, 2011, CHAPTER 1.

Life Cycle Models

Software Life Cycle

- Software life cycle (or software process):
  - series of identifiable stages that a software product undergoes during its life time:
    - Feasibility study
    - requirements analysis and specification,
    - design,
    - coding,
    - testing
    - maintenance.

Life Cycle Model

- A software life cycle model (or process model):
  - a descriptive and diagrammatic model of software life cycle:
  - identifies all the activities required for product development,
  - establishes a precedence ordering among the different activities,
  - Divides life cycle into phases.

Life Cycle Model (CONT.)

- Several different activities may be carried out in each life cycle phase.
  - For example, the design stage might consist of:
    - structured analysis activity followed by
    - structured design activity.

Why Model Life Cycle ?

- A written description:
  - forms a common understanding of activities among the software developers.
  - helps in identifying inconsistencies, redundancies, and omissions in the development process.
  - Helps in tailoring a process model for specific projects.

Life Cycle Model (CONT.)

- When a software product is being developed by a team:
  - there must be a precise understanding among team members as to when to do what,
  - otherwise it would lead to chaos and project failure.

Life Cycle Model (CONT.)

- A life cycle model:
  - defines entry and exit criteria for every phase.
  - A phase is considered to be complete:
    - only when all its exit criteria are satisfied.

Life Cycle Model (CONT.)

- It becomes easier for software project managers:
  - to monitor the progress of the project.

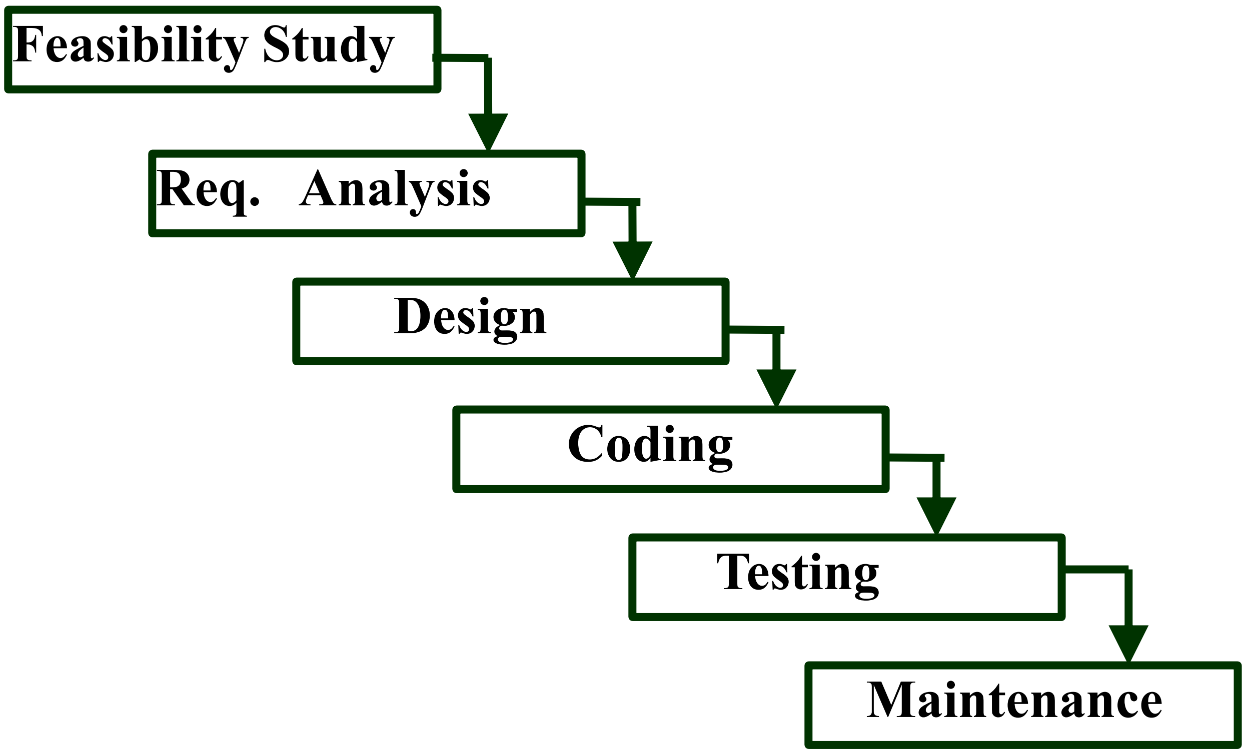
Life Cycle Model (CONT.)

- Many life cycle models have been proposed.
- We will confine our attention to a few important and commonly used models.
  - classical waterfall model
  - iterative waterfall,
  - evolutionary,
  - prototyping, and
  - spiral model

Classical Waterfall Model

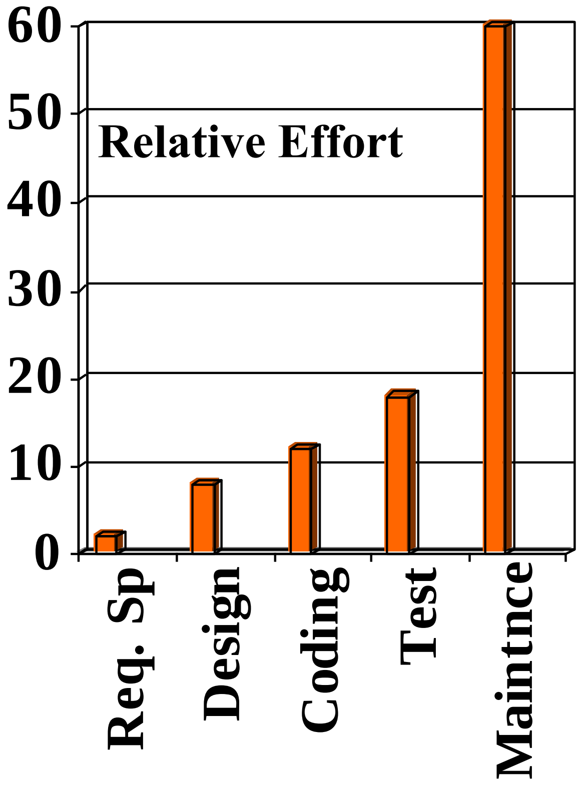
- Classical waterfall model divides life cycle into phases:
  - feasibility study,
  - requirements analysis and specification,
  - design,
  - coding and unit testing,
  - integration and system testing,
  - maintenance.

Classical Waterfall Model



Relative Effort for Phases

- **Phases between feasibility study and testing**
  - **known as development phases.**
- **Among all life cycle phases**
  - **maintenance phase consumes maximum effort.**
- **Among development phases,**
  - **testing phase consumes the maximum effort.**



Classical Waterfall Model
(CONT.)

- **Most organizations usually define:**
  - standards on the outputs (deliverables) produced at the end of every phase
  - entry and exit criteria for every phase.
- **They also prescribe specific methodologies for:**
  - specification,
  - design,
  - testing,
  - project management, etc.

Classical Waterfall Model
(CONT.)

- **The guidelines and methodologies of an organization:**
  - called the organization's software development methodology.
- **Software development organizations:**
  - expect fresh engineers to master the organization's software development methodology.

Feasibility Study

- **Main aim of feasibility study:determine whether developing the product**
  - **financially worthwhile**
  - **technically feasible.**
- **First roughly understand what the customer wants:**
  - **different data which would be input to the system,**
  - **processing needed on these data,**
  - **output data to be produced by the system,**
  - **various constraints on the behavior of the system.**

Activities during Feasibility Study

- Work out an overall understanding of the problem.
- Formulate different solution strategies.
- Examine alternate solution strategies in terms of:
  - resources required,
  - cost of development, and
  - development time.

Activities during Feasibility Study

- Perform a cost/benefit analysis:
  - to determine which solution is the best.
  - you may determine that none of the solutions is feasible due to:
    - high cost,
    - resource constraints,
    - technical reasons.

Requirements Analysis and Specification

- <u>**Aim of this phase:**</u>
  - **understand the <u>exact requirements</u> of the customer,**
  - **document them properly.**
- **Consists of two distinct activities:**
  - **requirements gathering and analysis**
  - **requirements specification.**

Goals of Requirements Analysis

- Collect all related data from the customer:
  - analyze the collected data to clearly understand what the customer wants,
  - find out any inconsistencies and incompleteness in the requirements,
  - resolve all inconsistencies and incompleteness.

Requirements Gathering

- **Gathering relevant data:**
  - **usually collected from the end-users through interviews and discussions.**
  - **For example, for a business accounting software:**
    - **interview all the accountants of the organization to find out their requirements.**

Requirements Analysis (CONT.)

- The data you initially collect from the users:
  - would usually contain several contradictions and ambiguities:
  - each user typically has only a partial and incomplete view of the system.

Requirements Analysis (CONT.)

- Ambiguities and contradictions:
  - must be identified
  - resolved by discussions with the customers.
- Next, requirements are organized:
  - into a Software Requirements Specification (SRS) document.

Requirements Analysis (CONT.)

- Engineers doing requirements analysis and specification:
  - are designated as analysts.

Design

- Design phase transforms requirements specification:
  - into a form suitable for implementation in some programming language.

Design

- **In technical terms:**
  - **during design phase, software architecture is derived from the SRS document.**
- **Two design approaches:**
  - **traditional approach,**
  - **object oriented approach.**

Traditional Design Approach

- Consists of two activities:
  - Structured analysis
  - Structured design

Structured Analysis Activity

- **Identify all the functions to be performed.**
- **Identify data flow among the functions.**
- **Decompose each function recursively into sub-functions.**
  - **Identify data flow among the subfunctions as well.**

Structured Analysis (CONT.)

- Carried out using **Data flow diagrams (DFDs)**.
- After structured analysis, carry out structured design:
  - architectural design (or high-level design)
  - detailed design (or low-level design).

Structured Design

- **<u>High-level design:</u>**
  - decompose the system into ***<u>modules</u>***,
  - represent invocation relationships among the modules.
- **<u>Detailed design:</u>**
  - different modules designed in greater detail:
    - data structures and algorithms for each module are designed.

Object Oriented Design

- First identify various objects (real world entities) occurring in the problem:
  - identify the relationships among the objects.
  - For example, the objects in a pay-roll software may be:
    - employees,
    - managers,
    - pay-roll register,
    - Departments, etc.

Object Oriented Design (CONT.)

- Object structure
  - further refined to obtain the detailed design.
- OOD has several advantages:
  - lower development effort,
  - lower development time,
  - better maintainability.

Implementation

- **Purpose of implementation phase (aka coding and unit testing phase):**
  - **translate software design into source code.**

Implementation

- During the implementation phase:
  - each module of the design is coded,
  - each module is unit tested
    - tested independently as a stand alone unit, and debugged,
  - each module is documented.

Implementation (CONT.)

- **The purpose of unit testing:**
  - **test if individual modules work correctly.**
- **The end product of implementation phase:**
  - **a set of program modules that have been tested individually.**

Integration and System Testing

- **Different modules are integrated in a planned manner:**
  - **modules are almost never integrated in one shot.**
  - **Normally integration is carried out through a number of steps.**
- **During each integration step,**
  - **the partially integrated system is tested.**

System Testing

- **After all the modules have been successfully integrated and tested:**
  - **system testing is carried out.**
- **<u>Goal of system testing:</u>**
  - ensure that the developed system functions according to its requirements as specified in the SRS document.

Maintenance

- **Maintenance of any software product:**
  - requires much more effort than the effort to develop the product itself.
  - development effort to maintenance effort is typically 40:60.

Maintenance (CONT.)

- **Corrective maintenance:**
  - Correct errors which were not discovered during the product development phases.
- **Perfective maintenance:**
  - Improve implementation of the system
  - enhance functionalities of the system.
- **Adaptive maintenance:**
  - Port software to a new environment,
    - e.g. to a new computer or to a new operating system.

Iterative Waterfall Model

- Classical waterfall model is idealistic:
  - assumes that no defect is introduced during any development activity.
  - in practice:
    - defects do get introduced in almost every phase of the life cycle.

Iterative Waterfall Model
(CONT.)

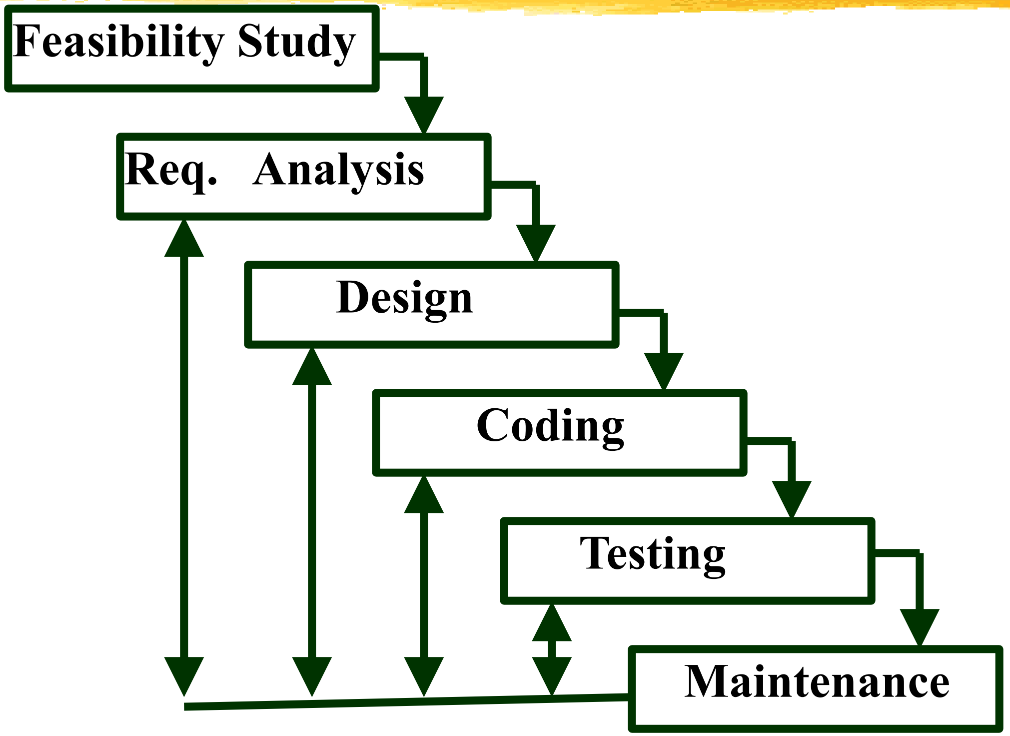
- **Defects usually get detected much later in the life cycle:**
  - For example, a design defect might go unnoticed till the coding or testing phase.

Iterative Waterfall Model
(CONT.)

- Once a defect is detected:
  - we need to go back to the phase where it was introduced
  - redo some of the work done during that and all subsequent phases.
- Therefore we need feedback paths in the classical waterfall model.

Iterative Waterfall Model

(CONT.)



Iterative Waterfall Model
(CONT.)

- **Errors should be detected**
  - in the same phase in which they are introduced.
- **For example:**
  - **if a design problem is detected in the design phase itself,**
    - the problem can be taken care of much more easily
    - than say if it is identified at the end of the integration and system testing phase.

Phase containment of errors

- Reason: rework must be carried out not only to the design but also to code and test phases.
- **The principle of detecting errors as close to its point of introduction as possible:**
  - is known as **phase containment of errors.**
- Iterative waterfall model is by far the most widely used model.
  - Almost every other model is derived from the waterfall model.

Shortcomings of Iterative Waterfall Model

- Does not consider risks
  - Does not consider uncertainties concerning the requirements
  - Cannot be used by customer’s having rough or unclear idea of her requirements
- Rigid Phase Sequence
  - May lead to blocking states
  - May lead to some efficient engineers sitting idly

Classical Waterfall Model
(CONT.)

- Irrespective of the life cycle model actually followed:
  - the documents should reflect a classical waterfall model of development,
  - comprehension of the documents is facilitated.

Classical Waterfall Model
(CONT.)

- **Metaphor of mathematical theorem proving:**
  - **A mathematician presents a proof as a single chain of deductions,**
    - **even though the proof might have come from a convoluted set of partial attempts, blind alleys and backtracks.**

Prototyping Model

- Before starting actual development,
  - a working prototype of the system should first be built.
- A prototype is a toy implementation of a system:
  - limited functional capabilities,
  - low reliability,
  - inefficient performance.

Reasons for developing a prototype

- Illustrate to the customer:
  - input data formats, messages, reports, or interactive dialogs.
- Examine technical issues associated with product development:
  - Often major design decisions depend on issues like:
    - response time of a hardware controller,
    - efficiency of a sorting algorithm, etc.

Prototyping Model (CONT.)

- The third reason for developing a prototype is:
  - it is impossible to ``get it right'' the first time,
  - we must plan to throw away the first product
    - if we want to develop a good product.

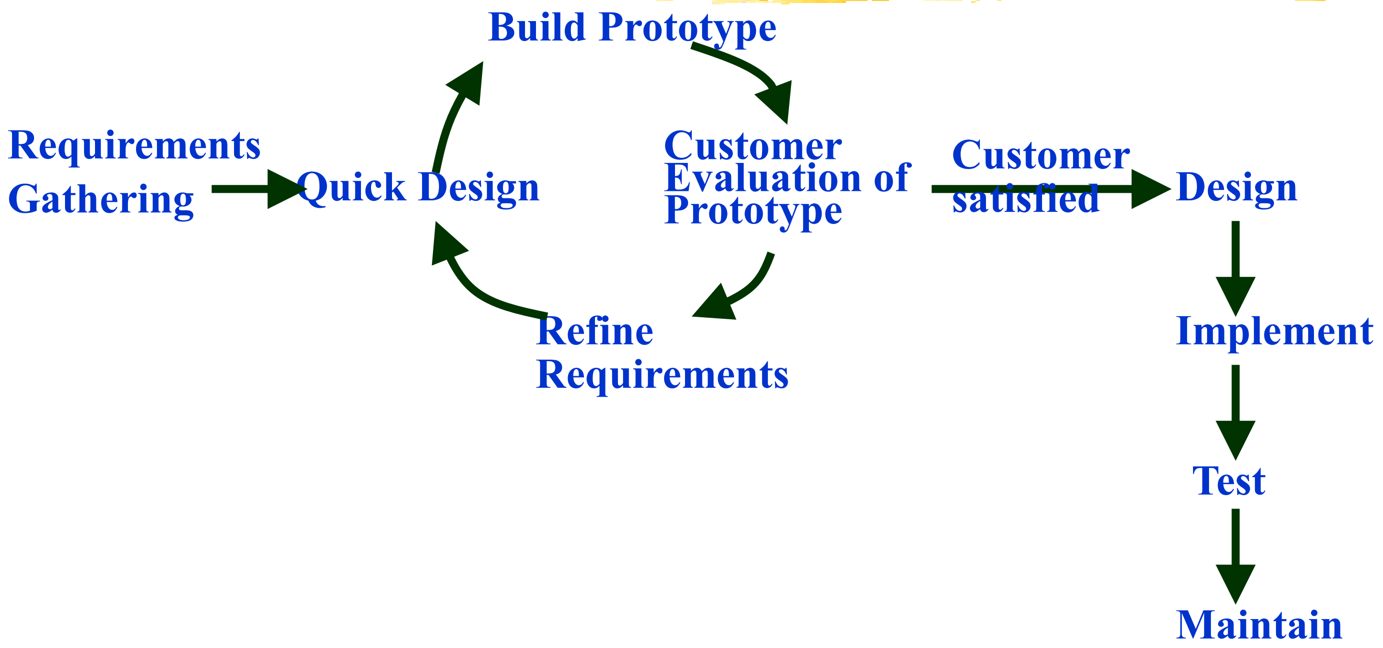
Prototyping Model (CONT.)

- **Start with approximate requirements.**
- **Carry out a quick design.**
- **Prototype model is built using several short-cuts:**
  - Short-cuts might involve using inefficient, inaccurate, or dummy functions.
    - A function may use a table look-up rather than performing the actual computations.

Prototyping Model (CONT.)

- **The developed prototype is submitted to the customer for his evaluation:**
  - Based on the user feedback, requirements are refined.
  - This cycle continues until the user approves the prototype.
- **The actual system is developed using the classical waterfall approach.**

Prototyping Model (CONT.)



Prototyping Model (CONT.)

- Requirements analysis and specification phase becomes redundant:
  - final working prototype (with all user feedbacks incorporated) serves as an **animated requirements specification.**
- Design and code for the prototype is usually thrown away:
  - However, the experience gathered from developing the prototype helps a great deal while developing the actual product.

Prototyping Model (CONT.)

- **Even though construction of a working prototype model involves additional cost --- overall development cost might be lower for:**
  - systems with unclear user requirements,
  - systems with unresolved technical issues.
- **Many user requirements get properly defined and technical issues get resolved:**
  - these would have appeared later as change requests and resulted in incurring massive redesign costs.

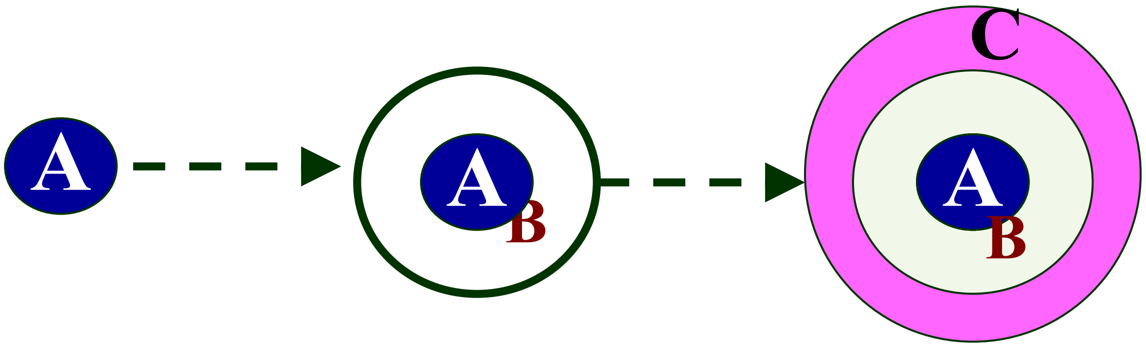
Evolutionary Model

- Evolutionary model (aka successive versions or incremental model):
  - The system is broken down into several modules which can be incrementally implemented and delivered.
- First develop the core modules of the system.
- The initial product skeleton is refined into increasing levels of capability:
  - by adding new functionalities in successive versions.

Evolutionary Model (CONT.)

- Successive version of the product:
  - functioning systems capable of performing some useful work.
  - A new release may include new functionality:
    - also existing functionality in the current release might have been enhanced.

Evolutionary Model (CONT.)



Advantages of Evolutionary Model

- Users get a chance to experiment with a partially developed system:
  - much before the full working version is released,
- Helps finding exact user requirements:
  - much before fully working system is developed.
- Core modules get tested thoroughly:
  - reduces chances of errors in final product.

Disadvantages of Evolutionary Model

- **Often, difficult to subdivide problems into functional units:**
  - **which can be incrementally implemented and delivered.**
  - **evolutionary model is useful for very large problems,**
    - **where it is easier to find modules for incremental implementation.**

Evolutionary Model with Iteration

- **Many organizations use a combination of iterative and incremental development:**
  - **a new release may include new functionality**
  - **existing functionality from the current release may also have been modified.**

Evolutionary Model with iteration

- Several advantages:
  - Training can start on an earlier release
    - customer feedback taken into account
  - Markets can be created:
    - for functionality that has never been offered.
  - Frequent releases allow developers to fix unanticipated problems quickly.

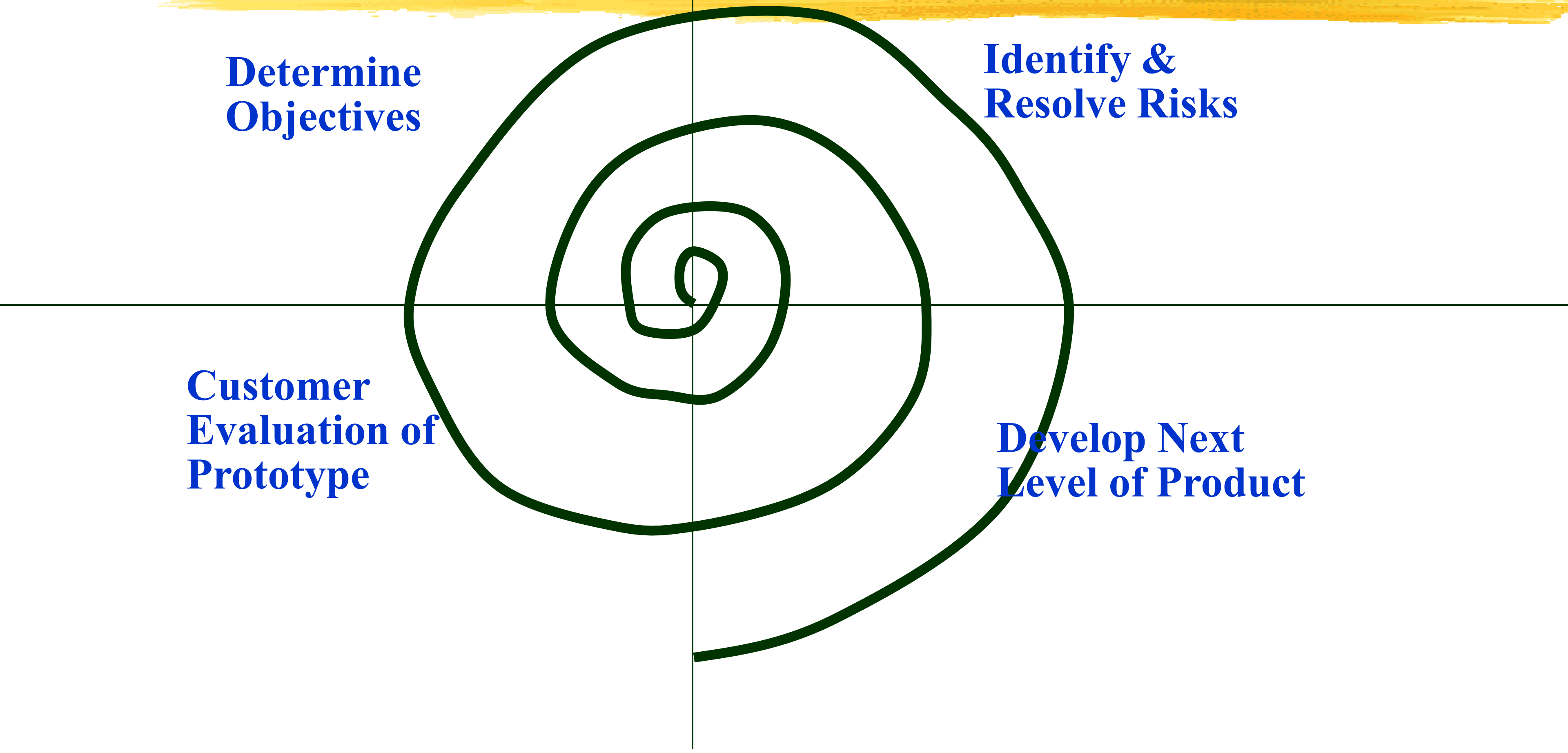
Spiral Model

- Proposed by Boehm in 1988.
- Each loop of the spiral represents a phase of the software process:
  - the innermost loop might be concerned with system feasibility,
  - the next loop with system requirements definition,
  - the next one with system design, and so on.
- There are no fixed phases in this model, the phases shown in the figure are just examples.

Spiral Model (CONT.)

- The team must decide:
  - how to structure the project into phases.
- Start work using some generic model:
  - add extra phases
    - for specific projects or when problems are identified during a project.
- Each loop in the spiral is split into four sectors (quadrants).

Spiral Model (CONT.)



Objective Setting (First Quadrant)

- **Identify objectives of the phase,**
- **Examine the risks associated with these objectives.**
  - **Risk:**
    - any adverse circumstance that might hamper successful completion of a software project.
- **Find alternate solutions possible.**

Risk Assessment and Reduction (Second Quadrant)

- For each identified project risk,
  - a detailed analysis is carried out.
- Steps are taken to reduce the risk.
- For example, if there is a risk that the requirements are inappropriate:
  - a prototype system may be developed.

Spiral Model (CONT.)

- Development and Validation (Third quadrant):
  - develop and validate the next level of the product.
- Review and Planning (Fourth quadrant):
  - review the results achieved so far with the customer and plan the next iteration around the spiral.
- With each iteration around the spiral:
  - progressively more complete version of the software gets built.

Spiral Model as a meta model

- **Subsumes all discussed models:**
  - a single loop spiral represents waterfall model.
  - uses an evolutionary approach --
    - iterations through the spiral are evolutionary levels.
  - enables understanding and reacting to risks during each iteration along the spiral.
  - uses:
    - prototyping as a risk reduction mechanism
    - retains the step-wise approach of the waterfall model

Comparison of Different Life Cycle Models

- **Iterative waterfall model**
  - most widely used model.
  - But, suitable only for well-understood problems.
- **Prototype model is suitable for projects not well understood:**
  - **user requirements**
  - **technical aspects**

Comparison of Different Life Cycle Models (CONT.)

- **Evolutionary model is suitable for large problems:**
  - can be decomposed into a set of modules that can be incrementally implemented,
  - incremental delivery of the system is acceptable to the customer.
- **The spiral model:**
  - suitable for development of technically challenging software products that are subject to several kinds of risks.

Requirements Analysis and Specification

Organization of this Lecture

- Brief review of previous lectures
- Introduction
- Requirements analysis
- Requirements specification
- SRS document
- Decision table
- Decision tree
- Summary

Requirements Analysis and Specification

- Many projects fail:
  - because they start implementing the system:
  - without determining whether they are building what the customer really wants.

Requirements Analysis and Specification

- It is important to learn:
  - requirements analysis and specification techniques thoroughly.

Requirements Analysis and Specification

- Goals of requirements analysis and specification phase:
  - fully understand the user requirements
  - remove inconsistencies, anomalies, etc. from requirements
  - document requirements properly in an SRS document

Requirements Analysis and Specification

- Consists of two distinct activities:
  - Requirements Gathering and Analysis
  - Specification

Requirements Analysis and Specification

- The person who undertakes requirements analysis and specification:
  - known as systems analyst:
  - collects data pertaining to the product
  - analyzes collected data:
    - to understand what exactly needs to be done.
  - writes the Software Requirements Specification (SRS) document.

Requirements Analysis and Specification

- Final output of this phase:
  - Software Requirements Specification (SRS) Document.
- The SRS document is reviewed by the customer.
  - reviewed SRS document forms the basis of all future development activities.

Requirements Analysis

- Requirements analysis consists of two main activities:
  - Requirements gathering
  - Analysis of the gathered requirements

Requirements Analysis

- Analyst gathers requirements through:
  - Studying the existing documentation
  - Interviewing the customer and end-users
  - Analysis of what needs to be done
    - Task Analysis
    - Scenario Analysis
  - Form analysis

Requirements Gathering

- If the project is to automate some existing procedures
  - e.g., automating existing manual accounting activities,
  - the task of the system analyst is a little easier
  - analyst can immediately obtain:
    - input and output formats
    - accurate details of the operational procedures

Requirements Gathering
(CONT.)

- In the absence of a working system,
  - lot of imagination and creativity are required.
- Interacting with the customer to gather relevant data:
  - requires a lot of experience.

Requirements Gathering
(CONT.)

- Some desirable attributes of a good system analyst:
  - Good interaction skills,
  - imagination and creativity,
  - experience.

Analysis of the Gathered Requirements

- After gathering all the requirements:
  - analyze it:
    - Clearly understand the user requirements,
    - Detect inconsistencies, ambiguities, and incompleteness.
- Incompleteness and inconsistencies:
  - resolved through further discussions with the end-users and the customers.

Inconsistent requirement

- Some part of the requirement:
  - contradicts with some other part.
  - Two end-users give inconsistent description of the requirement.
- <u>Example:</u>
  - One customer says turn off heater and open water shower when temperature > 100 C
  - Another customer says turn off heater and turn ON cooler when temperature > 100 C

Ambiguity / Anomaly

- Several interpretations of the requirement are possible
- Example:
  - If you are absent for long you won’t be allowed for exams
  - If temperature becomes high, heater should be switched off
  - ‘Long’, ‘high’ are not defined: ambiguous

Incomplete requirement

- Some requirements have been overlooked:
  - Realized by the customer much later, possibly during usage
- <u>Example:</u>
  - The analyst has not recorded: when temperature falls below 90 C
    - heater should be turned ON
    - water shower turned OFF.

Analysis of the Gathered Requirements (CONT.)

- Requirements analysis involves:
  - obtaining a clear, in-depth understanding of the product to be developed,
  - remove all ambiguities and inconsistencies from the initial customer perception of the problem.

Analysis of the Gathered Requirements (CONT.)

- It is quite difficult to obtain:
  - a clear, in-depth understanding of the problem:
    - especially if there is no working model of the problem.

Analysis of the Gathered Requirements (CONT.)

- Experienced analysts take considerable time:
  - to understand the exact requirements the customer has in his mind.

Analysis of the Gathered Requirements (CONT.)

- Experienced systems analysts know -
  often as a result of painful experiences ---
  - without a clear understanding of the problem, it is impossible to develop a satisfactory system.

Analysis of the Gathered Requirements(CONT.)

- Several things about the project should be clearly understood by the analyst, in order to gain a good grasp of the problem:
  - What is the problem?
  - Why is it important to solve the problem?
  - What are the possible solutions to the problem?
  - What complexities might arise while solving the problem?

Analysis of the Gathered Requirements(CONT.)

- Some anomalies and inconsistencies can be very subtle:
  - escape even most experienced eyes.
  - If a formal model of the system is constructed,
    - many of the subtle anomalies and inconsistencies get detected.

Analysis of the Gathered Requirements(CONT.)

- After collecting all data regarding the system to be developed,
  - remove all inconsistencies and anomalies from the requirements,
  - systematically organize requirements into a Software Requirements Specification (SRS) document.

Software Requirements Specification

- Main aim of requirements specification:
  - systematically organize the requirements arrived during requirements analysis
  - document requirements properly.

Software Requirements Specification

- The SRS document is useful in various contexts:
  - Statement of user needs
  - Contract document
  - Test document
  - Goals of implementation

Software Requirements Specification: A Contract Document

- Requirements document is a reference document.
- SRS document is a contract between the development team and the customer.
  - Once the SRS document is approved by the customer,
    - any subsequent controversies are settled by referring the SRS document.

Software Requirements Specification: A Contract Document

- Once customer agrees to the SRS document:
  - development team starts to develop the product according to the requirements recorded in the SRS document.
- The final product will be acceptable to the customer:
  - as long as it satisfies all the requirements recorded in the SRS document.

SRS Document (CONT.)

- The SRS document is known as black-box specification:
  - the system is considered as a black box whose internal details are not known.
  - only its visible external (i.e. input/output) behavior is documented.

SRS Document (CONT.)

- SRS document concentrates on:
  - what needs to be done
  - carefully avoids the solution ("how to do") aspects.
- The SRS document serves as a contract
  - between development team and the customer.
  - Should be carefully written

SRS Document (CONT.)

- The requirements at this stage:
  - written using end-user terminology.
- If necessary:
  - later a formal requirement specification may be developed from it.

Properties of a good SRS document

- It should be concise
  - and at the same time should not be ambiguous.
- It should specify what the system must do
  - and not say how to do it.
- Easy to change.,
  - i.e. it should be well-structured.

Properties of a good SRS document (cont...)

- It should be traceable
  - you should be able to trace which part of the specification corresponds to which part of the design and code, etc and vice versa.
- It should be verifiable
  - e.g. "system should be user friendly" is not verifiable

SRS Document Organization

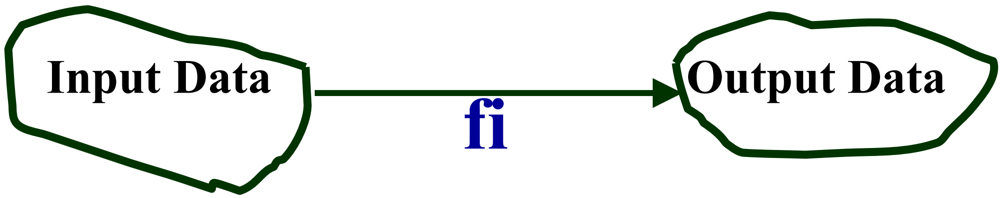
- Introduction
  - Purpose
  - Overview
- Goals of Implementation
- **Functional Requirements**
  - **Functional Requirement 1**
  - **Functional Requirement 2**
  - **…**
- **Non-Functional Requirements**
  - **External Interfaces**
  - **User Interfaces**
  - **Software Interfaces**
  - **Performance**

SRS Document (CONT.)

- SRS document, normally contains three important parts:
  - functional requirements,
  - nonfunctional requirements,
  - constraints.

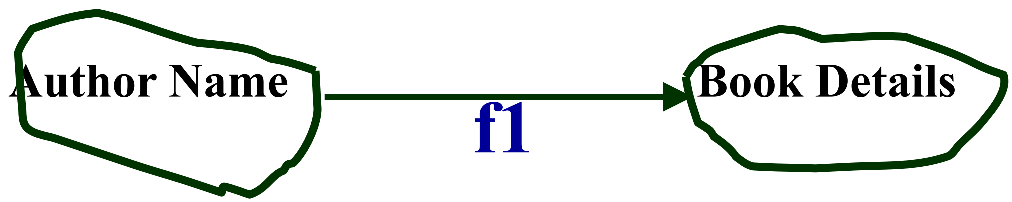
SRS Document (CONT.)

- It is desirable to consider every system:
  - performing a set of functions {fi}.
  - Each function fi considered as:
  - transforming a set of input data to corresponding output data.



Example: Functional Requirement

- F1: Search Book
  - Input:
    - an author's name:
  - Output:
    - details of the author's books and the locations of these books in the library.



Functional Requirements

- Functional requirements describe:
  - A set of high-level requirements
  - Each high-level requirement:
    - takes in some data from the user
    - outputs some data to the user
  - Each high-level requirement:
    - might consist of a set of identifiable functions

Functional Requirements

- For each high-level requirement:
  - every function is described in terms of
    - input data set
    - output data set
    - processing required to obtain the output data set from the input data set

Nonfunctional Requirements

- Characteristics of the system which can not be expressed as functions:
  - maintainability,
  - portability,
  - usability, etc.

Nonfunctional Requirements

- Nonfunctional requirements include:
  - reliability issues,
  - performance issues,
  - human-computer interface issues,
  - Interface with other external systems,
  - security, maintainability, etc.

Constraints

- Constraints describe things that the system should or should not do.
  - For example,
    - standards compliance
    - how fast the system can produce results
      - so that it does not overload another system to which it supplies data, etc.

Examples of constraints

- Hardware to be used,
- Operating system
  - or DBMS to be used
- Capabilities of I/O devices
- Standards compliance
- Data representations
  - by the interfaced system

Organization of the SRS Document

- Introduction.
- Functional Requirements
- Nonfunctional Requirements
  - External interface requirements
  - Performance requirements
- Constraints

Example Functional Requirements

- List all functional requirements
  - with proper numbering.
- Req. 1:
  - Once the user selects the “search” option,
    - he is asked to enter the key words.
  - The system should output details of all books
    - whose title or author name matches any of the key words entered.
    - Details include: Title, Author Name, Publisher name, Year of Publication, ISBN Number, Catalog Number, Location in the Library.

Example Functional Requirements

- Req. 2:
  - When the "renew" option is selected,
    - the user is asked to enter his membership number and password.
  - After password validation,
    - the list of the books borrowed by him are displayed.
  - The user can renew any of the books:
    - by clicking in the corresponding renew box.

Req. 1:

- R.1.1:
  - Input: “search” option,
  - Output: user prompted to enter the key words.
- R1.2:
  - Input: key words
  - Output: Details of all books whose title or author name matches any of the key words.
    - Details include: Title, Author Name, Publisher name, Year of Publication, ISBN Number, Catalog Number, Location in the Library.
  - Processing: Search the book list for the keywords

Req. 2:

- R2.1:
  - Input: “renew” option selected,
  - Output: user prompted to enter his membership number and password.
- R2.2:
  - Input: membership number and password
  - Output:
    - list of the books borrowed by user are displayed. User prompted to enter books to be renewed or
    - user informed about bad password
  - Processing: Password validation, search books issued to the user from borrower list and display.

Req. 2:

- R2.3:
  - Input: user choice for renewal of the books issued to him through mouse clicks in the corresponding renew box.
  - Output: Confirmation of the books renewed
  - Processing: Renew the books selected by the in the borrower list.

Examples of Bad SRS Documents

- Unstructured Specifications:
  - Narrative essay --- one of the worst types of specification document:
    - Difficult to change,
    - difficult to be precise,
    - difficult to be unambiguous,
    - scope for contradictions, etc.

Examples of Bad SRS Documents

- Noise:
  - Presence of text containing information irrelevant to the problem.
- Silence:
  - aspects important to proper solution of the problem are omitted.

Examples of Bad SRS Document

- Overspecification:
  - Addressing "how to" aspects
  - For example, "Library member names should be stored in a sorted descending order"
  - Overspecification restricts the solution space for the designer.
- Forward References:
  - References to aspects of problem
    - defined only later on in the text.
- Wishful Thinking:
  - Descriptions of aspects
    - for which realistic solutions will be hard to find.

Representation of complex processing logic:

- Decision trees
- Decision tables

Decision Trees

- Decision trees:
  - edges of a decision tree represent conditions
  - leaf nodes represent actions to be performed.
- A decision tree gives a graphic view of:
  - logic involved in decision making
  - corresponding actions taken.

Example: LMS

- A Library Membership automation Software (LMS) should support the following three options:
  - new member,
  - renewal,
  - cancel membership.

Example: LMS

- When the new member option is selected,
  - the software asks details about the member:
    - name,
    - address,
    - phone number, etc.

Example(cont.)

- If proper information is entered,
  - a membership record for the member is created
  - a bill is printed for the annual membership charge plus the security deposit payable.

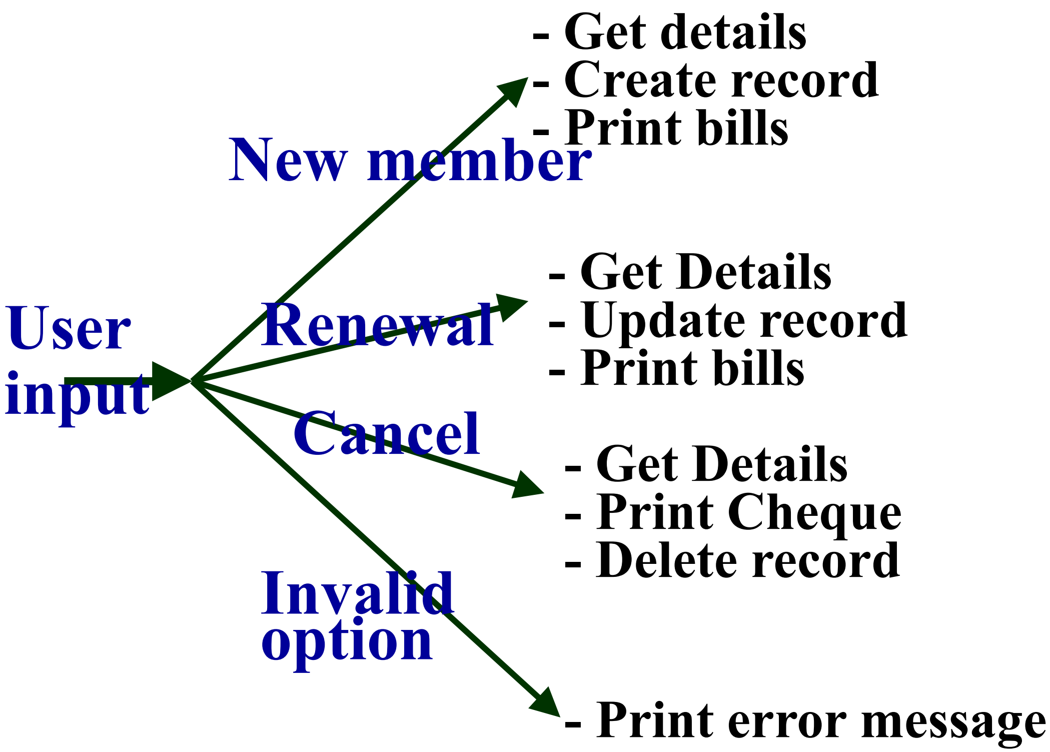
Example(cont.)

- If the renewal option is chosen,
  - LMS asks the member's name and his membership number
    - checks whether he is a valid member.
  - If the name represents a valid member,
    - the membership expiry date is updated and the annual membership bill is printed,
    - otherwise an error message is displayed.

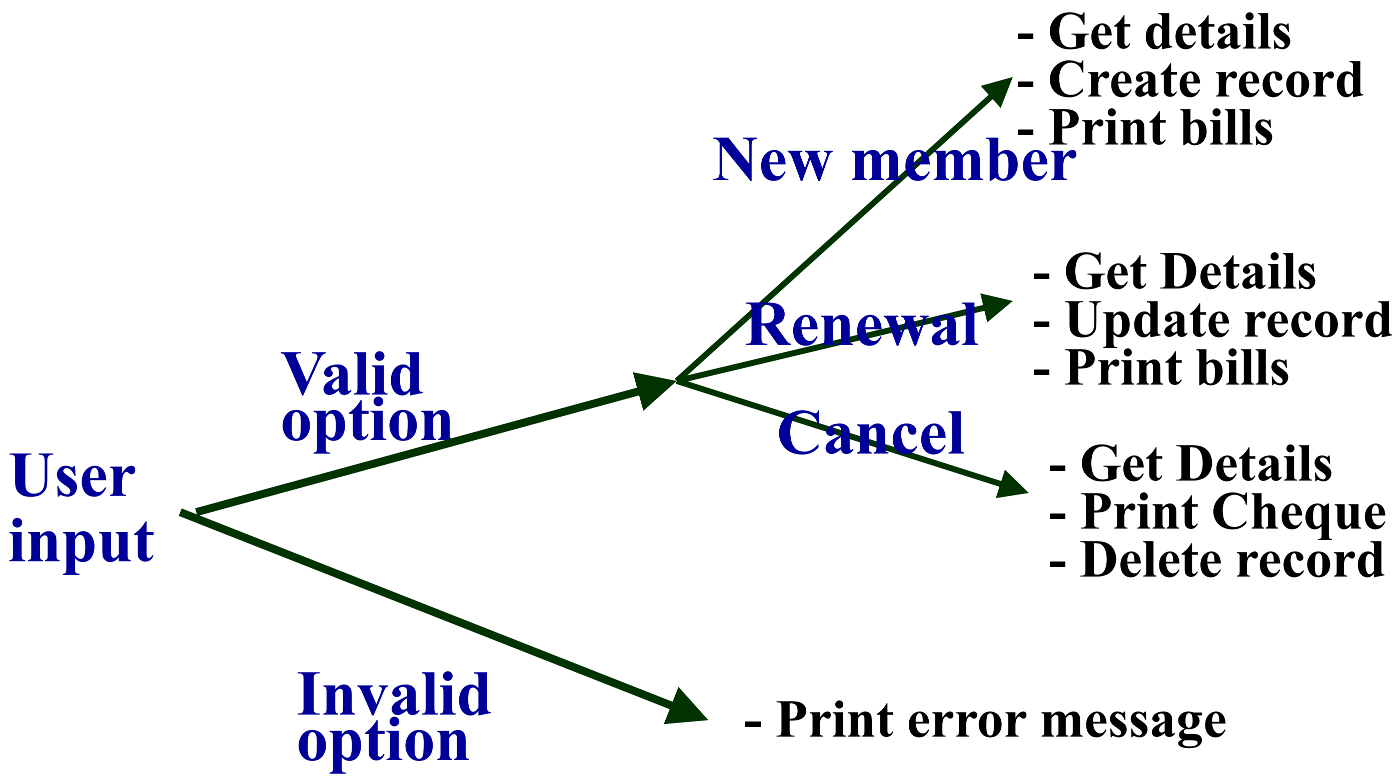
Example(cont.)

- If the cancel membership option is selected and the name of a valid member is entered,
  - the membership is cancelled,
  - a cheque for the balance amount due to the member is printed
  - the membership record is deleted.

Decision Tree



Decision Tree



Decision Table

- Decision tables specify:
  - which variables are to be tested
  - what actions are to be taken if the conditions are true,
  - the order in which decision making is performed.

Decision Table

- A decision table shows in a tabular form:
  - processing logic and corresponding actions
- Upper rows of the table specify:
  - the variables or conditions to be evaluated
- Lower rows specify:
  - the actions to be taken when the corresponding conditions are satisfied.

Decision Table

- In technical terminology,
  - a column of the table is called a <u>rule</u>:
  - A rule implies:
    - if a condition is true, then execute the corresponding action.

Example:

- Conditions

Valid selection	NO	YES	YES	YES
New member	--	YES	NO	NO
Renewal	--	NO	YES	NO
Cancellation	--	NO	NO	YES

- Actions

Display error message	⊗			
Ask member's name etc.		⊗		
Build customer record		⊗		
Generate bill		⊗	⊗	
Ask membership details			⊗	⊗
Update expiry date			⊗	
Print cheque				⊗
Delete record				⊗

Comparison

- Both decision tables and decision trees
  - can represent complex program logic.
- Decision trees are easier to read and understand
  - when the number of conditions are small.
- Decision tables help to look at every possible combination of conditions.

Comparison

- Order of decision making is abstracted out in decision tables
  - Decision trees support multi-level or hierarchical decision making
- Decision tables are appropriate where very large number of decisions is involved
  - Decision trees become complex

Formal Specification

- A formal specification technique is a mathematical method to:
  - accurately specify a system
  - verify that implementation satisfies specification
  - prove properties of the specification

Formal Specification

- Advantages:
  - Well-defined semantics, no scope for ambiguity
  - Automated tools can check properties of specifications
  - Executable specification

Formal Specification

- Disadvantages of formal specification techniques:
  - Difficult to learn and use
  - Not able to handle complex systems

Formal Specification

- Mathematical techniques used include:
  - Logic-based
  - set theoretic
  - algebraic specification
  - finite state machines, etc.

Semiformal Specification

- Structured specification languages
  - SADT (Structured Analysis and Design Technique)
  - PSL/PSA (Problem Statement Language/Problem Statement Analyzer)
    - PSL is a semi-formal specification language
    - PSA can analyze the specifications expressed in PSL

Executable Specification Language

- If specification is expressed in formal language:
  - it becomes possible to execute the specification to provide a system prototype.
- However, executable specifications are usually slow and inefficient.

Executable Specification Language

- Executable specifications only test functional requirements:
  - If non-functional requirements are important for some product,
    - the utility of an executable specification prototype is limited.

4GLs

- 4GLs (Fourth Generation Languages) are examples of
  - executable specification languages.
- 4GLs are successful
  - because there is a lot of commonality across data processing applications.

4GLs

- 4GLs rely on software reuse
  - where common abstractions have been identified and parameterized.
- Rewriting 4GL programs in higher level languages:
  - result in upto 50% lower memory requirements
  - also the programs run upto 10 times faster.

Summary

- Requirements analysis and specification
  - an important phase of software development:
  - any error in this phase would affect all subsequent phases of development.
- Consists of two different activities:
  - Requirements gathering and analysis
  - Requirements specification

Summary

- The aims of requirements analysis:
  - Gather all user requirements
  - Clearly understand exact user requirements
  - Remove inconsistencies and incompleteness.
- The goal of specification:
  - systematically organize requirements
  - document the requirements in an SRS document.

Summary

- Main components of SRS document:
  - functional requirements
  - nonfunctional requirements
  - constraints
- Techniques to express complex logic:
  - Decision tree
  - Decision table

Summary

- Formal requirements specifications have several advantages.
  - But the major shortcoming is that these are hard to use.

Software Design

Organization of this Lecture

- Brief review of previous lectures
- Introduction to software design
- Goodness of a design
- Functional Independence
- Cohesion and Coupling
- Function-oriented design vs. Object-oriented design
- Summary

Review of previous lectures

- Introduction to software engineering
- Life cycle models
- Requirements Analysis and Specification:
  - Requirements gathering and analysis
  - Requirements specification

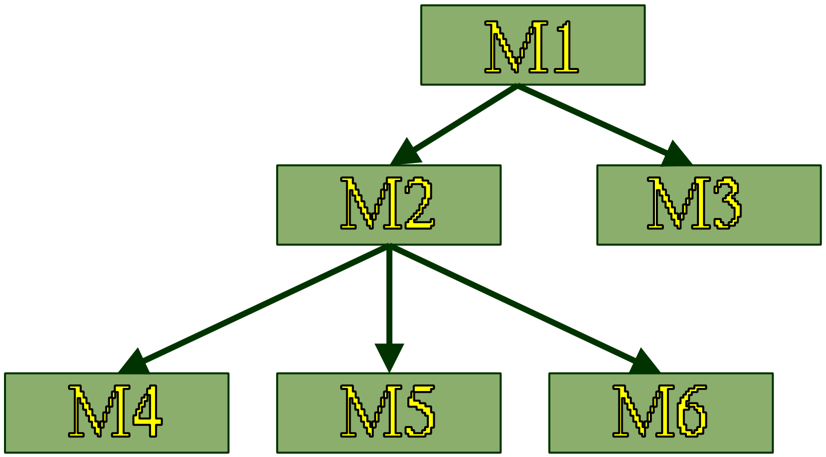
Introduction

- Design phase transforms SRS document:
  - into a form easily implementable in some programming language.

Items Designed During Design Phase

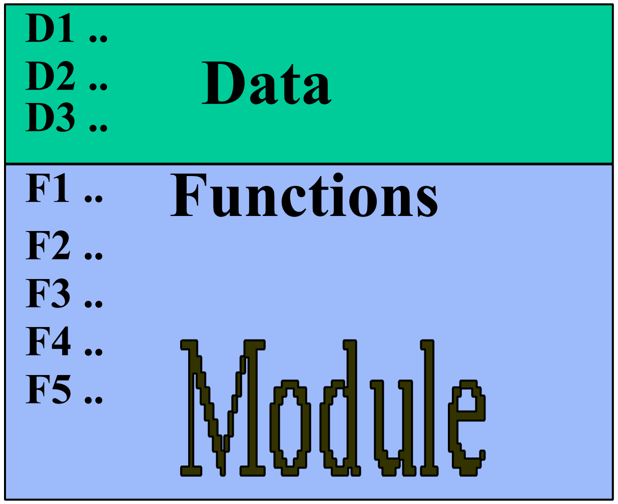
- module structure,
- control relationship among the modules
  - call relationship or invocation relationship
- interface among different modules,
  - data items exchanged among different modules,
- data structures of individual modules,
- algorithms for individual modules.

Module Structure



Introduction

- A module consists of:
  - several functions
  - associated data structures.



Introduction

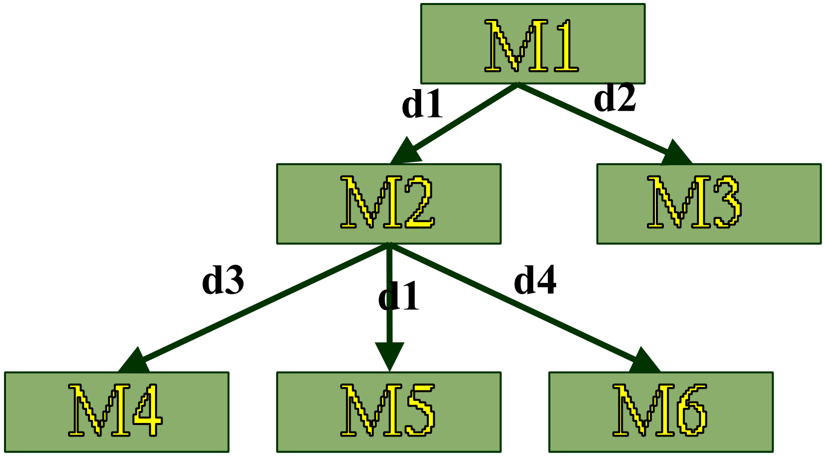
- Good software designs:
  - seldom arrived through a single step procedure:
  - but through a series of steps and iterations, called the design activities.

Introduction

- Design activities are usually classified into two stages:
  - preliminary (or high-level) design
  - detailed design.
- Meaning and scope of the two stages:
  - vary considerably from one methodology to another.

High-level design

- Identify:
  - modules
  - control relationships among modules
  - interfaces among modules.



High-level design

- The outcome of high-level design:
  - program structure (or software architecture).

High-level Design

- Several notations are available to represent high-level design:
  - Usually a tree-like diagram called structure chart is used.
  - Other notations:
    - Jackson diagram or Warnier-Orr diagram can also be used.

Detailed design

- For each module, design:
  - data structure
  - algorithms
- Outcome of detailed design:
  - module specification.

A fundamental question:

- How to distinguish between good and bad designs?
  - Unless we know what a good software design is:
    - we can not possibly design one.

Good and bad designs

- There is no unique way to design a system.
- Even using the same design methodology:
  - different engineers can arrive at very different design solutions.
- We need to distinguish between good and bad designs.

What Is Good Software Design?

- Should implement all functionalities of the system correctly.
- Should be easily understandable.
- Should be efficient
  - Resource, time and cost optimization issues
- Should be easily amenable to change,
  - i.e. easily maintainable.

What Is Good Software Design?

- Understandability of a design is a major issue:
  - determines goodness of design:
  - a design that is easy to understand:
    - also easy to maintain and change.

What Is Good Software Design?

- Unless a design is easy to understand,
  - tremendous effort needed to maintain it
  - We already know that about 60% effort is spent in maintenance.
- If the software is not easy to understand:
  - maintenance effort would increase many times.

Understandability

- Use consistent and meaningful names
  - for various design components,
- Design solution should consist of:
  - a cleanly decomposed set of modules (modularity),
- Different modules should be neatly arranged in a hierarchy:
  - in a neat tree-like diagram.

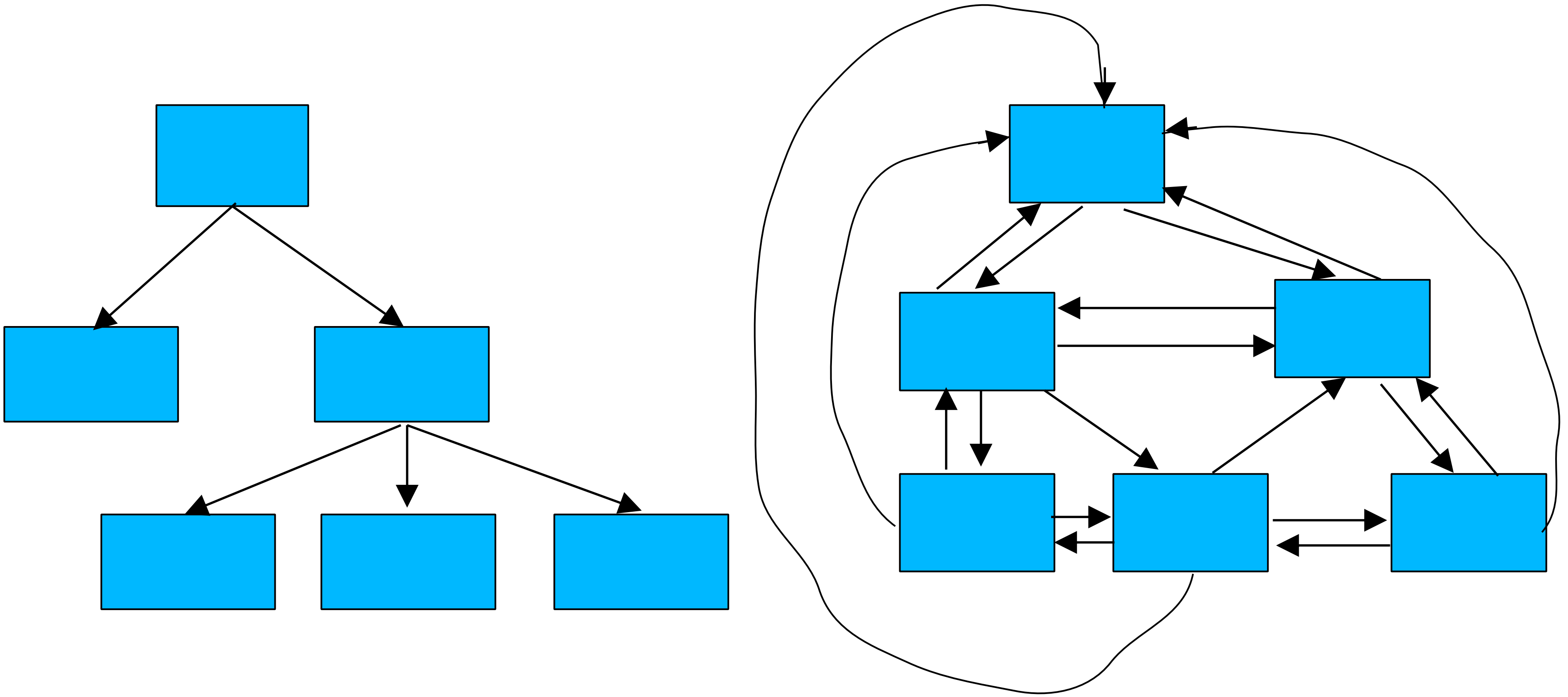
Modularity

- Modularity is a fundamental attributes of any good design.
  - Decomposition of a problem cleanly into modules:
  - Modules are almost independent of each other
  - divide and conquer principle.

Modularity

- If modules are independent:
  - modules can be understood separately,
    - reduces the complexity greatly.
  - To understand why this is so,
    - remember that it is very difficult to break a bunch of sticks but very easy to break the sticks individually.

Example of Cleanly and Non-cleanly Decomposed Modules



Modularity

- In technical terms, modules should display:
  - high cohesion
  - low coupling.
- We will shortly discuss:
  - cohesion and coupling.

Modularity

- Neat arrangement of modules in a hierarchy means:
  - low fan-out
  - abstraction

Cohesion and Coupling

- Cohesion is a measure of:
  - functional strength of a module.
  - A cohesive module performs a single task or function.
- Coupling between two modules:
  - a measure of the degree of interdependence or interaction between the two modules.

Cohesion and Coupling

- A module having high cohesion and low coupling:
  - functionally independent of other modules:
    - A functionally independent module has minimal interaction with other modules.

Advantages of Functional Independence

- Better understandability and good design:
- Complexity of design is reduced,
- Different modules easily understood in isolation:
  - modules are independent

Advantages of Functional Independence

- Functional independence reduces error propagation.
  - degree of interaction between modules is low.
  - an error existing in one module does not directly affect other modules.
- Reuse of modules is possible.

Advantages of Functional Independence

- A functionally independent module:
  - can be easily taken out and reused in a different program.
    - each module does some well-defined and precise function
    - the interfaces of a module with other modules is simple and minimal.

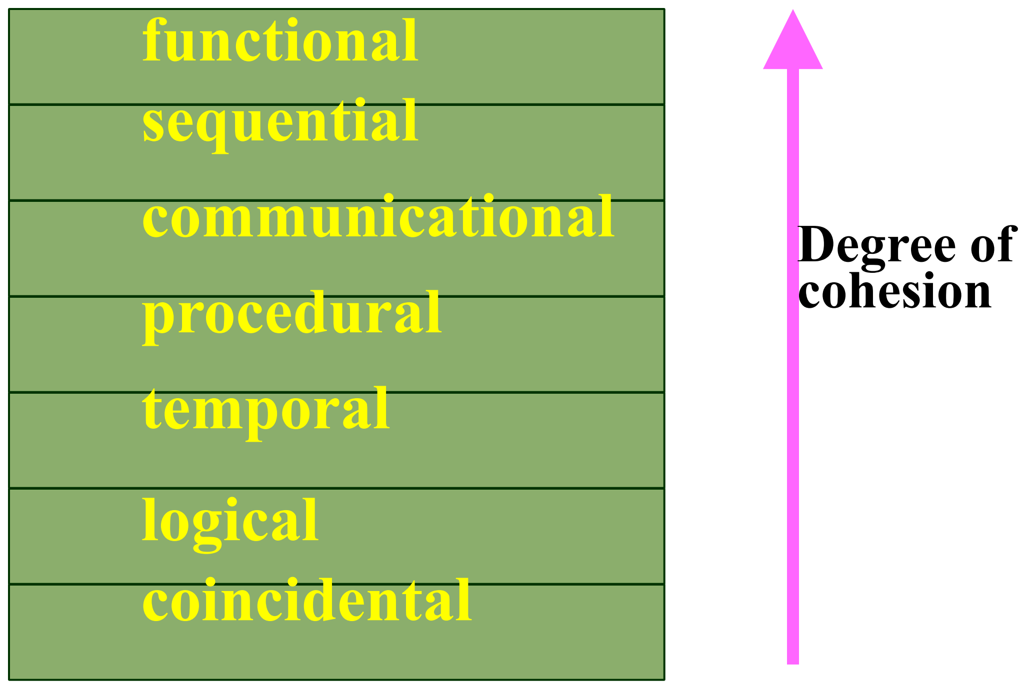
Functional Independence

- Unfortunately, there are no ways:
  - to quantitatively measure the degree of cohesion and coupling:
  - classification of different kinds of cohesion and coupling:
    - will give us some idea regarding the degree of cohesiveness of a module.

Classification of Cohesiveness

- Classification is often subjective:
  - yet gives us some idea about cohesiveness of a module.
- By examining the type of cohesion exhibited by a module:
  - we can roughly tell whether it displays high cohesion or low cohesion.

Classification of Cohesiveness



Coincidental cohesion

- The module performs a set of tasks:
  - which relate to each other very loosely, if at all.
    - the module contains a random collection of functions.
    - functions have been put in the module out of pure coincidence without any thought or design.

Logical cohesion

- All elements of the module perform similar operations:
  - e.g. error handling, data input, data output, etc.
- An example of logical cohesion:
  - a set of print functions to generate an output report arranged into a single module.

Temporal cohesion

- The module contains tasks that are related by the fact:
  - all the tasks must be executed in the same time span.
- Example:
  - The set of functions responsible for
    - initialization,
    - start-up, shut-down of some process, etc.

Procedural cohesion

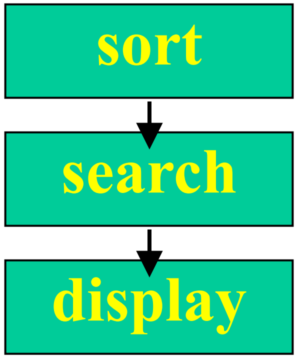
- The set of functions of the module:
  - all part of a procedure (algorithm)
  - certain sequence of steps have to be carried out in a certain order for achieving an objective,
    - e.g. the algorithm for decoding a message.

Communicational cohesion

- All functions of the module:
  - reference or update the same data structure,
- Example:
  - the set of functions defined on an array or a stack.

Sequential cohesion

- Elements of a module form different parts of a sequence,
  - output from one element of the sequence is input to the next.
  - Example:



Functional cohesion

- Different elements of a module cooperate:
  - to achieve a single function,
  - e.g. managing an employee's pay-roll.
- When a module displays functional cohesion,
  - we can describe the function using a single sentence.

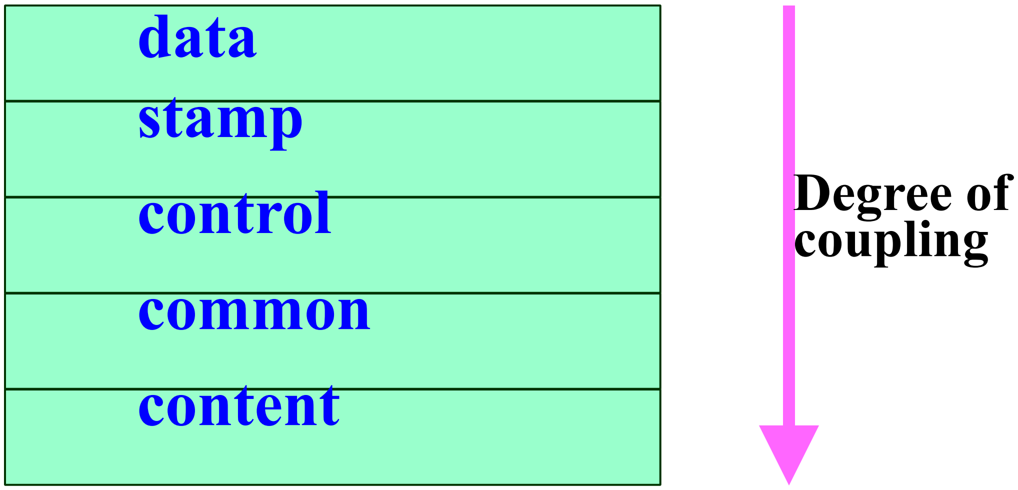
Coupling

- Coupling indicates:
  - how closely two modules interact or how interdependent they are.
  - The degree of coupling between two modules depends on their interface complexity.

Coupling

- There are no ways to precisely determine coupling between two modules:
  - classification of different types of coupling will help us to approximately estimate the degree of coupling between two modules.
- Five types of coupling can exist between any two modules.

Classes of coupling



Data coupling

- Two modules are data coupled,
  - if they communicate via a parameter:
    - an elementary data item,
    - e.g an integer, a float, a character, etc.
  - The data item should be problem related:
    - not used for control purpose.

Stamp coupling

- Two modules are <u>stamp coupled</u>,
  - if they communicate via a composite data item
    - such as a record in PASCAL
    - or a structure in C.

Control coupling

- Data from one module is used to direct
  - order of instruction execution in another.
- Example of control coupling:
  - a flag set in one module and tested in another module.

Common Coupling

- Two modules are <u>common coupled</u>,
  - if they share some global data.

Content coupling

- Content coupling exists between two modules:
  - if they share code,
  - e.g, branching from one module into another module.
- The degree of coupling increases
  - from data coupling to content coupling.

Neat Hierarchy

- Control hierarchy represents:
  - organization of modules.
  - control hierarchy is also called program structure.
- Most common notation:
  - a tree-like diagram called structure chart.

Neat Arrangement of modules

- Essentially means:
  - low fan-out
  - abstraction

Characteristics of Module Structure

- Depth:
  - number of levels of control
- Width:
  - overall span of control.
- Fan-out:
  - a measure of the number of modules directly controlled by given module.

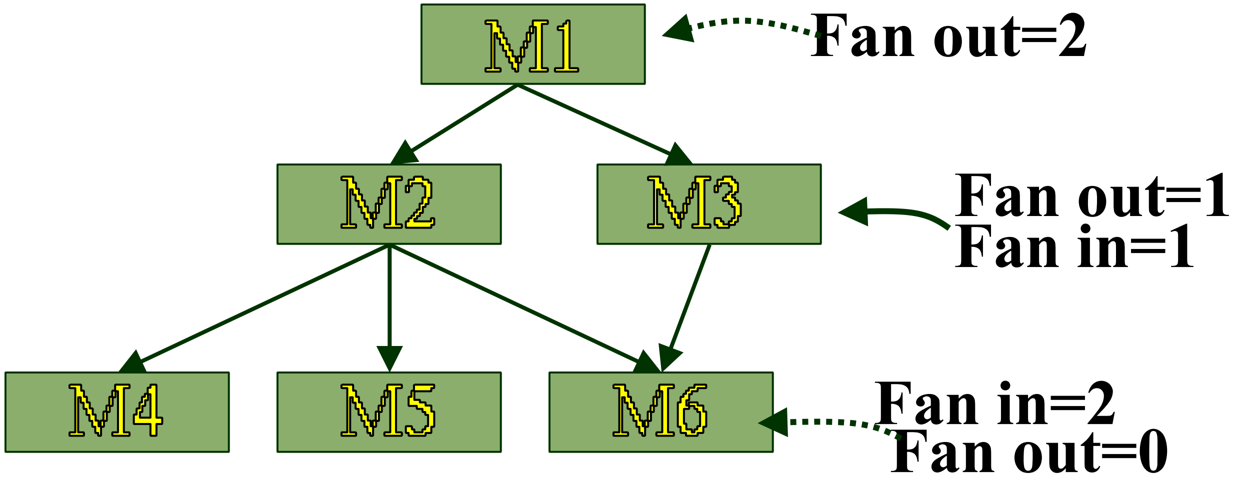
Characteristics of Module Structure

- Fan-in:
  - indicates how many modules directly invoke a given module.
  - High fan-in represents code reuse and is in general encouraged.

Characteristics of Module Structure

- Visibility
  - Module B is visible to module A if A directly calls B
- Control Abstraction
  - Modules at higher layer should not be visible to modules at lower layers

Module Structure



Goodness of Design

- A design having modules:
  - with high fan-out numbers is not a good design:
  - a module having high fan-out lacks cohesion.

Goodness of Design

- A module that invokes a large number of other modules:
  - likely to implement several different functions:
  - not likely to perform a single cohesive function.

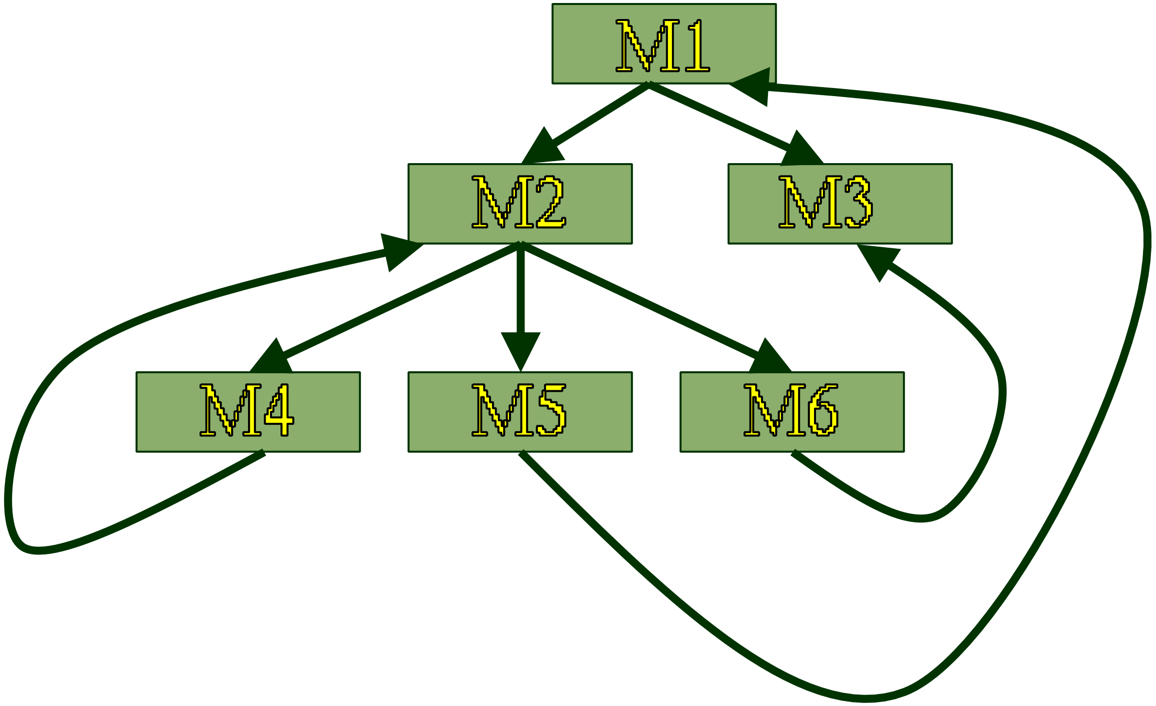
Control Relationships

- A module that controls another module:
  - said to be superordinate to it.
- Conversely, a module controlled by another module:
  - said to be subordinate to it.

Visibility and Layering

- A module A is said to be visible by another module B,
  - if A directly or indirectly calls B.
- The layering principle requires
  - modules at a layer can call only the modules immediately below it.

Bad Design



Abstraction

- Lower-level modules:
  - do input/output and other low-level functions.
- Upper-level modules:
  - do more managerial functions.

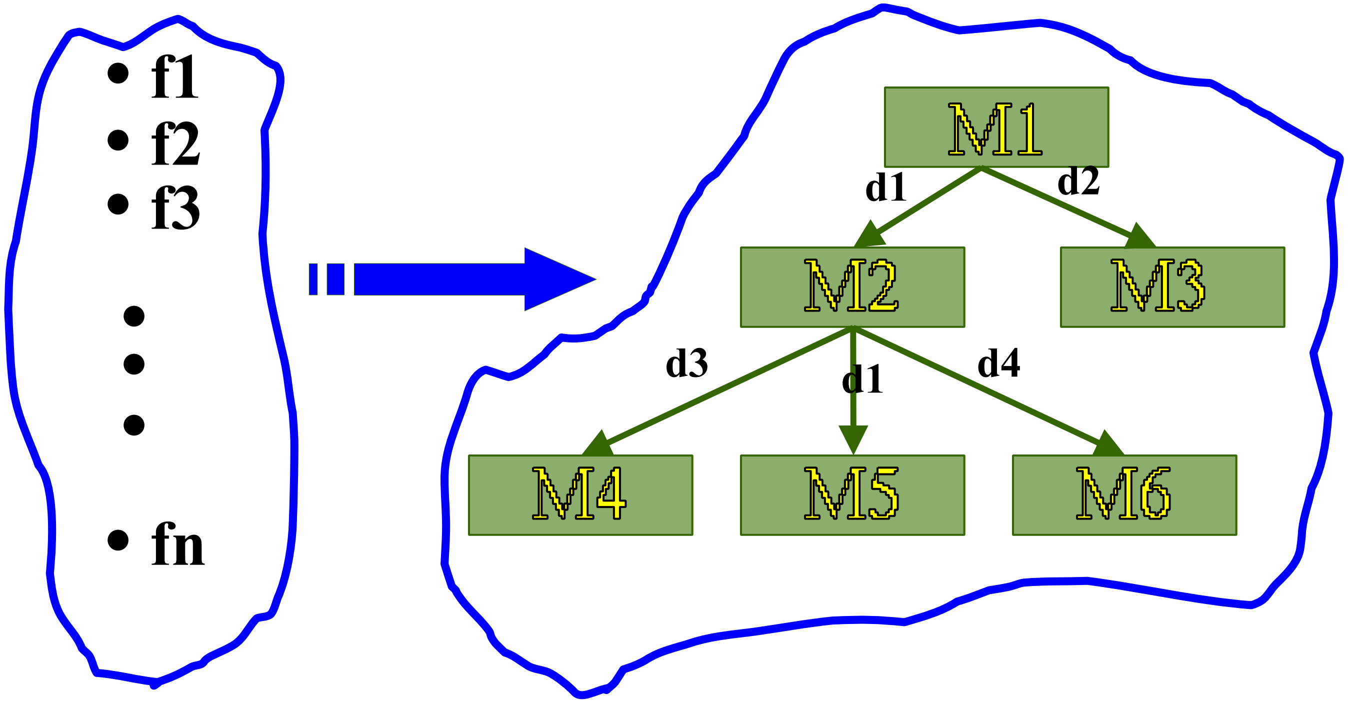
Abstraction

- The principle of abstraction requires:
  - lower-level modules do not invoke functions of higher level modules.
  - Also known as layered design.

High-level Design

- High-level design maps functions into modules such that:
  - Each module has high cohesion
  - Coupling among modules is as low as possible
  - Modules are organized in a neat hierarchy

High-level Design



Design Approaches

- Two fundamentally different software design approaches:
  - Function-oriented design
  - Object-oriented design

Design Approaches

- These two design approaches are radically different.
  - However, are complementary
    - rather than competing techniques.
  - Each technique is applicable at
    - different stages of the design process.

Function-Oriented Design

- A system is looked upon as something
  - that performs a set of functions.
- Starting at this high-level view of the system:
  - each function is successively refined into more detailed functions.
  - Functions are mapped to a module structure.

Example

- The function create-new-library-member:
  - creates the record for a new member,
  - assigns a unique membership number
  - prints a bill towards the membership

Example

- Create-library-member function consists of the following sub-functions:
  - assign-membership-number
  - create-member-record
  - print-bill

Function-Oriented Design

- Each subfunction:
  - split into more detailed subfunctions and so on.

Function-Oriented Design

- The system state is centralized:
  - accessible to different functions,
  - member-records:
    - available for reference and updation to several functions:
      - create-new-member
      - delete-member
      - update-member-record

Object-Oriented Design

- System is viewed as a collection of objects (i.e. entities).
- System state is decentralized among the objects:
  - each object manages its own state information.

Object-Oriented Design Example

- Library Automation Software:
  - each library member is a separate object
    - with its own data and functions.
  - Functions defined for one object:
    - cannot directly refer to or change data of other objects.

Object-Oriented Design

- Objects have their own internal data:
  - defines their state.
- Similar objects constitute a class.
  - each object is a member of some class.
- Classes may inherit features
  - from a super class.
- Conceptually, objects communicate by message passing.

Object-Oriented Design

- Data Abstraction
- Data Structure
- Data Type

Object-Oriented versus Function-Oriented Design

- Unlike function-oriented design,
  - in OOD the basic abstraction is not functions such as "sort", "display", "track", etc.,
  - but real-world entities such as "employee", "picture", "machine", "radar system", etc.

Object-Oriented versus Function-Oriented Design

- In OOD:
  - software is not developed by designing functions such as:
    - update-employee-record,
    - get-employee-address, etc.
  - but by designing objects such as:
    - employees,
    - departments, etc.

Object-Oriented versus Function-Oriented Design

- Grady Booch sums up this fundamental difference saying:
  - “Identify verbs if you are after procedural design and nouns if you are after object-oriented design.”

Object-Oriented versus Function-Oriented Design

- In OOD:
  - state information is not shared in a centralized data.
  - but is distributed among the objects of the system.

Example:

- In an employee pay-roll system, the following can be global data:
  - names of the employees,
  - their code numbers,
  - basic salaries, etc.
- Whereas, in object oriented systems:
  - data is distributed among different employee objects of the system.

Object-Oriented versus Function-Oriented Design

- Objects communicate by message passing.
  - one object may discover the state information of another object by interrogating it.

Object-Oriented versus Function-Oriented Design

- Of course, somewhere or other the functions must be implemented:
  - the functions are usually associated with specific real-world entities (objects)
  - directly access only part of the system state information.

Object-Oriented versus Function-Oriented Design

- Function-oriented techniques group functions together if:
  - as a group, they constitute a higher level function.
- On the other hand, object-oriented techniques group functions together:
  - on the basis of the data they operate on.

Object-Oriented versus Function-Oriented Design

- To illustrate the differences between object-oriented and function-oriented design approaches,
  - let us consider an example ---
  - An automated fire-alarm system for a large building.

Fire-Alarm System:

- We need to develop a computerized fire alarm system for a large multi-storied building:
  - There are 80 floors and 1000 rooms in the building.

Fire-Alarm System:

- Different rooms of the building:
  - fitted with smoke detectors and fire alarms.
- The fire alarm system would monitor:
  - status of the smoke detectors.

Fire-Alarm System

- Whenever a fire condition is reported by any smoke detector:
  - the fire alarm system should:
    - determine the location from which the fire condition was reported
    - sound the alarms in the neighboring locations.

Fire-Alarm System

- The fire alarm system should:
  - flash an alarm message on the computer console:
    - fire fighting personnel man the console round the clock.

Fire-Alarm System

- After a fire condition has been successfully handled,
  - the fire alarm system should let fire fighting personnel reset the alarms.

Function-Oriented Approach:

- /* Global data (system state) accessible by various functions */

```
BOOL  detector_status[1000];
int   detector_locs[1000];
BOOL  alarm-status[1000]; /* alarm activated when status set */
int   alarm_locs[1000]; /* room number where alarm is located */
int   neighbor-alarms[1000][10];/*each detector has at most*/
                    /* 10 neighboring alarm locations */
```

The functions which operate on the system state:

```
 interrogate_detectors();
 get_detector_location();
 determine_neighbor();
 ring_alarm();
 reset_alarm();
 report_fire_location();
```

Object-Oriented Approach:

- class detector
  - attributes: status, location, neighbors
  - operations: create, sense-status, get-location,
  - find-neighbors
- class alarm
  - attributes: location, status
  - operations: create, ring-alarm, get_location,
  - reset-alarm
- In the object oriented program,
  - appropriate number of instances of the class detector and alarm should be created.

Object-Oriented versus Function-Oriented Design

- In the function-oriented program :
  - the system state is centralized
  - several functions accessing these data are defined.
- In the object oriented program,
  - the state information is distributed among various sensor and alarm objects.

Object-Oriented versus Function-Oriented Design

- Use OOD to design the classes:
  - then applies top-down function oriented techniques
    - to design the internal methods of classes.

Object-Oriented versus Function-Oriented Design

- Though outwardly a system may appear to have been developed in an object oriented fashion,
  - but inside each class there is a small hierarchy of functions designed in a top-down manner.

Summary

- We started with an overview of:
  - activities undertaken during the software design phase.
- We identified:
  - the information need to be produced at the end of the design phase:
    - so that the design can be easily implemented using a programming language.

Summary

- We characterized the features of a good software design by introducing the concepts of:
  - fan-in, fan-out,
  - cohesion, coupling,
  - abstraction, etc.

Summary

- We classified different types of cohesion and coupling:
  - enables us to approximately determine the cohesion and coupling existing in a design.

Summary

- Two fundamentally different approaches to software design:
  - function-oriented approach
  - object-oriented approach

Summary

- We looked at the essential philosophy behind these two approaches
  - these two approaches are not competing but complementary approaches.

Function-Oriented Software Design

Organization of this Lecture

- Brief review of last lecture
- Introduction to function-oriented design
- Structured Analysis and Structured Design
- Data flow diagrams (DFDs)
  - A major objective of this lecture is that you should be able to develop DFD model for any problem.
- Examples
- Summary

Review of last lecture

- Last lecture we started
  - with an overview of activities carried out during the software design phase.
- We identified different information that must be produced at the end of the design phase:
  - so that the design can be easily implemented using a programming language.

Review of last lecture

- We characterized the features of a good software design by introducing the concepts:
  - cohesion, coupling,
  - fan-in, fan-out,
  - abstraction, etc.
- We classified different types of cohesion and coupling:
  - enables us to approximately determine the cohesion and coupling existing in a design.

Review of last lecture

- There are two fundamentally different approaches to software design:
  - function-oriented approach
  - object-oriented approach
- We looked at the essential philosophy of these two approaches:
  - the approaches are not competing but complementary approaches.

Introduction

- Function-oriented design techniques are very popular:
  - currently in use in many software development organizations.
- Function-oriented design techniques:
  - start with the functional requirements specified in the SRS document.

Introduction

- During the design process:
  - high-level functions are successively decomposed:
    - into more detailed functions.
  - finally the detailed functions are mapped to a module structure.

Introduction

- Successive decomposition of high-level functions:
  - into more detailed functions.
  - Technically known as top-down decomposition.

SA/SD (Structured Analysis/Structured Design)

- SA/SD methodology:
  - has essential features of several important function-oriented design methodologies ---
    - if you need to use any specific design methodology later on,
    - you can do so easily with small additional effort.

Overview of SA/SD Methodology

- SA/SD methodology consists of two distinct activities:
  - Structured Analysis (SA)
  - Structured Design (SD)
- During structured analysis:
  - functional decomposition takes place.
- During structured design:
  - module structure is formalized.

Functional decomposition

- Each function is analyzed:
  - hierarchically decomposed into more detailed functions.
  - simultaneous decomposition of high-level data
    - into more detailed data.

Structured analysis

- Transforms a textual problem description into a graphic model.
  - done using data flow diagrams (DFDs).
  - DFDs graphically represent the results of structured analysis.

Structured design

- All the functions represented in the DFD:
  - mapped to a module structure.
- The module structure:
  - also called as the <u>software architecture:</u>

Detailed Design

- Software architecture:
  - refined through detailed design.
  - Detailed design can be directly implemented:
    - using a conventional programming language.

Structured Analysis vs. Structured Design

- **Purpose of structured analysis:**
  - capture the detailed structure of the system as the user views it.
- **Purpose of structured design:**
  - arrive at a form that is suitable for implementation in some programming language.

Structured Analysis vs. Structured Design

- The results of structured analysis can be easily understood even by ordinary customers:
  - does not require computer knowledge
  - directly represents customer's perception of the problem
  - uses customer's terminology for naming different functions and data.
- The results of structured analysis can be reviewed by customers:
  - to check whether it captures all their requirements.

Structured Analysis

- Based on principles of:
  - Top-down decomposition approach.
  - Divide and conquer principle:
    - each function is considered individually (i.e. isolated from other functions)
    - decompose functions totally disregarding what happens in other functions.
  - Graphical representation of results using
    - data flow diagrams (or bubble charts).

Data flow diagrams

- DFD is an elegant modelling technique:
  - useful not only to represent the results of structured analysis
  - applicable to other areas also:
    - e.g. for showing the flow of documents or items in an organization,
- DFD technique is very popular because
  - it is simple to understand and use.

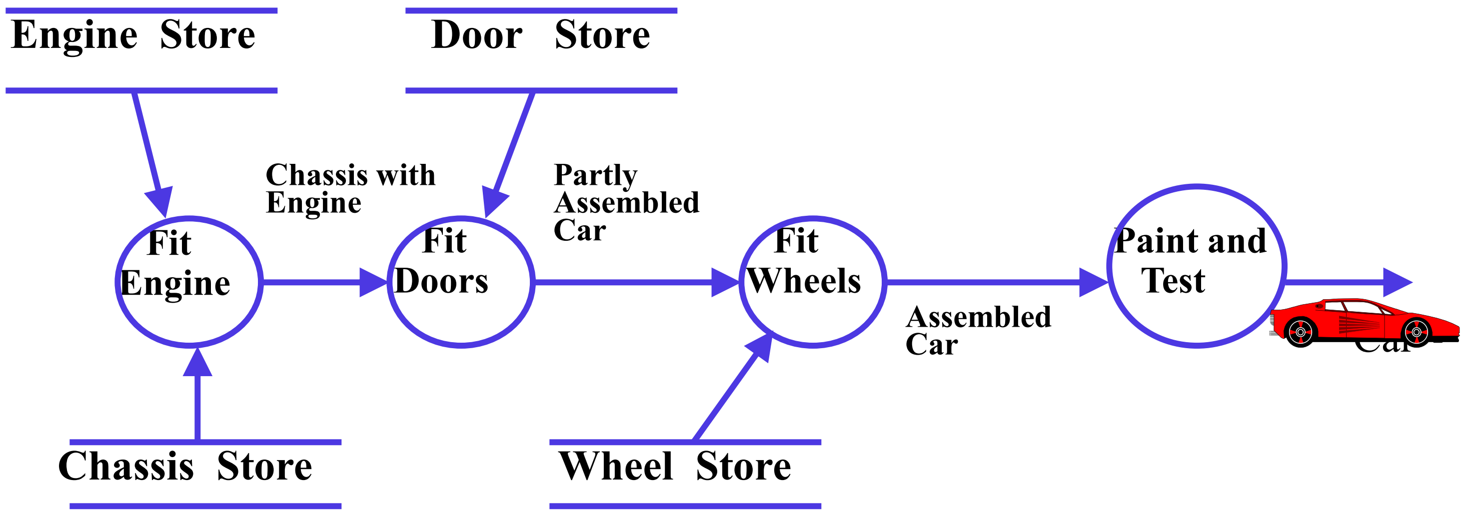
Data flow diagram

- DFD is a hierarchical graphical model:
  - shows the different functions (or processes) of the system and
  - data interchange among the processes.

DFD Concepts

- It is useful to consider each function as a processing station:
  - each function consumes some input data and
  - produces some output data.

Data Flow Model of a Car Assembly Unit



Data Flow Diagrams (DFDs)

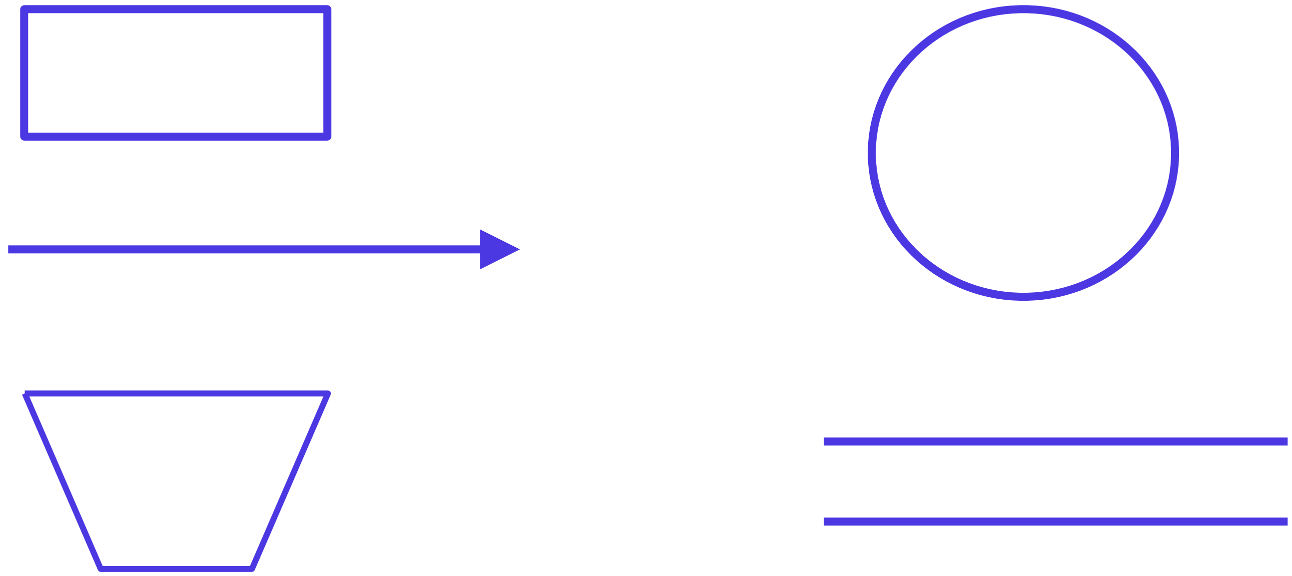
- A DFD model:
  - uses limited types of symbols.
  - simple set of rules
  - easy to understand:
    - it is a hierarchical model.

Hierarchical model

- Human mind can easily understand any hierarchical model:
  - in a hierarchical model:
    - we start with a very simple and abstract model of a system,
    - details are slowly introduced through the hierarchies.

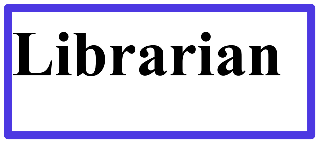
Data Flow Diagrams (DFDs)

- **Primitive Symbols Used for Constructing DFDs:**



External Entity Symbol

- Represented by a rectangle
- External entities are real physical entities:



  - input data to the system or
  - consume data produced by the system.
  - Sometimes external entities are called **terminator, source, or sink.**

Function Symbol

- A function such as "search-book" is represented using a circle:

  - This symbol is called a <u>process</u> or <u>bubble</u> or <u>transform</u>.
  - Bubbles are annotated with corresponding function names.
  - Functions represent some activity:
    - function names should be verbs.

Data Flow Symbol

- A directed arc or line.

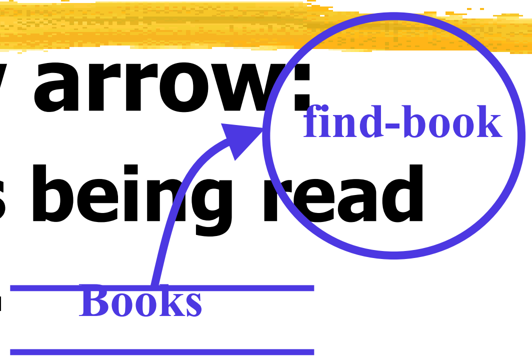
  - represents data flow in the direction of the arrow.
  - Data flow symbols are annotated with names of data they carry.

Data Store Symbol

- Represents a logical file:
  - A logical file can be:
    - a data structure
    - a physical file on disk.
  - Each data store is connected to a process:
    - by means of a data flow symbol.

Data Store Symbol

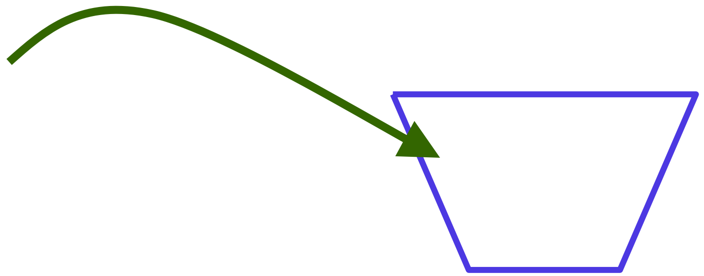
- Direction of data flow arrow:
  - shows whether data is being read from or written into it.



- An arrow into or out of a **data store:**
  - implicitly represents the entire data of the data store
  - arrows connecting to a data store need not be annotated with any data name.

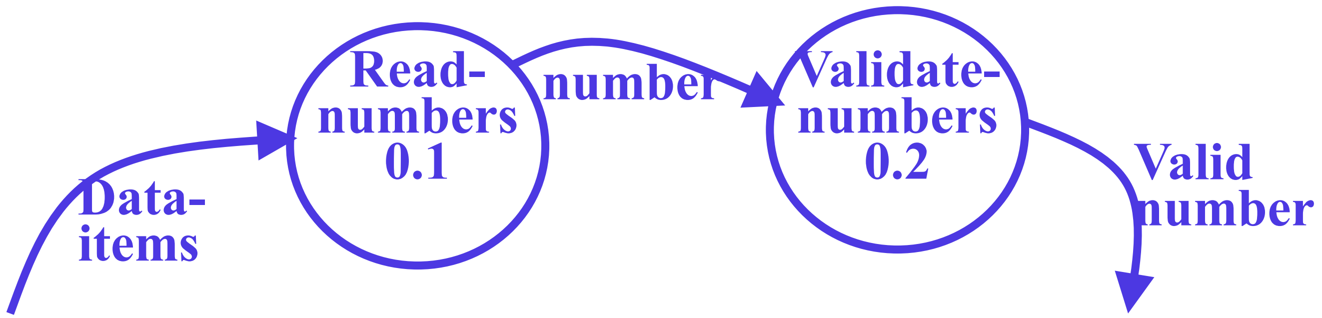
Output Symbol

- Output produced by the system



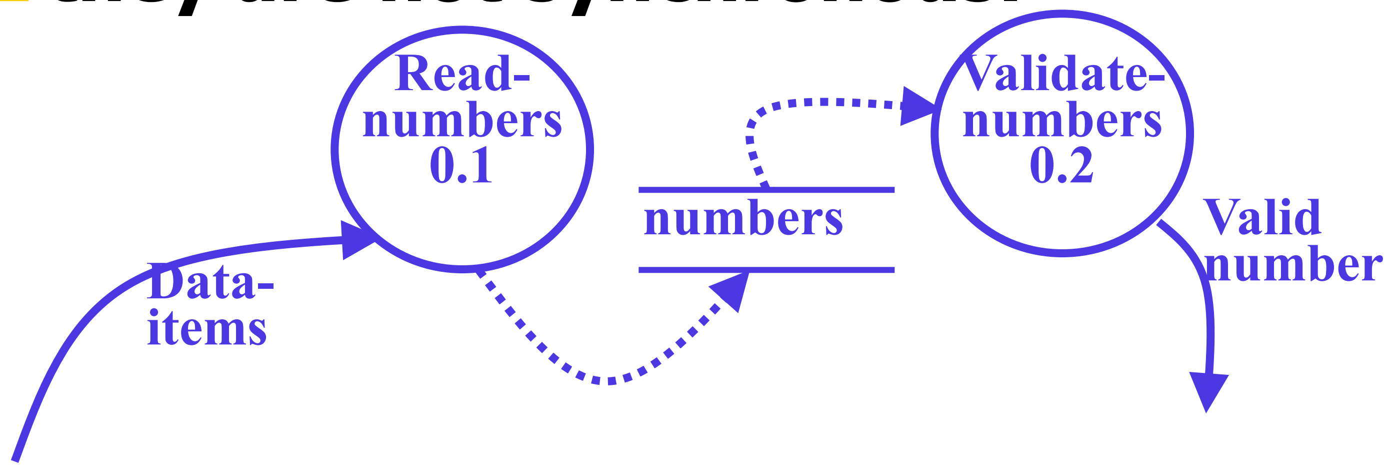
Synchronous operation

- If two bubbles are directly connected by a data flow arrow:
  - they are synchronous



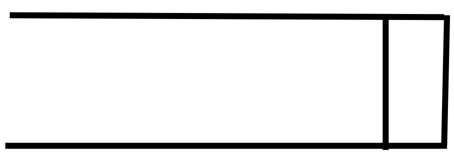
Asynchronous operation

- If two bubbles are connected via a data store:
  - they are not synchronous.



Yourdon's vs. Gane Sarson Notations

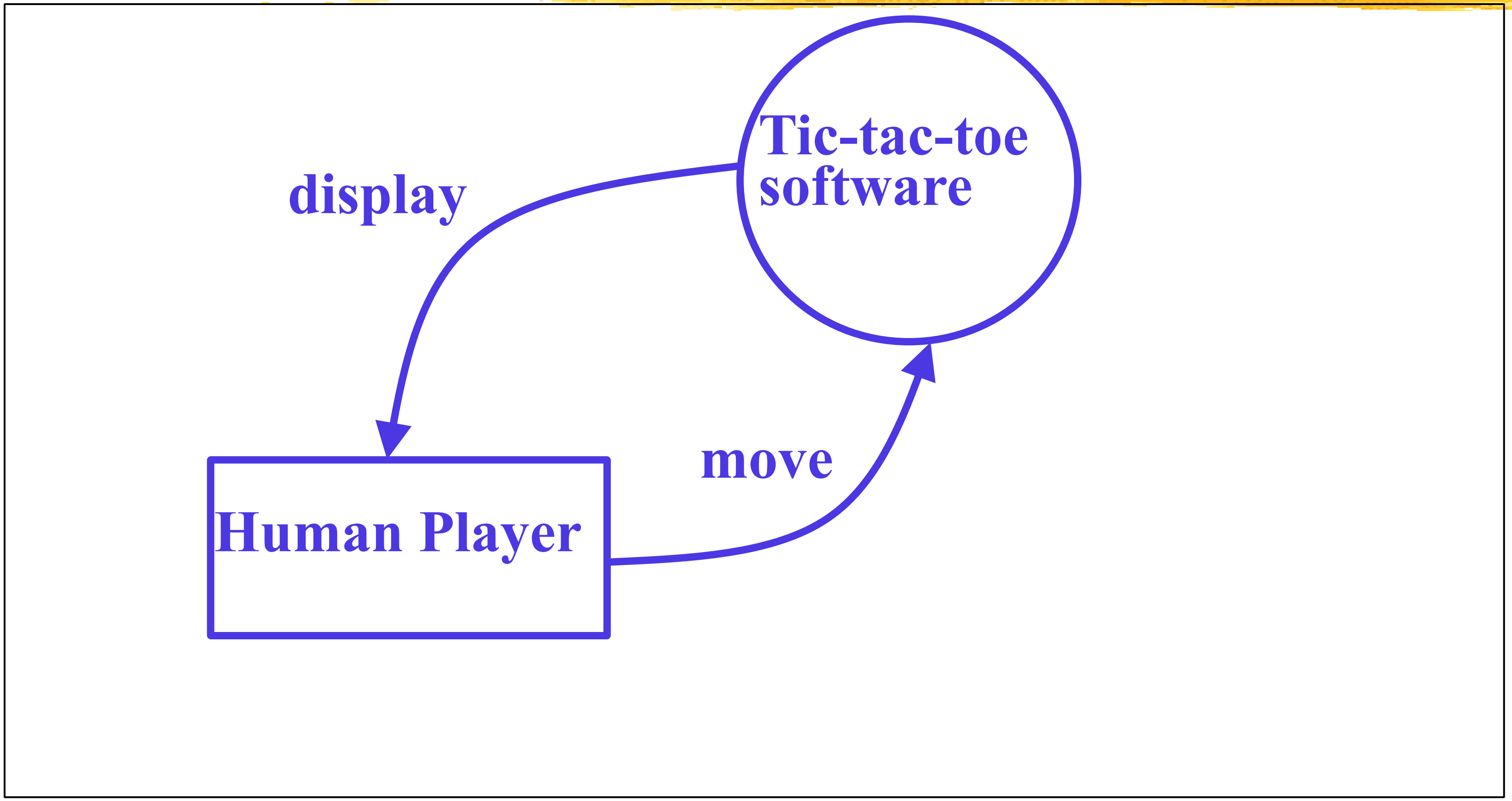
- The notations that we would be following are closer to the Yourdon's notations
- You may sometimes find notations in books that are slightly different
  - For example, the data store may look like a box with one end closed



How is Structured Analysis Performed?

- Initially represent the software at the most abstract level:
  - called the context diagram.
  - the entire system is represented as a single bubble,
  - this bubble is labelled according to the main function of the system.

Tic-tac-toe: Context Diagram



Context Diagram

- A context diagram shows:
  - data input to the system,
  - output data generated by the system,
  - external entities.

Context Diagram

- Context diagram captures:
  - various entities external to the system and interacting with it.
  - data flow occurring between the system and the external entities.
- The context diagram is also called as the level 0 DFD.

Context Diagram

- Context diagram
  - establishes the context of the system, i.e.
  - represents:
    - Data sources
    - Data sinks.

Level 1 DFD

- Examine the SRS document:
  - Represent each high-level function as a bubble.
  - Represent data input to every high-level function.
  - Represent data output from every high-level function.

Higher level DFDs

- Each high-level function is separately decomposed into subfunctions:
  - identify the subfunctions of the function
  - identify the data input to each subfunction
  - identify the data output from each subfunction
- These are represented as DFDs.

Decomposition

- Decomposition of a bubble:
  - also called **factoring** or **exploding**.
- Each bubble is decomposed to
  - between 3 to 7 bubbles.

Decomposition

- Too few bubbles make decomposition superfluous:
  - if a bubble is decomposed to just one or two bubbles:
    - then this decomposition is redundant.

Decomposition

- Too many bubbles:
  - more than 7 bubbles at any level of a DFD
  - make the DFD model hard to understand.

Decompose how long?

- Decomposition of a bubble should be carried on until:
  - a level at which the function of the bubble can be described using a simple algorithm.

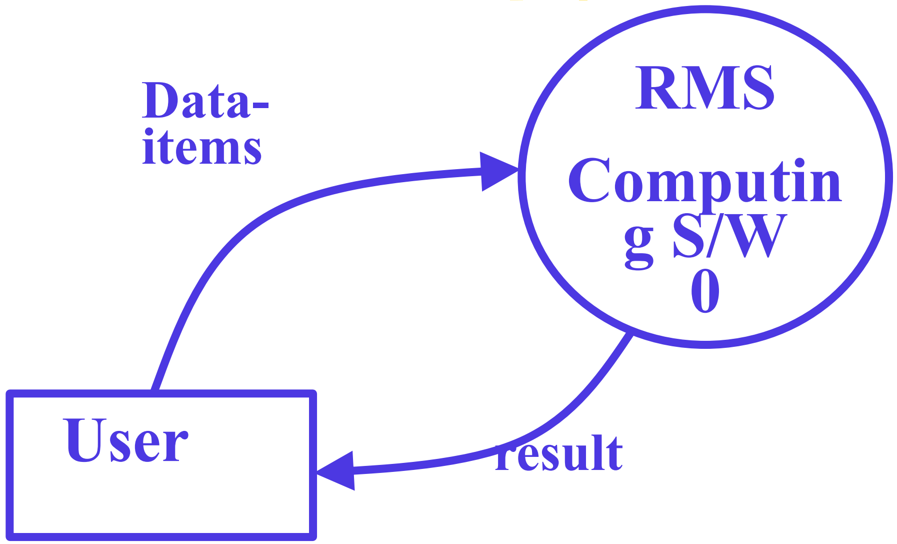
Example 1: RMS Calculating Software

- Consider a software called RMS calculating software:
  - reads three integers in the range of -1000 and +1000
  - finds out the root mean square (rms) of the three input numbers
  - displays the result.

Example 1: RMS Calculating Software

- The context diagram is simple to develop:
  - The system accepts 3 integers from the user
  - returns the result to him.

Example 1: RMS Calculating Software



Context Diagram

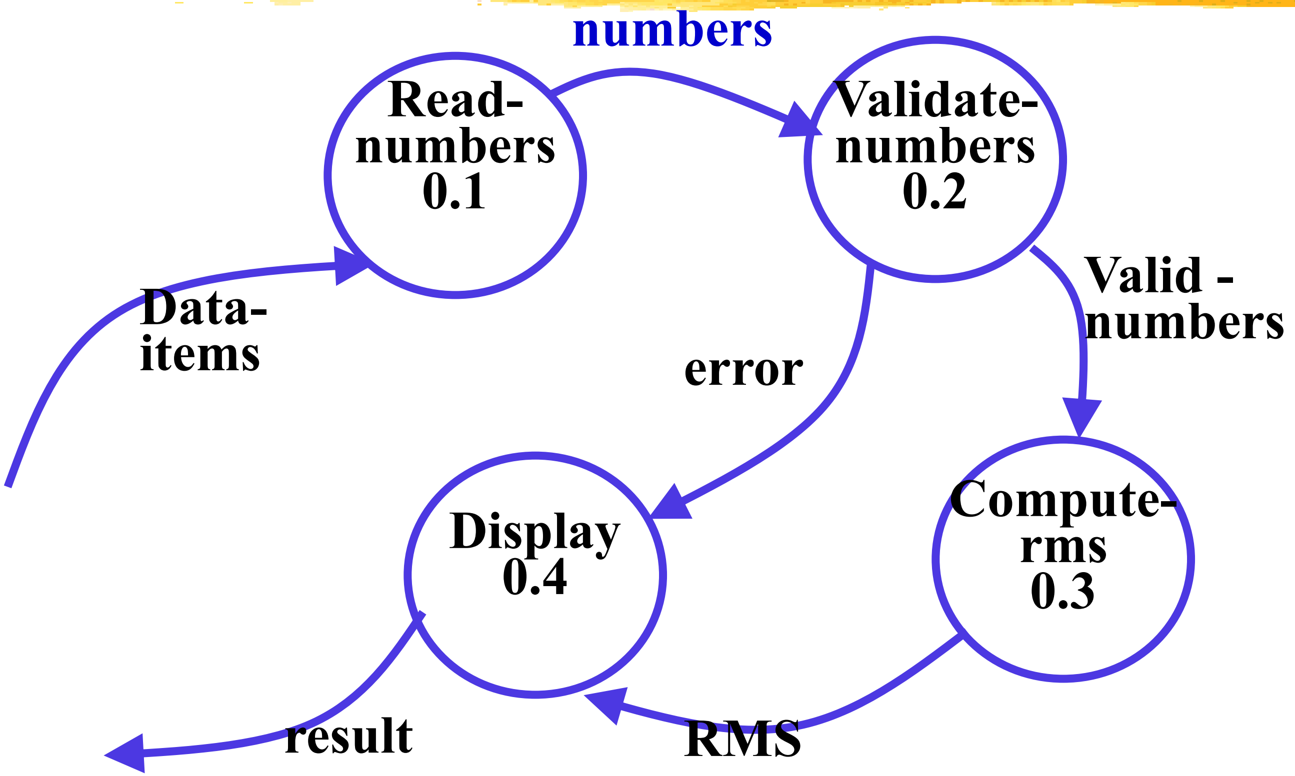
Example 1: RMS Calculating Software

- From a cursory analysis of the problem description:
  - we can see that the system needs to perform several things.

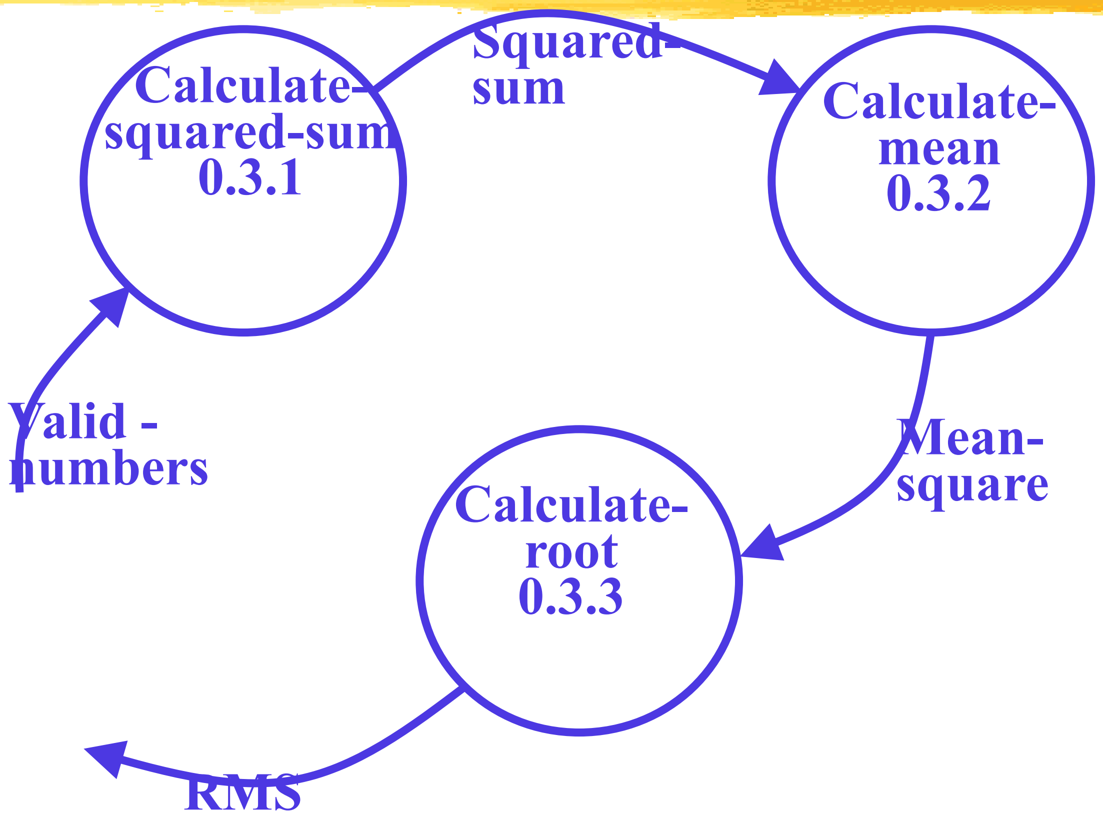
Example 1: RMS Calculating Software

- Accept input numbers from the user:
  - validate the numbers,
  - calculate the root mean square of the input numbers
  - display the result.

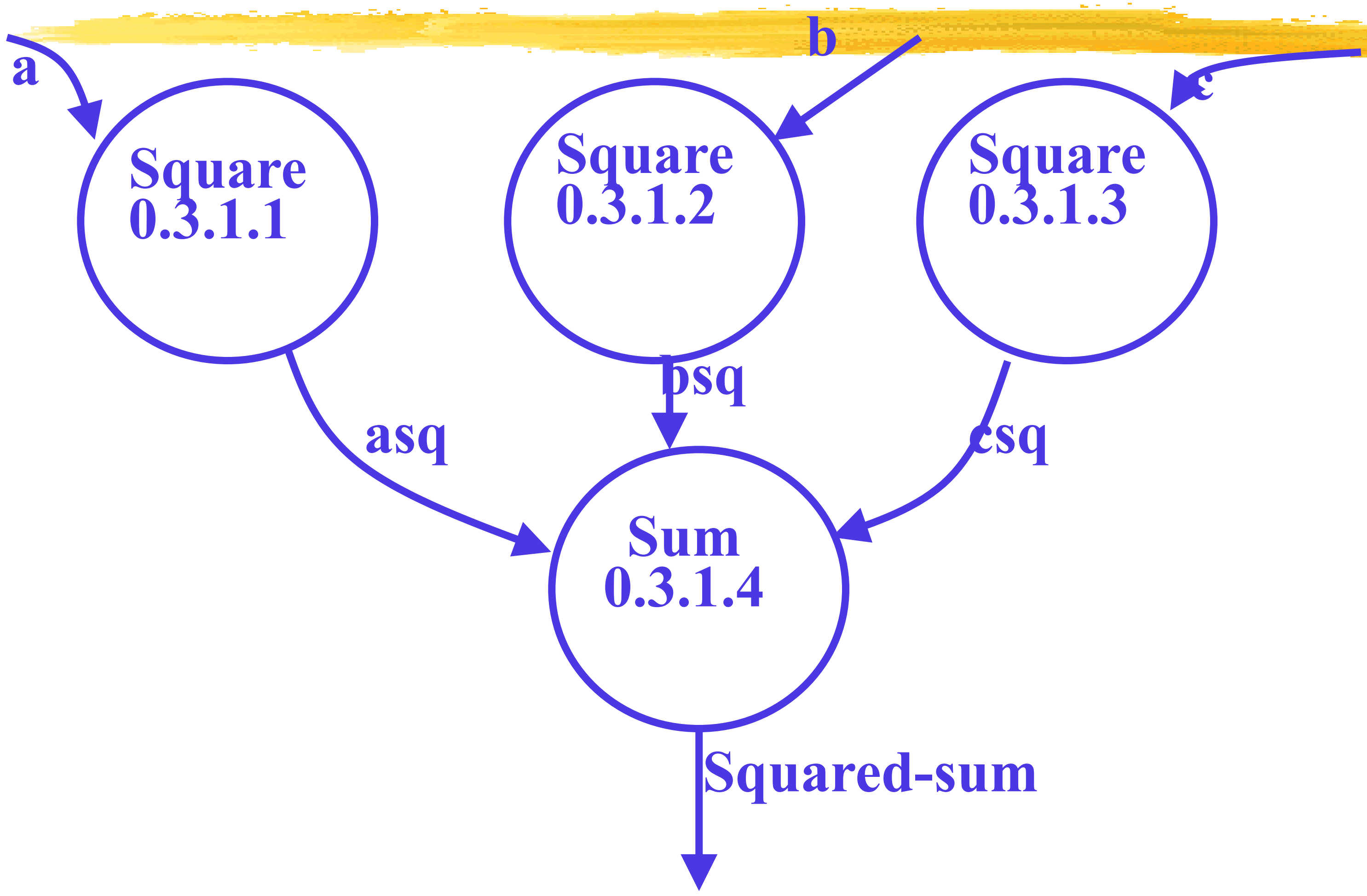
Example 1: RMS Calculating Software



Example 1: RMS Calculating Software



Example: RMS Calculating Software



Example: RMS Calculating Software

- Decomposition is never carried on up to basic instruction level:
  - a bubble is not decomposed any further:
    - if it can be represented by a simple set of instructions.

Data Dictionary

- A DFD is always accompanied by a data dictionary.
- A data dictionary lists all data items appearing in a DFD:
  - definition of all composite data items in terms of their component data items.
  - all data names along with the purpose of data items.
- For example, a data dictionary entry may be:
  - grossPay = regularPay+overtimePay

Importance of Data Dictionary

- Provides all engineers in a project with standard terminology for all data:
  - A consistent vocabulary for data is very important
  - different engineers tend to use different terms to refer to the same data,
    - causes unnecessary confusion.

Importance of Data Dictionary

- Data dictionary provides the definition of different data:
  - in terms of their component elements.
- For large systems,
  - the data dictionary grows rapidly in size and complexity.
  - Typical projects can have thousands of data dictionary entries.
  - It is extremely difficult to maintain such a dictionary manually.

Data Dictionary

- CASE (Computer Aided Software Engineering) tools come handy:
  - CASE tools capture the data items appearing in a DFD automatically to generate the data dictionary.

Data Dictionary

- CASE tools support queries:
  - about definition and usage of data items.
- For example, queries may be made to find:
  - which data item affects which processes,
  - a process affects which data items,
  - the definition and usage of specific data items, etc.
- Query handling is facilitated:
  - if data dictionary is stored in a relational database management system (RDBMS).

Data Definition

- Composite data are defined in terms of primitive data items using following operators:
- +: denotes composition of data items, e.g
  - a+b represents data a and b.
- [,,,]: represents selection,
  - i.e. any one of the data items listed inside the square bracket can occur.
  - For example, [a,b] represents either a occurs or b occurs.

Data Definition

- ( ): contents inside the bracket represent optional data
  - which may or may not appear.
  - a+(b) represents either a or a+b occurs.
- {}: represents iterative data definition,
  - e.g. {name}5 represents five name data.

Data Definition

- {name}* represents
  - zero or more instances of name data.
- = represents equivalence,
  - e.g. a=b+c means that a represents b and c.
- /* */: Anything appearing within /* */ is considered as comment.

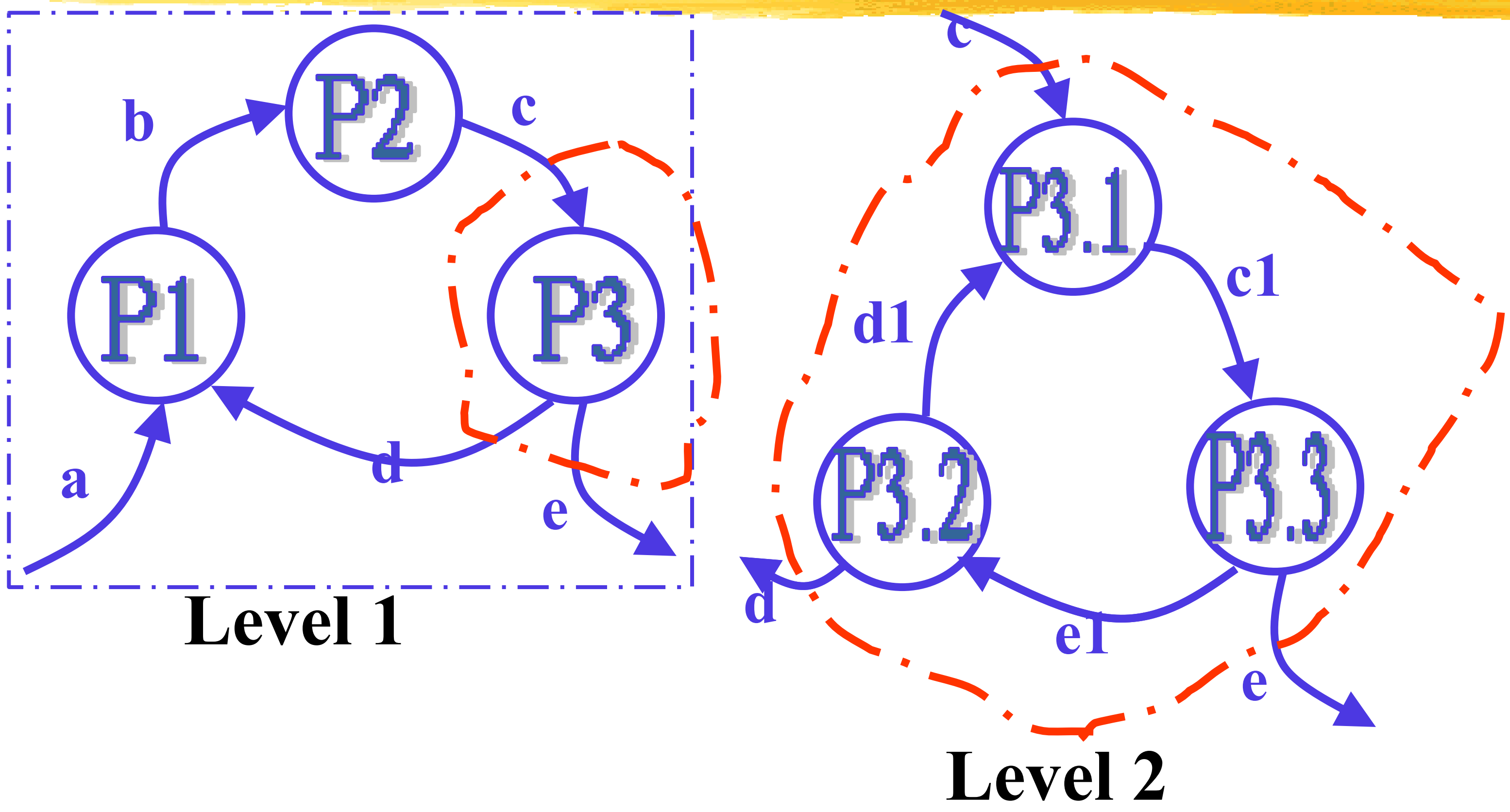
Data dictionary for RMS Software

- numbers=valid-numbers=a+b+c
- a:integer * input number *
- b:integer * input number *
- c:integer * input number *
- asq:integer
- bsq:integer
- csq:integer
- squared-sum: integer
- Result=[RMS,error]
- RMS: integer * root mean square value*
- error:string * error message*

Balancing a DFD

- Data flowing into or out of a bubble:
  - must match the data flows at the next level of DFD.
  - This is known as <u>balancing a DFD</u>
- In the level 1 of the DFD,
  - data item c flows into the bubble P3 and the data item d and e flow out.
- In the next level, bubble P3 is decomposed.
  - The decomposition is balanced as data item c flows into the level 2 diagram and d and e flow out.

Balancing a DFD



Numbering of Bubbles:

- Number the bubbles in a DFD:
  - numbers help in uniquely identifying any bubble from its bubble number.
- The bubble at context level:
  - assigned number 0.
- Bubbles at level 1:
  - numbered 0.1, 0.2, 0.3, etc
- When a bubble numbered x is decomposed,
  - its children bubble are numbered x.1, x.2, x.3, etc.

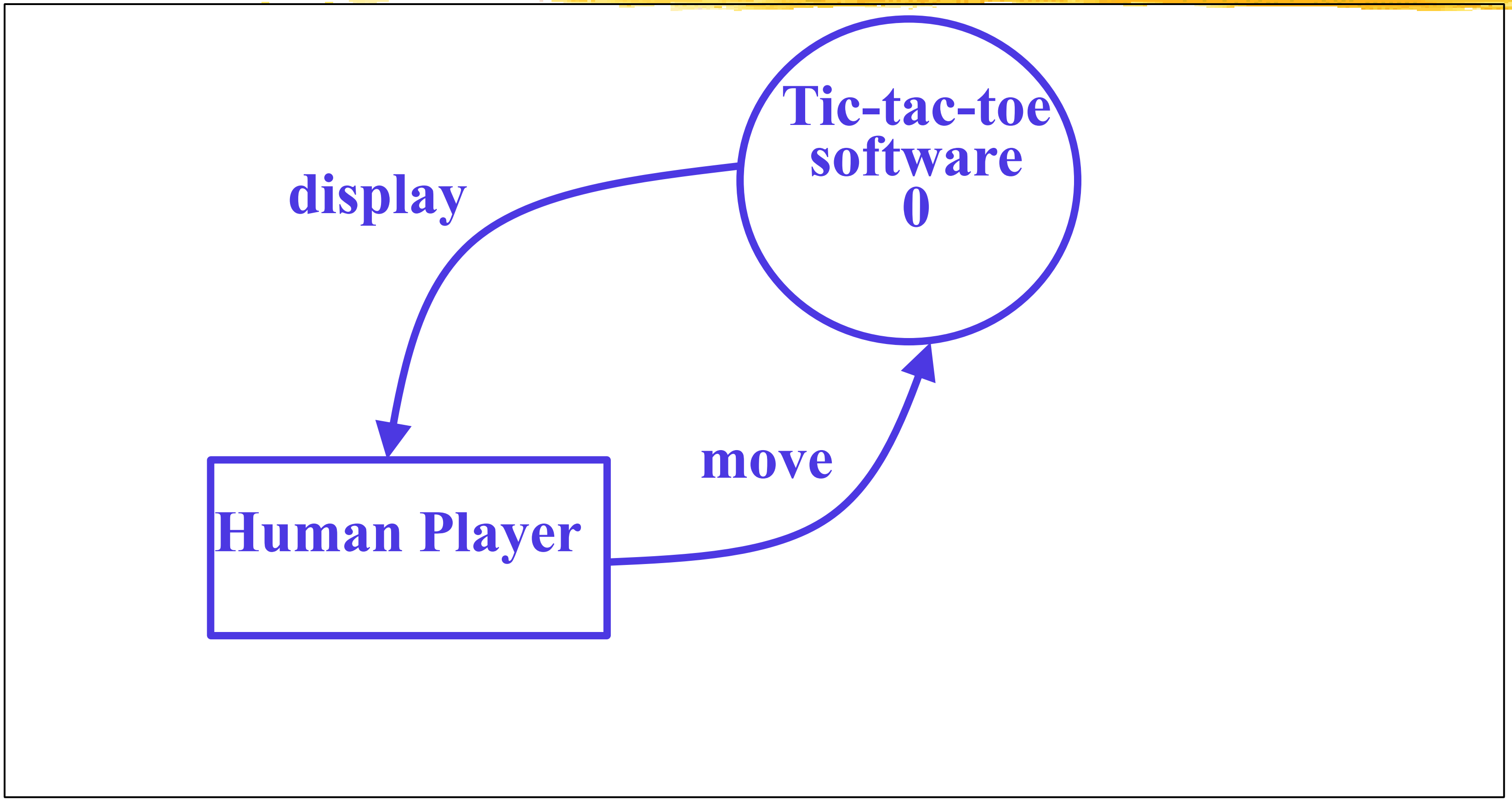
Example 2: Tic-Tac-Toe Computer Game

- A human player and the computer make alternate moves on a 3X3 square.
- A move consists of marking a previously unmarked square.
- The user marks a square
- Whoever is first to place three consecutive marks along a straight line (i.e., along a row, column, or diagonal) on the square wins.

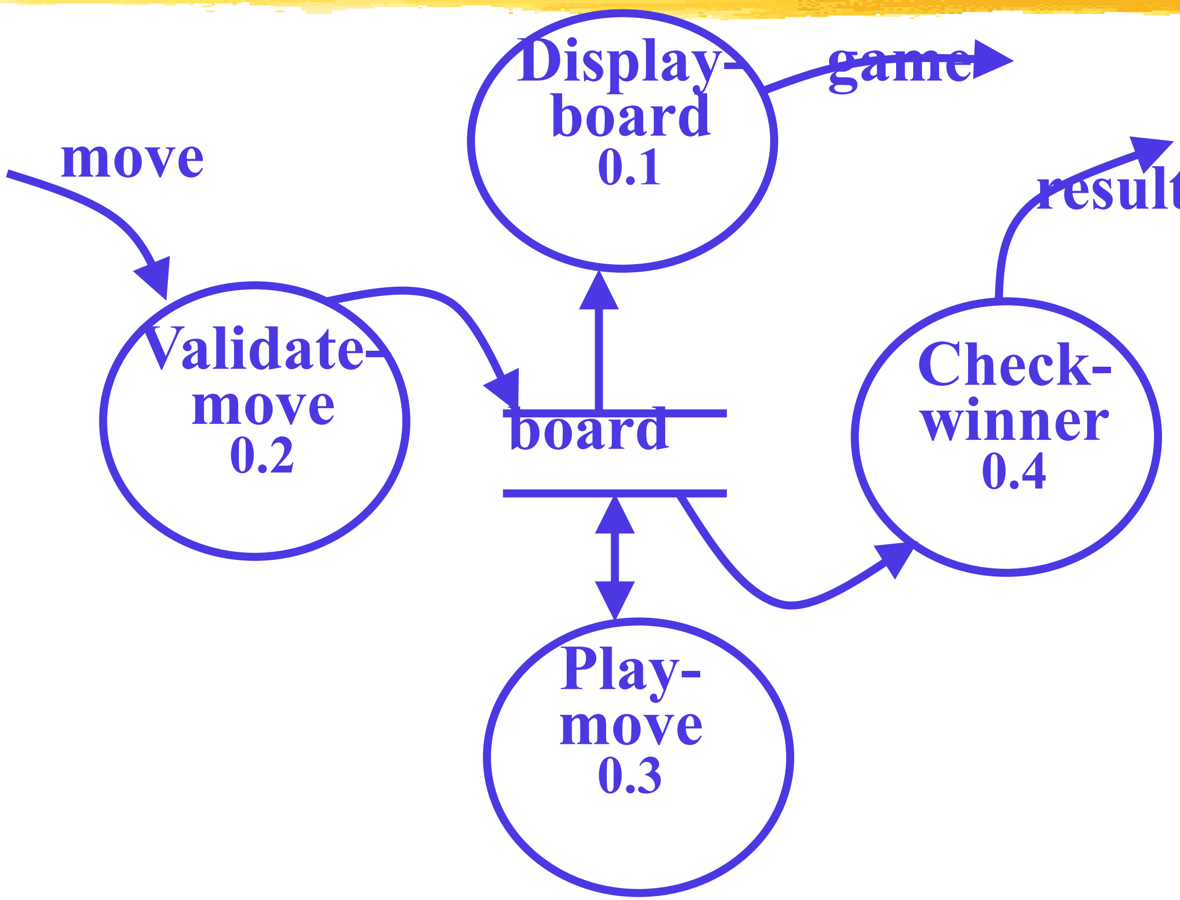
Example: Tic-Tac-Toe Computer Game

- As soon as either of the human player or the computer wins,
  - a message announcing the winner should be displayed.
- If neither player manages to get three consecutive marks along a straight line,
  - and all the squares on the board are filled up,
  - then the game is drawn.
- The computer always tries to win a game.

Context Diagram for Example



Level 1 DFD



Display-
board
0.1
game
move
result
Validate-
move
0.2
board
Check-
winner
0.4
Play-
move
0.3

Data dictionary

- Display=game + result
- move = integer
- board = {integer}9
- game = {integer}9
- result=string

Summary

- We discussed a sample function-oriented software design methodology:
  - Structured Analysis/Structured Design(SA/SD)
  - incorporates features from some important design methodologies.
- SA/SD consists of two parts:
  - structured analysis
  - structured design.

Summary

- The goal of structured analysis:
  - functional decomposition of the system.
- Results of structured analysis:
  - represented using Data Flow Diagrams (DFDs).
- We examined why any hierarchical model is easy to understand.
  - Number 7 is called the magic number.

Summary

- During structured design,
  - the DFD representation is transformed to a structure chart representation.
- DFDs are very popular:
  - because it is a very simple technique.

Summary

- A DFD model:
  - difficult to implement using a programming language:
  - structure chart representation can be easily implemented using a programming language.

Summary

- We discussed structured analysis of two small examples:
  - RMS calculating software
  - tic-tac-toe computer game software

Summary

- Several CASE tools are available:
  - support structured analysis and design.
  - maintain the data dictionary,
  - check whether DFDs are balanced or not.

Function-Oriented Software Design (continued)

Organization of this Lecture

- Brief review of previous lectures
- A larger example of Structured Analysis
- Structured Design
  - A major objective of this lecture is that you should be able to develop structured design from any DFD model.
- Examples
- Summary

Review of Last Lecture

- Last lecture we started discussion on Structured Analysis/ Structured Design (SA/SD) technique:
  - incorporates features from some important design methodologies.
- SA/SD consists of two important parts:
  - structured analysis
  - structured design.

Review of Last Lecture

- The goal of structured analysis:
  - perform functional decomposition.
  - represent using Data Flow Diagrams (DFDs).
- DFDs are a hierarchical model:
  - We examined why any hierarchical model is easy to understand
  - number 7 is called the magic number.

Review of Last Lecture

- During structured analysis:
  - Functional decomposition takes place
  - in addition, data decomposition takes place.
- At the most abstract level:
  - context diagram
  - refined to more detailed levels.
- We discussed two small examples:
  - RMS calculating software
  - tic-tac-toe computer game software

Review of Last Lecture

- Several CASE tools are available
  - help in design activities:
  - help maintain the data dictionary,
  - check whether DFDs are balanced, etc.
- DFD model:
  - difficult to implement using a programming language:
  - needs to be transformed to structured design.

Example 3: Trading-House Automation System (TAS)

- A large trading house wants us to develop a software:
  - to automate book keeping activities associated with its business.
- It has many regular customers:
  - who place orders for various kinds of commodities.

Example 3: Trading-House Automation System (TAS)

- The trading house maintains names and addresses of its regular customers.
- Each customer is assigned a unique customer identification number (CIN).
- As per current practice when a customer places order:
  - the accounts department first checks the credit-worthiness of the customer.

Example: Trading-House Automation System (TAS)

- The credit worthiness of a customer is determined:
  - by analyzing the history of his payments to the bills sent to him in the past.
- If a customer is not credit-worthy:
  - his orders are not processed any further
  - an appropriate order rejection message is generated for the customer.

Example: Trading-House Automation System (TAS)

- If a customer is credit-worthy:
  - items he/she has ordered are checked against the list of items the trading house deals with.
- The items that the trading house does not deal with:
  - are not processed any further
  - an appropriate message for the customer for these items is generated.

Example: Trading-House Automation System (TAS)

- The items in a customer's order that the trading house deals with:
  - are checked for availability in the inventory.
- If the items are available in the inventory in desired quantities:
  - a bill with the forwarding address of the customer is printed.
  - a material issue slip is printed.

Example: Trading-House Automation System (TAS)

- The customer can produce the material issue slip at the store house:
  - take delivery of the items.
  - inventory data adjusted to reflect the sale to the customer.

Example: Trading-House Automation System (TAS)

- If an ordered item is not available in the inventory in sufficient quantity:
  - to be able to fulfill pending orders store details in a "pending-order" file :
    - out-of-stock items along with quantity ordered.
    - customer identification number

Example: Trading-House Automation System (TAS)

- The purchase department:
  - would periodically issue commands to generate indents.
- When generate indents command is issued:
  - the system should examine the "pending-order" file
  - determine the orders that are pending
  - total quantity required for each of the items.

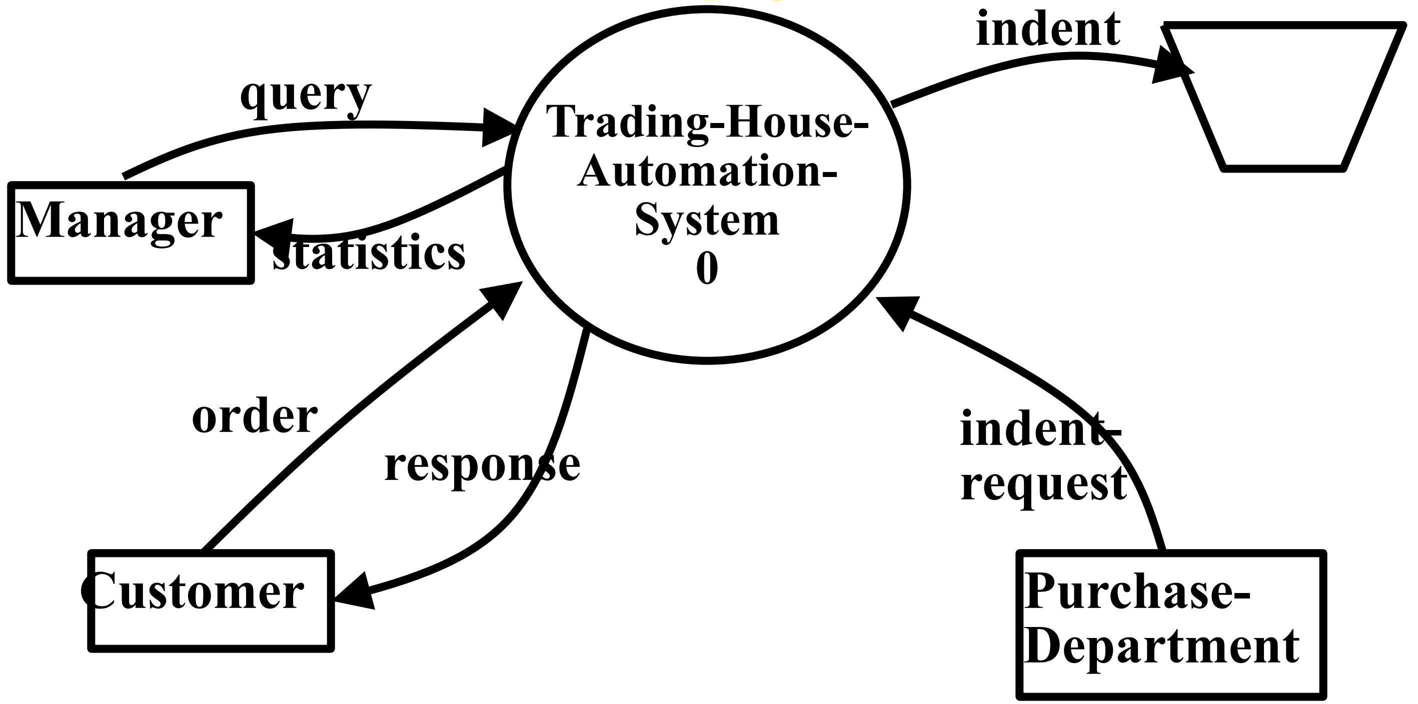
Example: Trading-House Automation System (TAS)

- TAS should find out the addresses of the vendors who supply the required items:
  - examine the file containing vendor details (their address, items they supply etc.)
  - print out indents to those vendors.

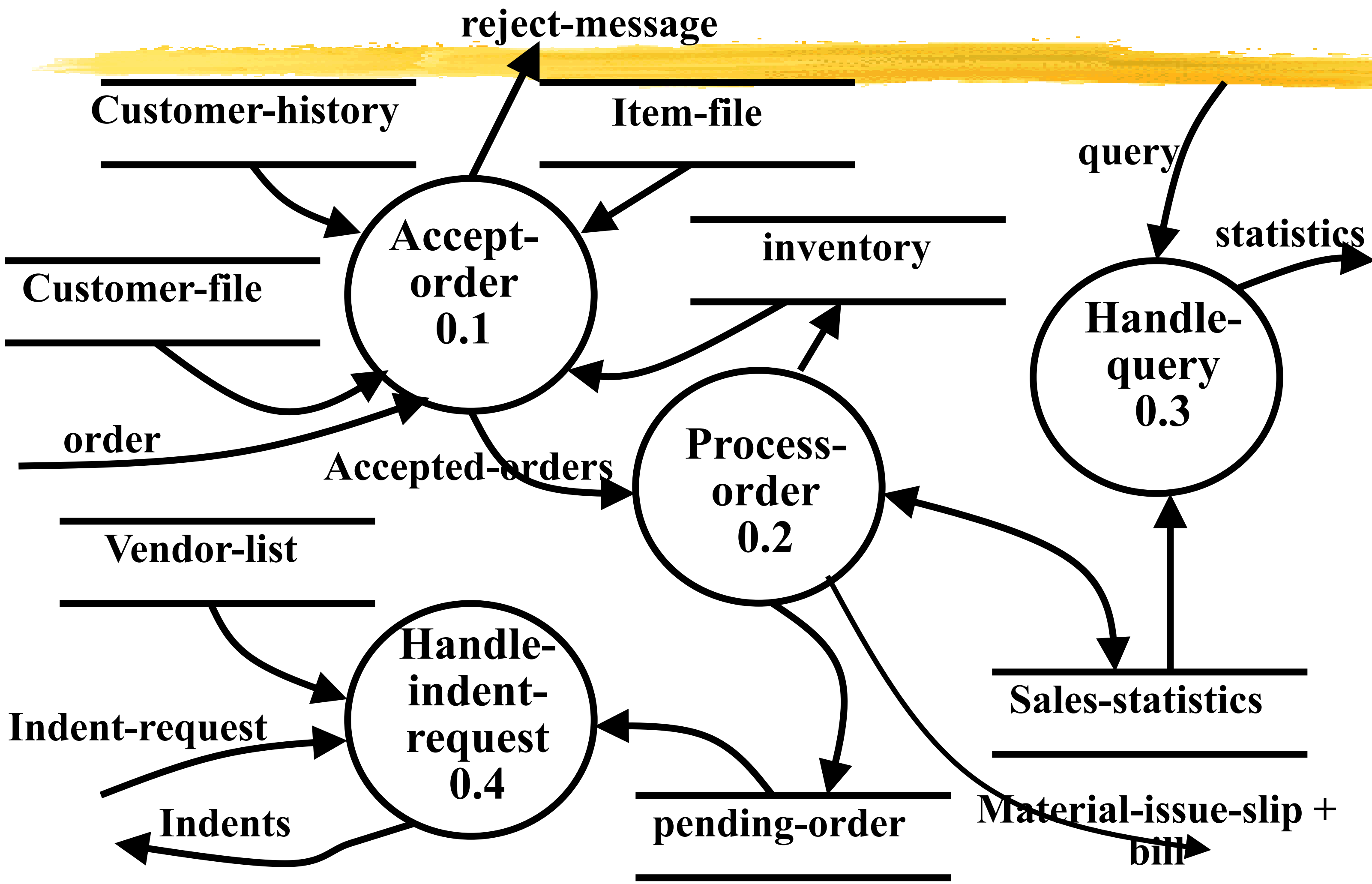
Example: Trading-House Automation System (TAS)

- TAS should also answers managerial queries:
  - statistics of different items sold over any given period of time
  - corresponding quantity sold and the price realized.

Context Diagram



Level 1 DFD


reject-message
Customer-history
Item-file
query
statistics
inventory
Accept-order 0.1
Customer-file
Handle-query 0.3
order
Accepted-orders
Process-order 0.2
Vendor-list
Handle-indent-request 0.4
Indent-request
Sales-statistics
Indents
pending-order
Material-issue-slip + bill

Example: Data Dictionary

- response: [bill + material-issue-slip, reject-message]
- query: period /* query from manager regarding sales statistics*/
- period: [date+date,month,year,day]
- date: year + month + day
- year: integer
- month: integer
- day: integer
- order: customer-id + {items + quantity}*
- accepted-order: order /* ordered items available in inventory */
- reject-message: order + message /* rejection message */
- pending-orders: customer-id + {items+quantity}*
- customer-address: name+house#+street#+city+pin

Example: Data Dictionary

- item-name: string
- house#: string
- street#: string
- city: string
- pin: integer
- customer-id: integer
- bill: {item + quantity + price}* + total-amount + customer-address
- material-issue-slip: message + item + quantity + customer-address
- message: string
- statistics: {item + quantity + price }*
- sales-statistics: {statistics}*
- quantity: integer

Observation

- From the examples,
  - observe that DFDs help create:
    - data model
    - function model

Observation

- As a DFD is refined into greater levels of detail:
  - the analyst performs an implicit functional decomposition.
  - At the same time, refinements of data takes place.

Guidelines For Constructing DFDs

- Context diagram should represent the system as a single bubble:
  - Many beginners commit the mistake of drawing more than one bubble in the context diagram.

Guidelines For Constructing DFDs

- All external entities should be represented in the context diagram:
  - external entities should not appear at any other level of DFD.
- Only 3 to 7 bubbles per diagram should be allowed:
  - each bubble should be decomposed to between 3 and 7 bubbles.

Guidelines For Constructing DFDs

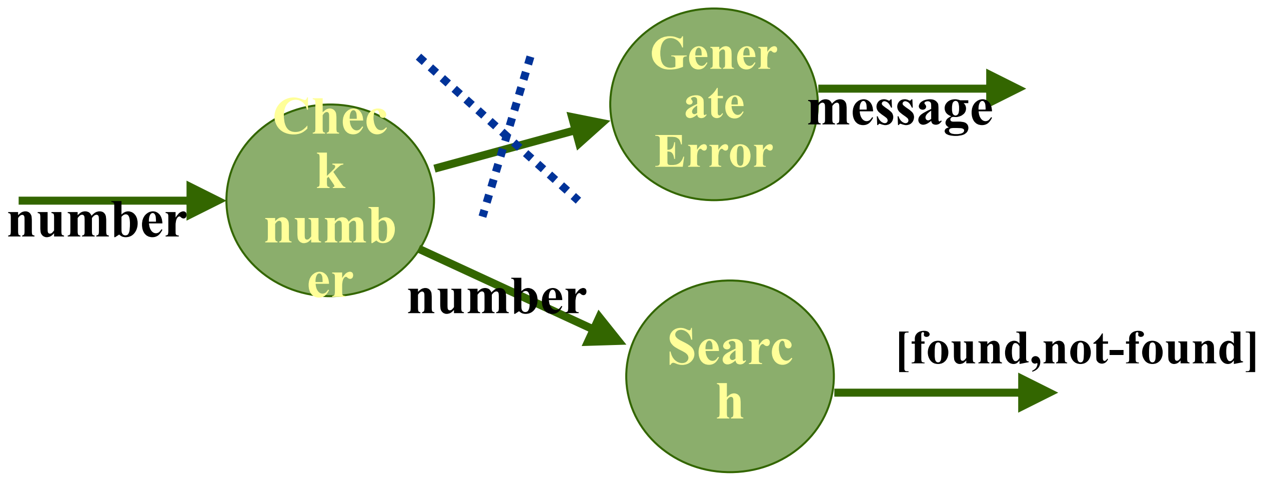
- A common mistake committed by many beginners:
  - attempting to represent control information in a DFD.
  - e.g. trying to represent the order in which different functions are executed.

Guidelines For Constructing DFDs

- A DFD does not represent control information:
  - when or in what order different functions (processes) are invoked
  - the conditions under which different functions are invoked are not represented.
  - For example, a function might invoke one function or another depending on some condition.
  - Many beginners try to represent this aspect by drawing an arrow between the corresponding bubbles.

Example-1

- Check the input value:
  - If the input value is less than -1000 or greater than +1000 generate an error message
  - otherwise search for the number

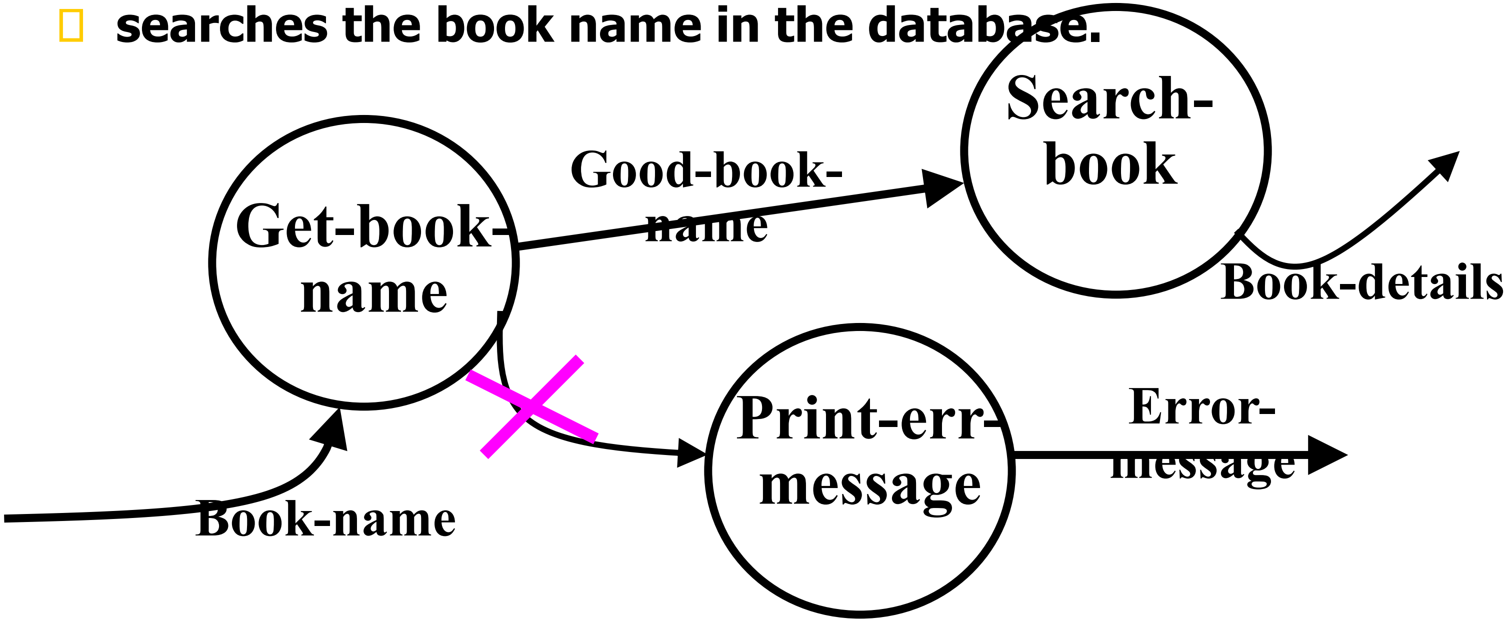


Guidelines For Constructing DFDs

- If a bubble A invokes either bubble B or bubble C depending on some conditions:
  - represent the data that flows from bubble A to bubble B and bubbles A to C
  - not the conditions depending on which a process is invoked.

Example-2

- A function accepts the book name to be searched from the user
- If the entered book name is not a valid book name
  - generates an error message,
- If the book name is valid,
  - searches the book name in the database.



Guidelines For Constructing DFDs

- All functions of the system must be captured in the DFD model:
  - no function specified in the SRS document should be overlooked.
- Only those functions specified in the SRS document should be represented:
  - do not assume extra functionality of the system not specified by the SRS document.

Commonly made errors

- Unbalanced DFDs
- Forgetting to mention the names of the data flows
- Unrepresented functions or data
- External entities appearing at higher level DFDs
- Trying to represent control aspects
- Context diagram having more than one bubble
- A bubble decomposed into too many bubbles in the next level
- Terminating decomposition too early
- Nouns used in naming bubbles

Shortcomings of the DFD Model

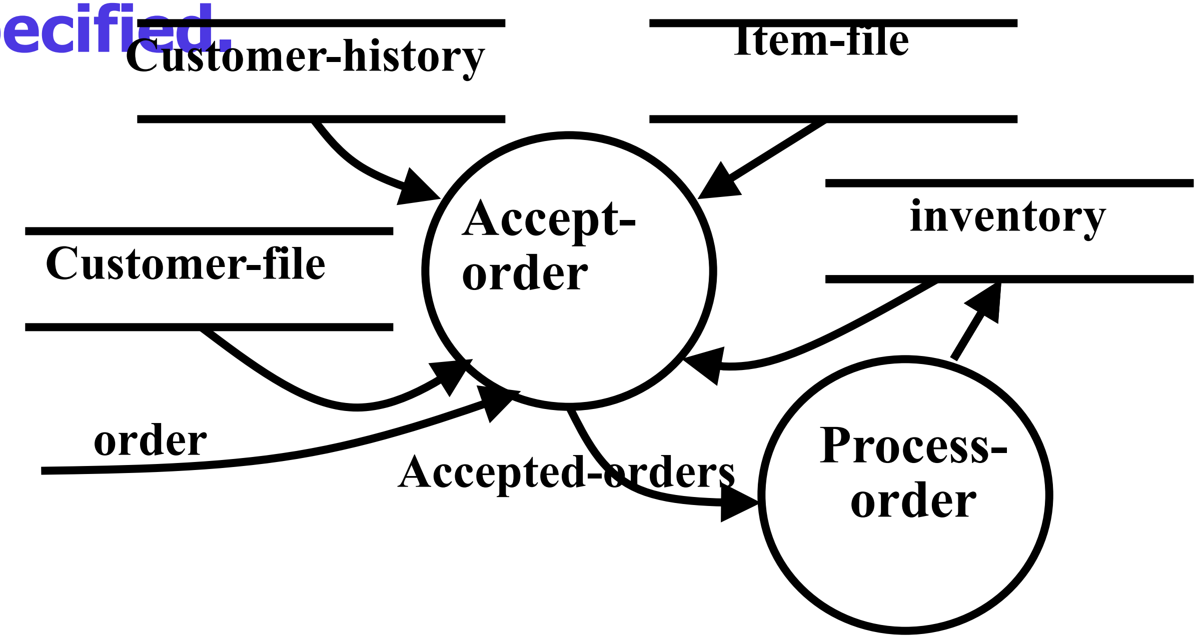
- DFD models suffer from several shortcomings:
- DFDs leave ample scope to be imprecise.
  - In a DFD model, we infer about the function performed by a bubble from its label.
  - A label may not capture all the functionality of a bubble.

Shortcomings of the DFD Model

- For example, a bubble named find-book-position has only intuitive meaning:
  - does not specify several things:
    - what happens when some input information is missing or is incorrect.
    - Does not convey anything regarding what happens when book is not found
    - or what happens if there are books by different authors with the same book title.

Shortcomings of the DFD Model

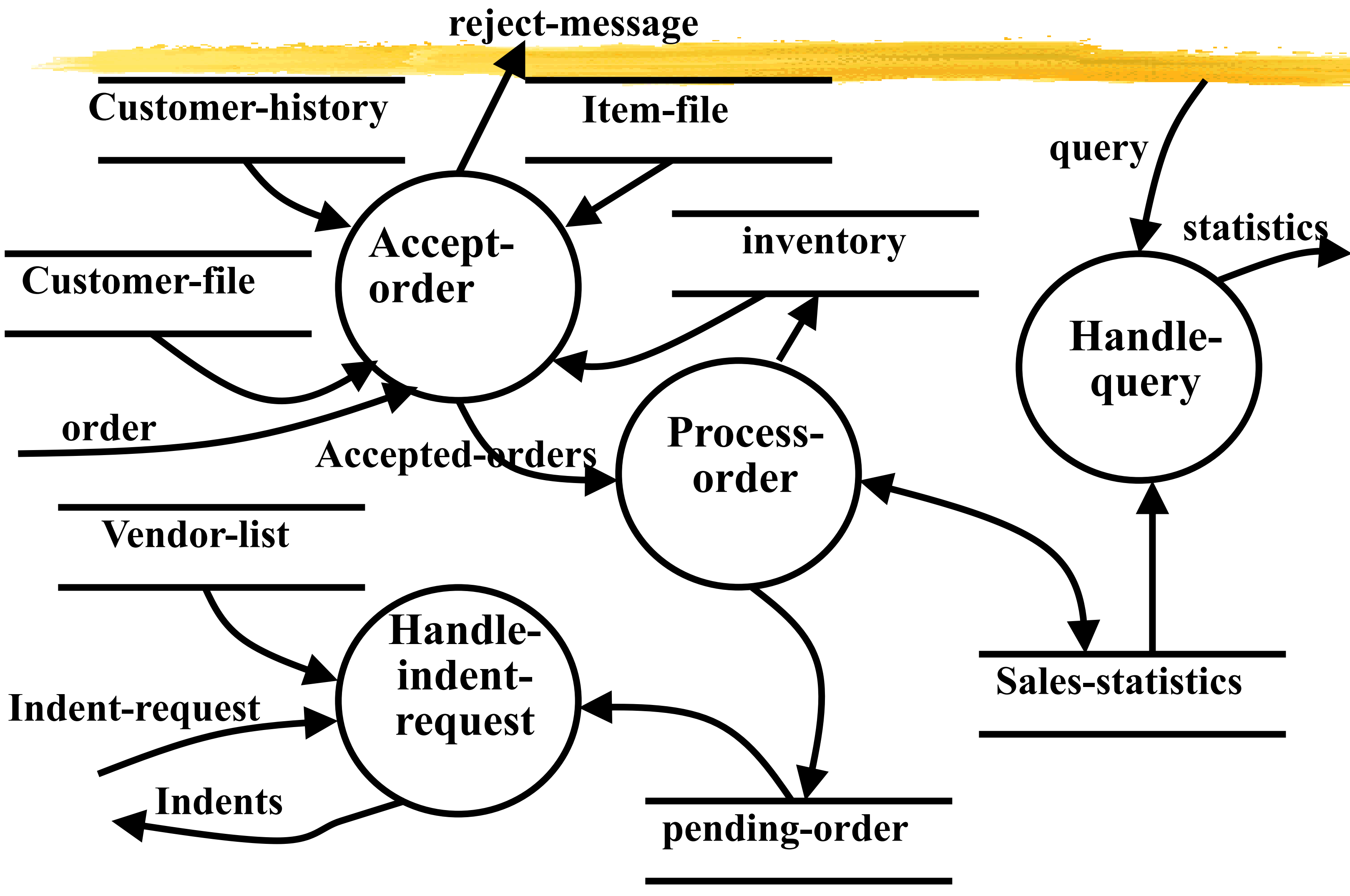
- Control information is not represented:
  - For instance, order in which inputs are consumed and outputs are produced is not specified.



Shortcomings of the DFD Model

- A DFD does not specify synchronization aspects:
  - For instance, the DFD in TAS example does not specify:
    - whether process-order may wait until the accept-order produces data
    - whether accept-order and handle-order may proceed simultaneously with some buffering mechanism between them.

TAS: Level 1 DFD



Shortcomings of the DFD Model

- The way decomposition is carried out to arrive at the successive levels of a DFD is subjective.
- The ultimate level to which decomposition is carried out is subjective:
  - depends on the choice and judgement of the analyst.
- Even for the same problem,
  - several alternative DFD representations are possible:
  - many times it is not possible to say which DFD representation is superior or preferable.

Shortcomings of the DFD Model

- DFD technique does not provide:
  - any clear guidance as to how exactly one should go about decomposing a function:
  - one has to use subjective judgement to carry out decomposition.
- Structured analysis techniques do not specify when to stop a decomposition process:
  - to what length decomposition needs to be carried out.

Extending DFD Technique to Real-Time Systems

- For real-time systems (systems having time bounds on their actions),
  - essential to model control flow and events.
  - Widely accepted technique: Ward and Mellor technique.
    - a type of process (bubbles) that handles only control flows is introduced.
    - These processes are represented using dashed circles.

Structured Design

- The aim of structured design
  - transform the results of structured analysis (i.e., a DFD representation) into a structure chart.
- A structure chart represents the software architecture:
  - various modules making up the system,
  - module dependency (i.e. which module calls which other modules),
  - parameters passed among different modules.

Structure Chart

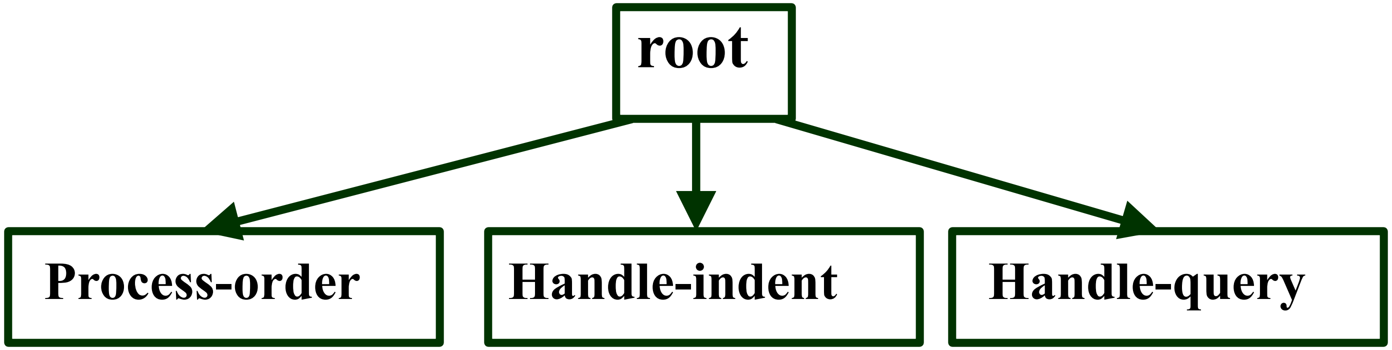
- Structure chart representation
  - easily implementable using programming languages.
- Main focus of a structure chart:
  - define the module structure of a software,
  - interaction among different modules,
  - procedural aspects (e.g, how a particular functionality is achieved) are not represented.

Basic building blocks of structure chart

- Rectangular box:
  - A rectangular box represents a module.
  - annotated with the name of the module it represents.

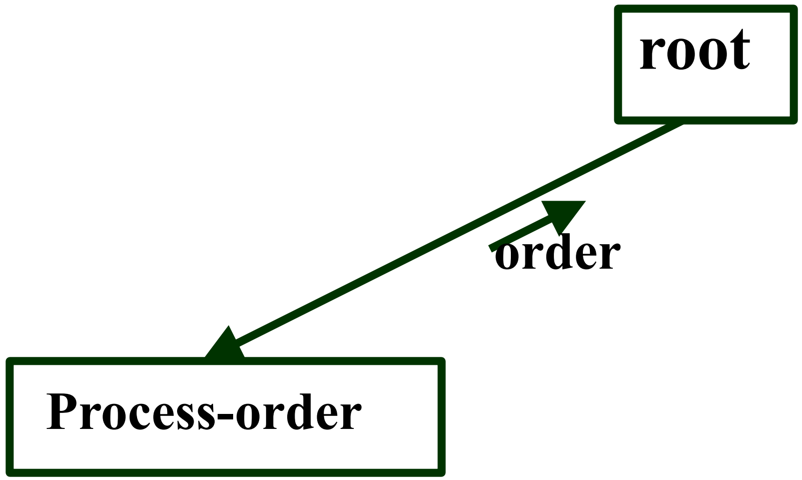
Arrows

- An arrow between two modules implies:
  - during execution control is passed from one module to the other in the direction of the arrow.



Data flow Arrows

- Data flow arrows represent:
  - data passing from one module to another in the direction of the arrow.

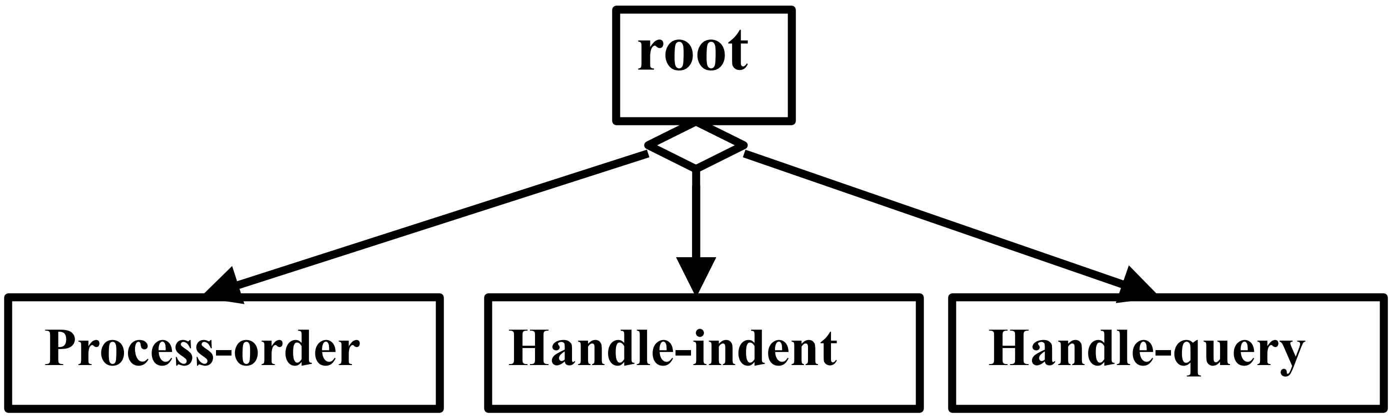


Library modules

- Library modules represent frequently called modules:
  - a rectangle with double side edges.
  - Simplifies drawing when a module is called by several modules.

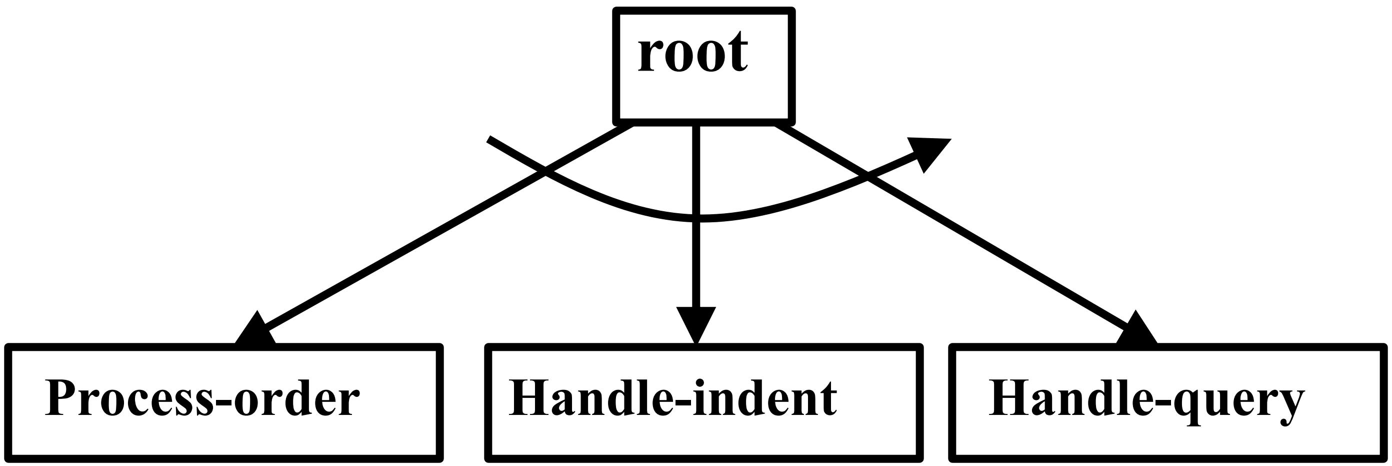
Selection

- The diamond symbol represents:
  - one module of several modules connected to the diamond symbol is invoked depending on some condition.



Repetition

- A loop around control flow arrows denotes that the concerned modules are invoked repeatedly.



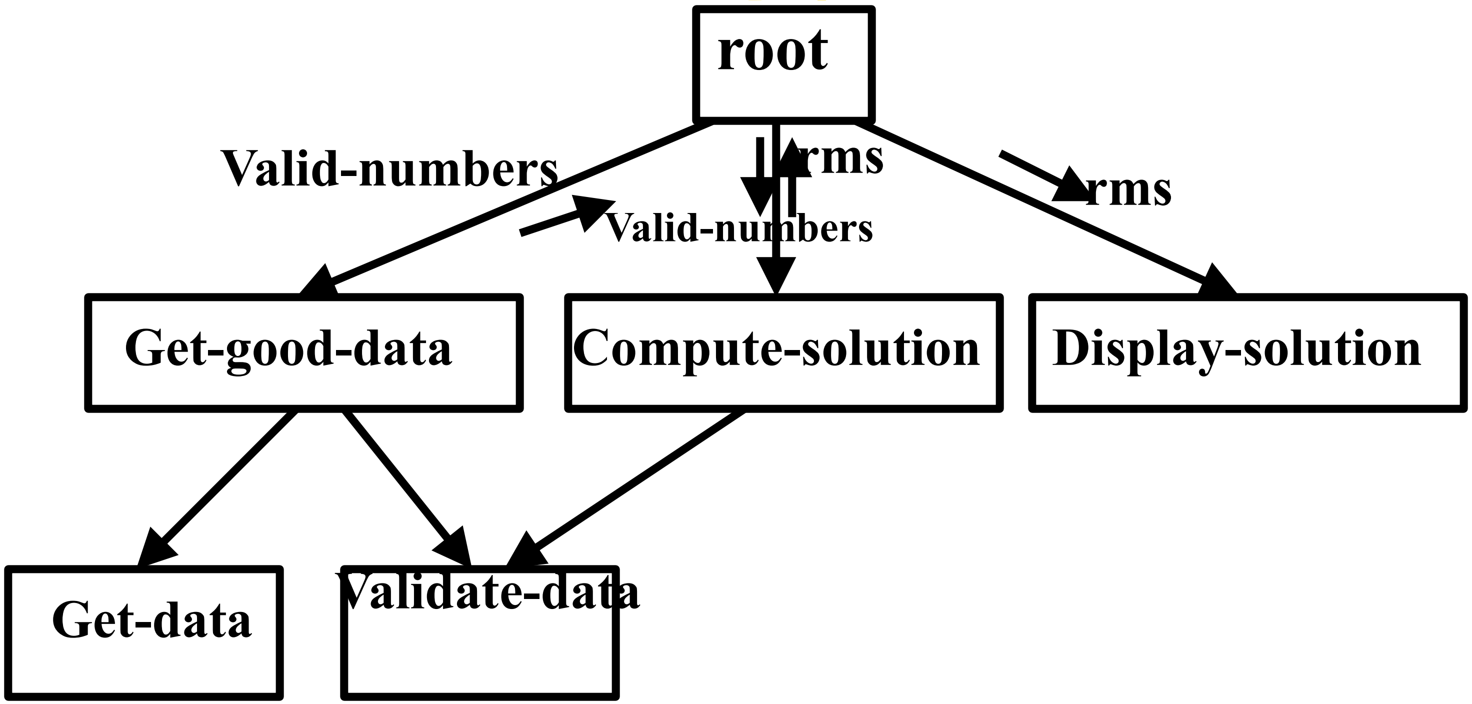
Structure Chart

- There is only one module at the top:
  - the root module.
- There is at most one control relationship between any two modules:
  - if module A invokes module B,
  - module B cannot invoke module A.
- The main reason behind this restriction:
  - consider modules in a structure chart to be arranged in layers or levels.

Structure Chart

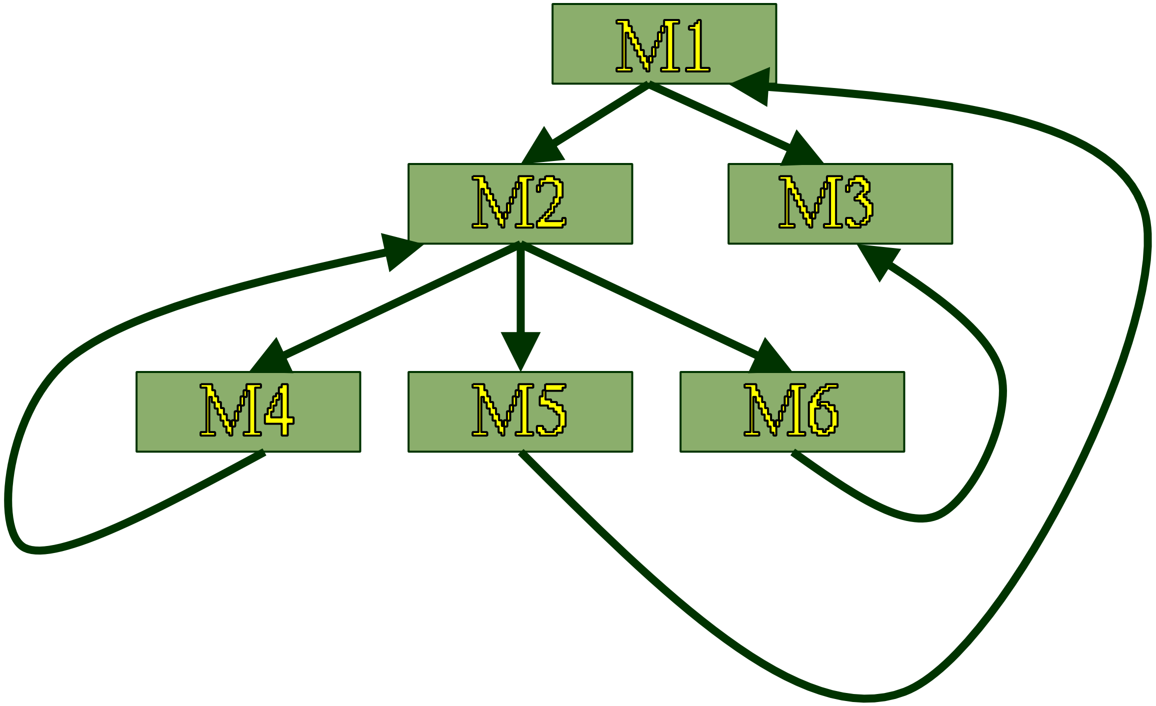
- The principle of abstraction:
  - does not allow lower-level modules to invoke higher-level modules:
  - But, two higher-level modules can invoke the same lower-level module.

Example



root
Valid-numbers
rms
Valid-numbers
rms
Get-good-data
Compute-solution
Display-solution
Get-data
Validate-data

Bad Design



Shortcomings of Structure Chart

- By looking at a structure chart:
  - we can not say whether a module calls another module just once or many times.
- Also, by looking at a structure chart:
  - we can not tell the order in which the different modules are invoked.

Transformation of a DFD Model into Structure Chart

- Two strategies exist to guide transformation of a DFD into a structure chart:
  - Transform Analysis
  - Transaction Analysis

Transform Analysis

- The first step in transform analysis:
  - divide the DFD into 3 types of parts:
    - input,
    - logical processing,
    - output.

Transform Analysis

- Input portion in the DFD:
  - processes which convert input data from physical to logical form.
  - e.g. read characters from the terminal and store in internal tables or lists.
- Each input portion:
  - called an afferent branch.
  - Possible to have more than one afferent branch in a DFD.

Transform Analysis

- Output portion of a DFD:
  - transforms output data from logical form to physical form.
    - e.g., from list or array into output characters.
  - Each output portion:
    - called an efferent branch.
- The remaining portions of a DFD
  - called central transform

Transform Analysis

- Derive structure chart by drawing one functional component for:
  - the central transform,
  - each afferent branch,
  - each efferent branch.

Transform Analysis

- Identifying the highest level input and output transforms:
  - requires experience and skill.

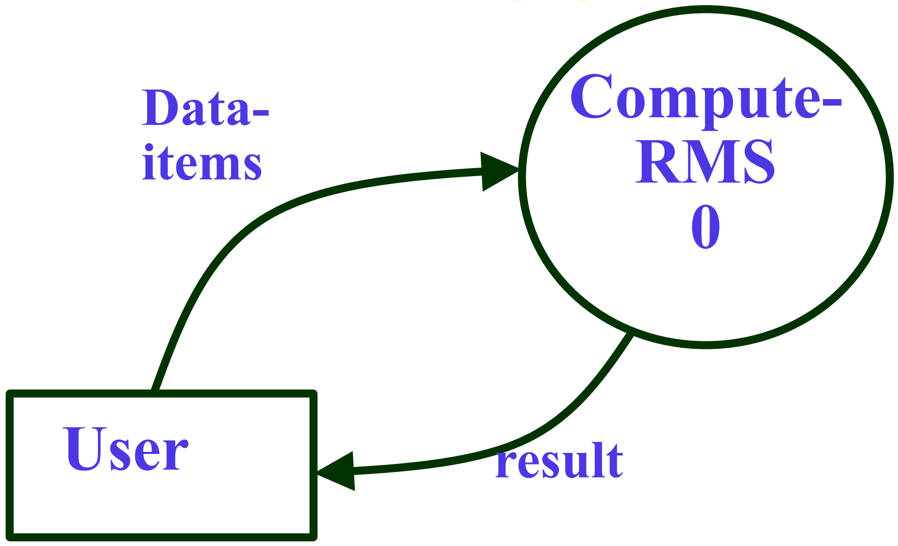
Transform Analysis

- First level of structure chart:
  - draw a box for each input and output units
  - a box for the central transform.
- Next, refine the structure chart:
  - add subfunctions required by each high-level module.
  - Many levels of modules may required to be added.

Factoring

- The process of breaking functional components into subcomponents.
- Factoring includes adding:
  - read and write modules,
  - error-handling modules,
  - initialization and termination modules, etc.
- Finally check:
  - whether all bubbles have been mapped to modules.

Example 1: RMS Calculating Software

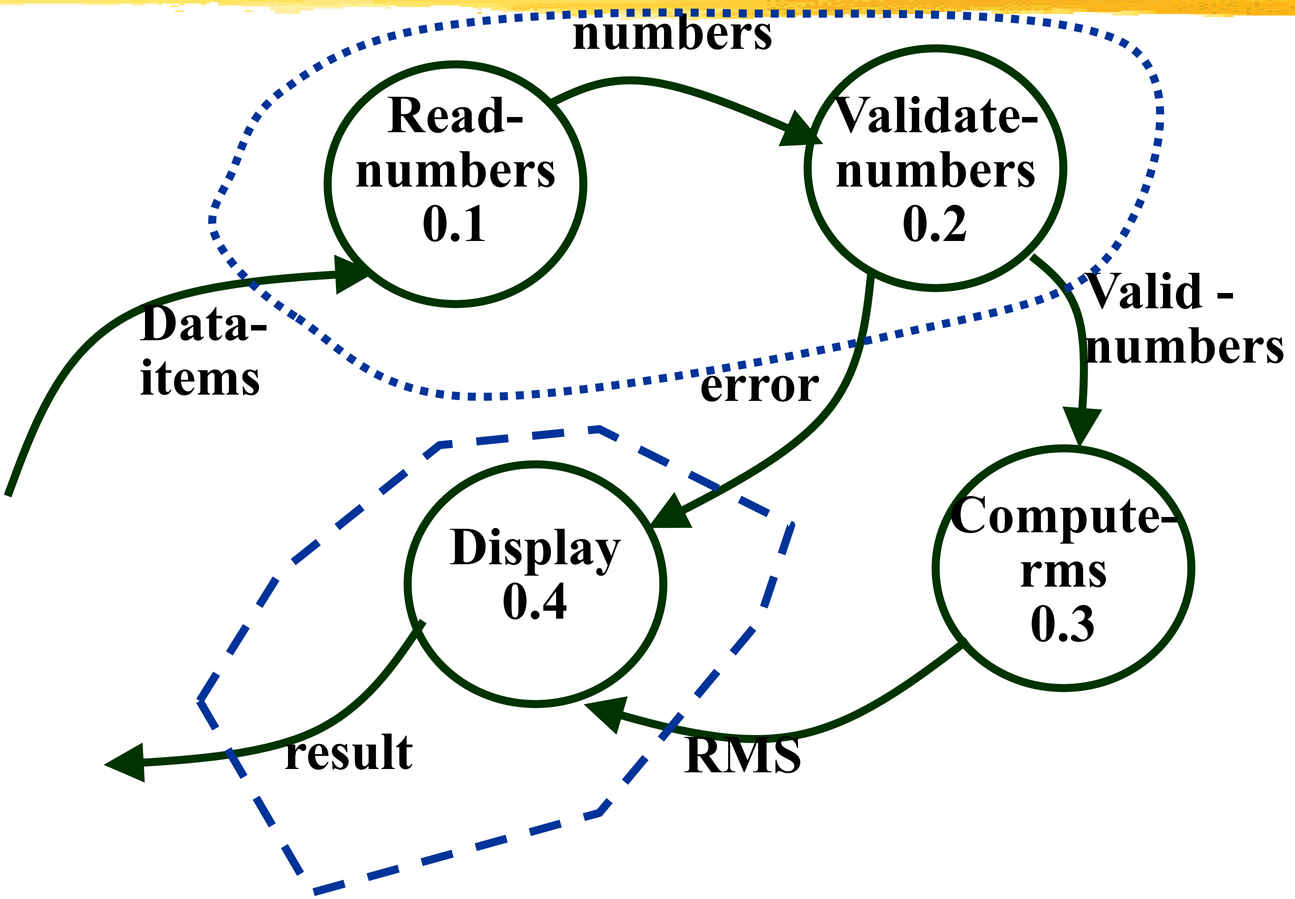


Context Diagram

Example 1: RMS Calculating Software

- From a cursory analysis of the problem description,
  - easy to see that the system needs to perform:
    - accept the input numbers from the user,
    - validate the numbers,
    - calculate the root mean square of the input numbers,
    - display the result.

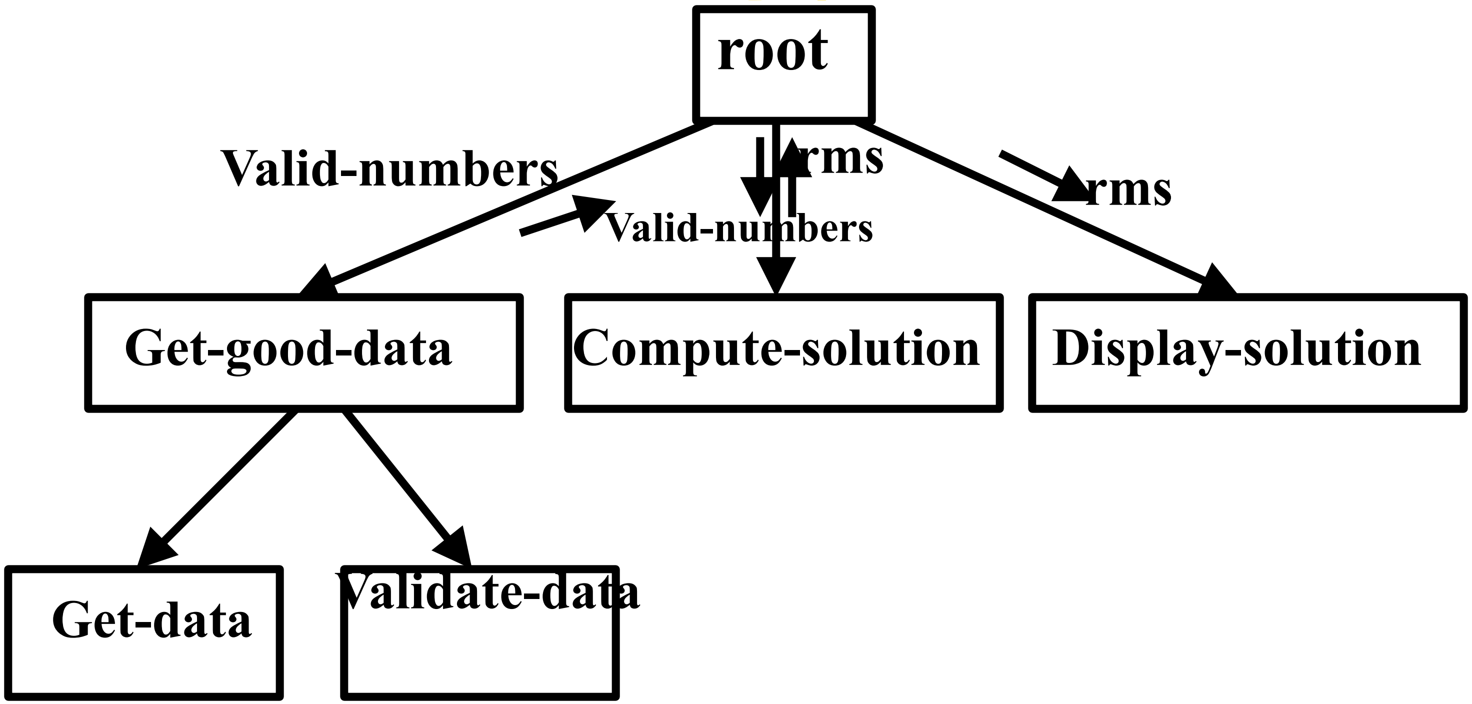
Example 1: RMS Calculating Software



Example 1: RMS Calculating Software

- By observing the level 1 DFD:
  - identify read-number and validate-number bubbles as the afferent branch
  - display as the efferent branch.

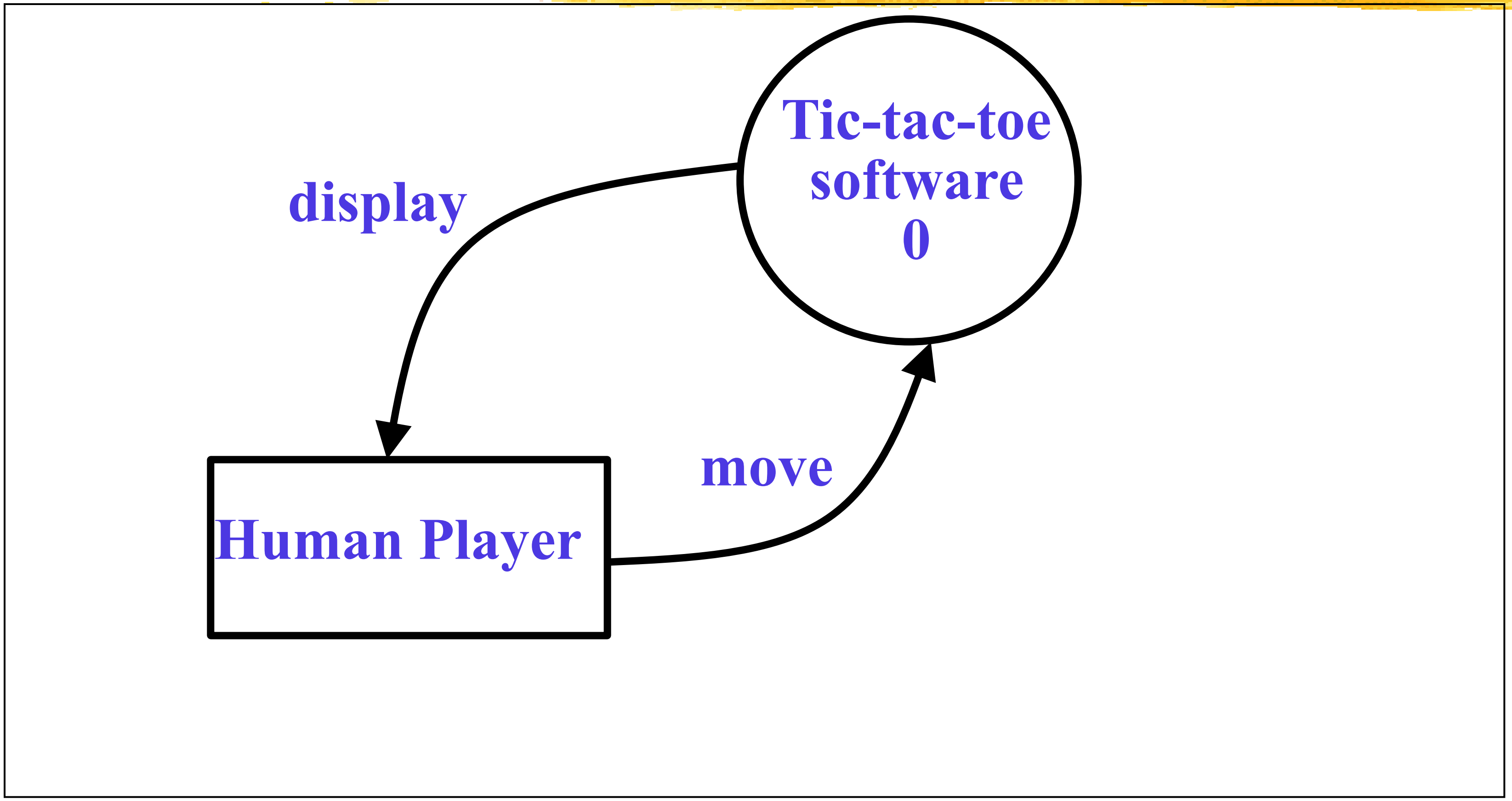
Example 1: RMS Calculating Software



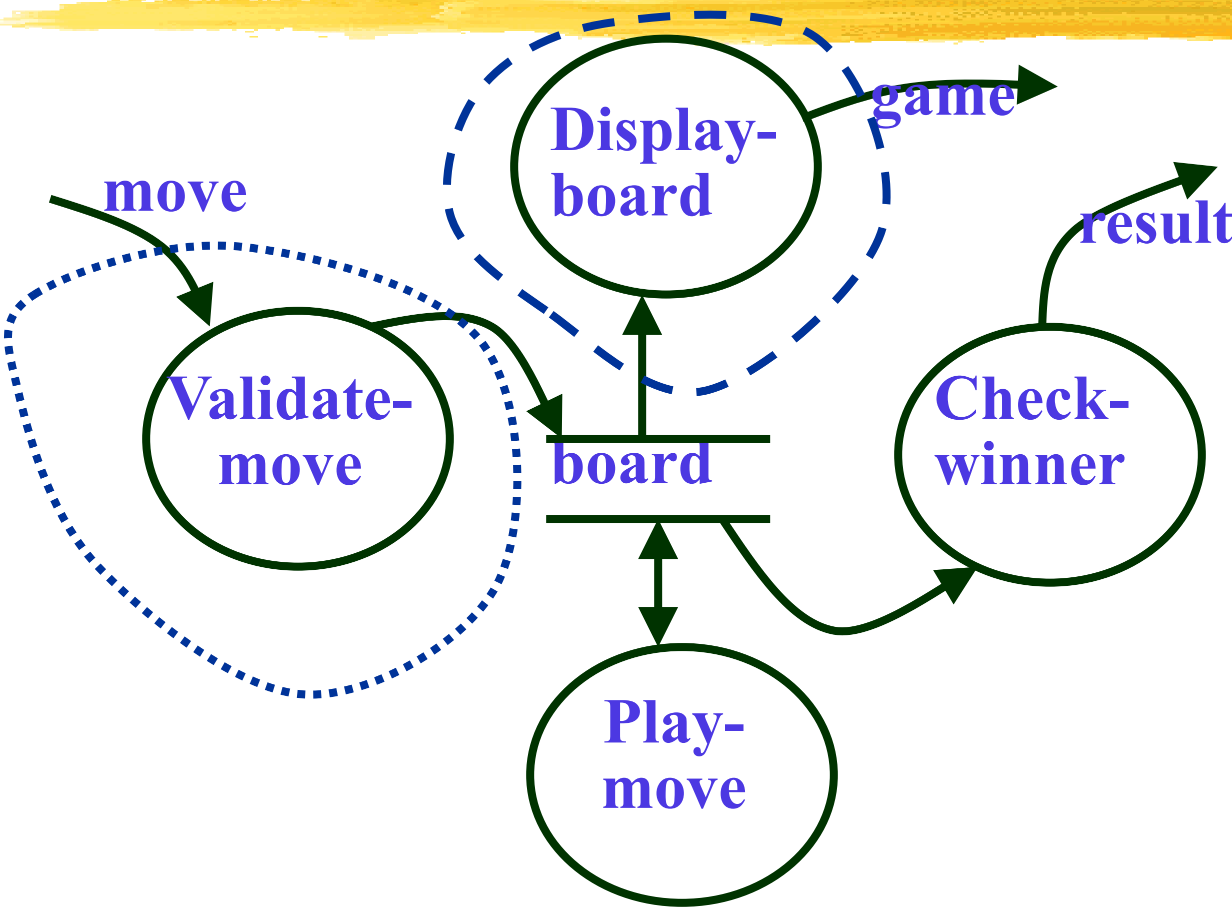
Example 2: Tic-Tac-Toe Computer Game

- As soon as either of the human player or the computer wins,
  - a message congratulating the winner should be displayed.
- If neither player manages to get three consecutive marks along a straight line,
  - and all the squares on the board are filled up,
  - then the game is drawn.
- The computer always tries to win a game.

Context Diagram for Example 2

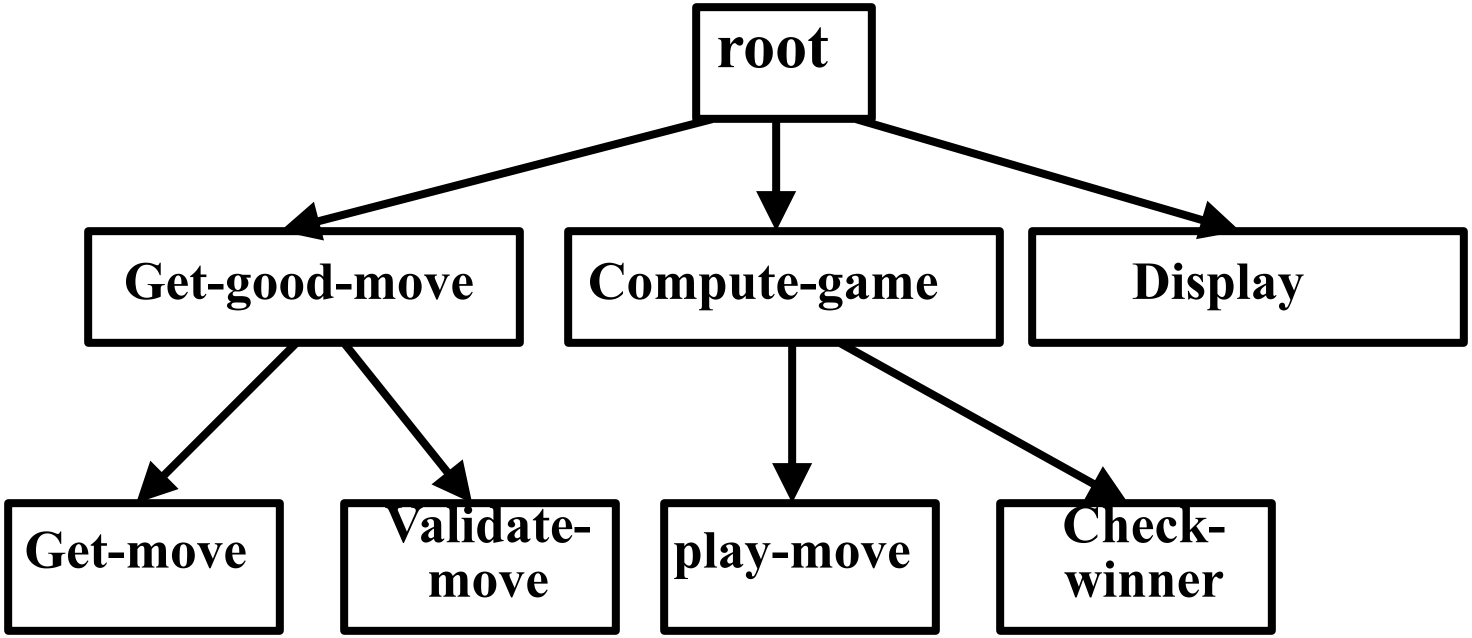


Level 1 DFD



game
Display-board
move
result
Validate-move
board
Check-winner
Play-move

Structure Chart



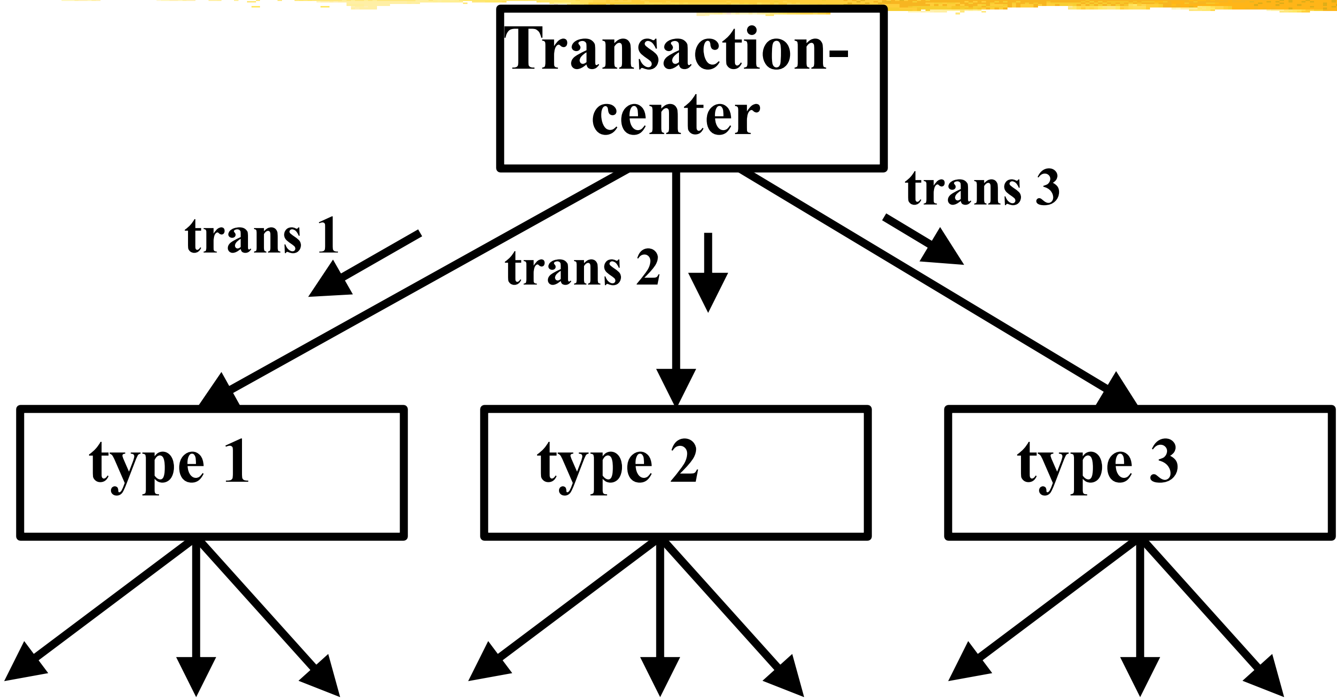
Transaction Analysis

- **Useful for designing transaction processing programs.**
  - **Transform-centered systems:**
    - **characterized by <u>similar processing steps for every data item</u> processed by input, process, and output bubbles.**
  - **Transaction-driven systems,**
    - **<u>one of several possible paths</u> through the DFD is traversed depending upon the input data value.**

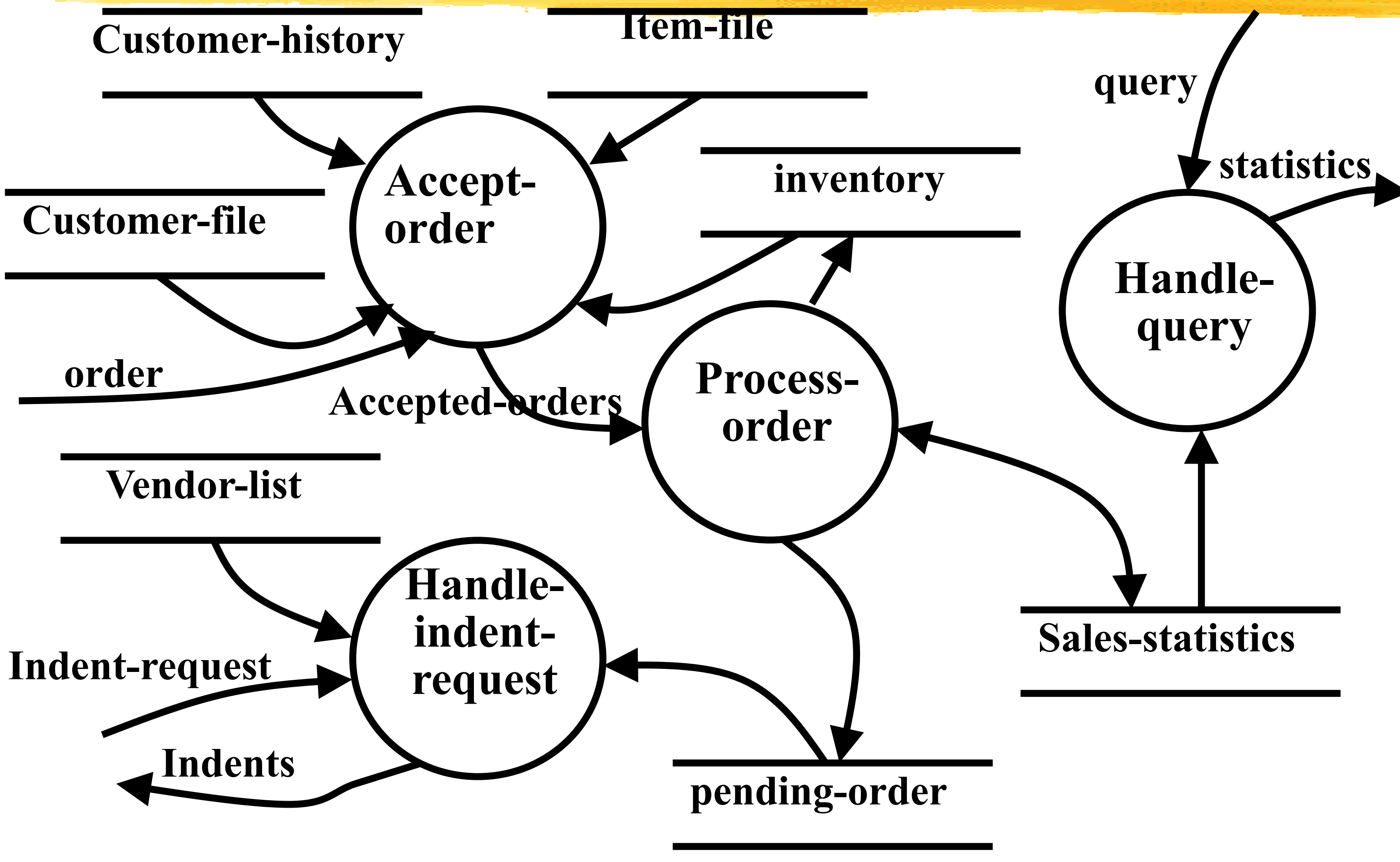
Transaction Analysis

- Transaction:
  - any input data value that triggers an action:
  - For example, selected menu options might trigger different functions.
  - Represented by a tag identifying its type.
- Transaction analysis uses this tag to divide the system into:
  - several transaction modules
  - one transaction-center module.

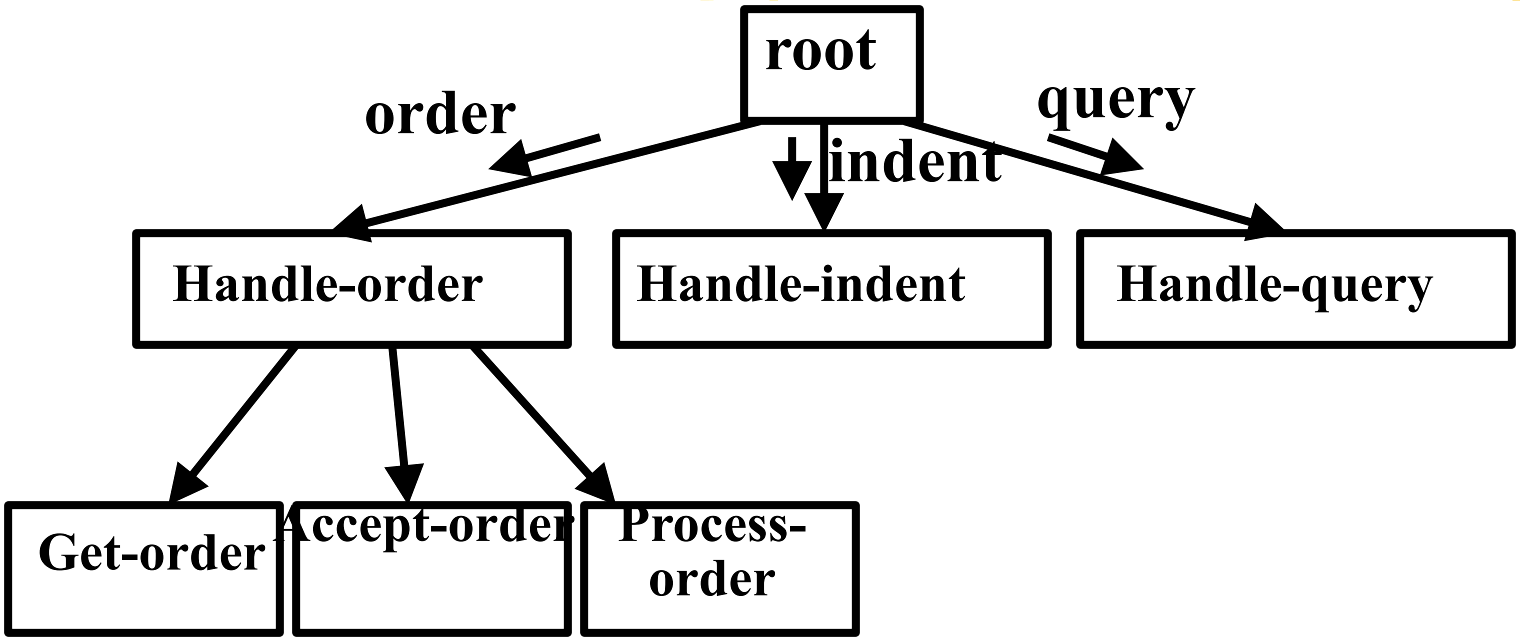
Transaction analysis



Level 1 DFD for TAS



Structure Chart



Summary

- We first discussed structured analysis of a larger problem.
- We defined some general guidelines
  - for constructing a satisfactory DFD model.
- The DFD model though simple and useful
  - does have several short comings.
- We then started discussing structured design.

Summary

- Aim of structured design:
  - transform a DFD representation into a structure chart.
- Structure chart represents:
  - module structure
  - interaction among different modules,
  - procedural aspects are not represented.

Summary

- Structured design provides two strategies to transform a DFD into a structure chart:
  - Transform Analysis
  - Transaction Analysis

Summary

- We Discussed three examples of structured design.
- It takes a lot of practice to become a good software designer:
  - Please try to solve all the problems listed in your assignment sheet,
  - not only the ones you are expected to submit.