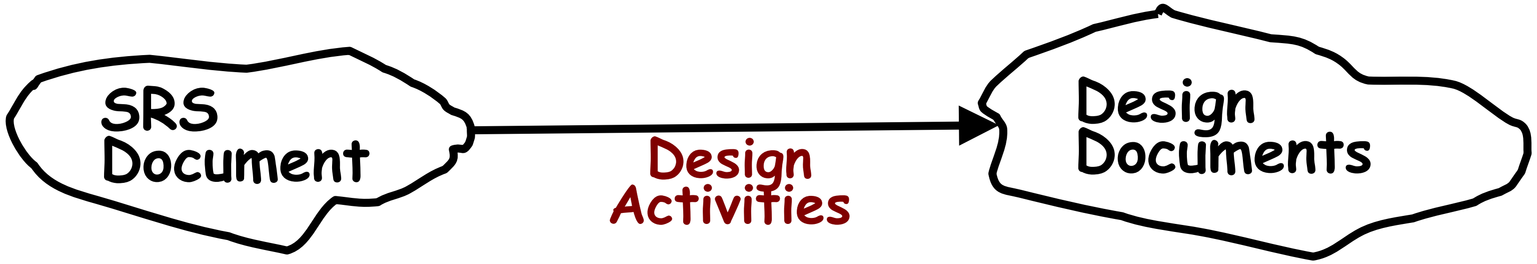
Software Design

Introduction

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



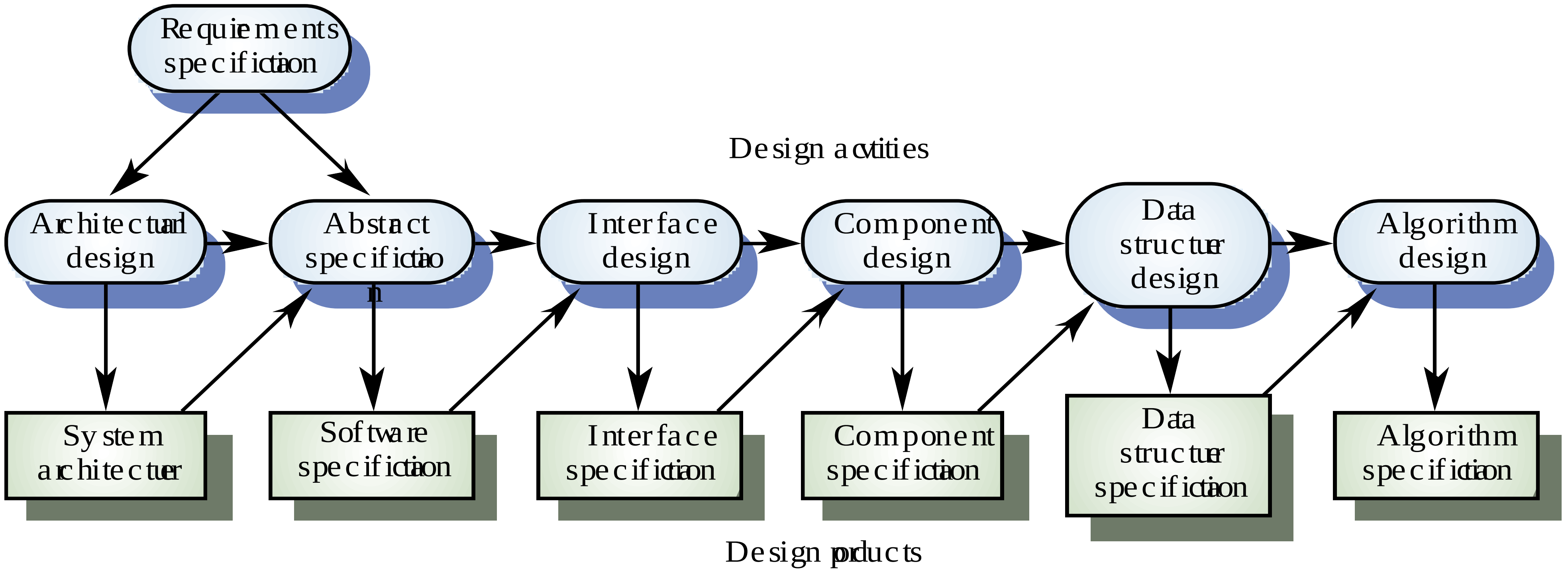
Stages of Design

- **Problem understanding**
  - Look at the problem from different angles to discover the design requirements.
- **Identify one or more solutions**
  - Evaluate possible solutions and choose the most appropriate depending on the designer's experience and available resources.
- **Describe solution abstractions**
  - Use graphical, formal or other descriptive notations to describe the components of the design.
- **Repeat process for each identified abstraction**
  until the design is expressed in primitive terms.

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.

Items Designed During Design Phase



Design Phases

- ***Architectural design:*** Identify sub-systems.
- ***Abstract specification:*** Specify sub-systems.
- ***Interface design:*** Describe sub-system interfaces.
- ***Component design:*** Decompose sub-systems into components.
- ***Data structure design:*** Design data structures to hold problem data.
- ***Algorithm design:*** Design algorithms for problem functions.
- Design activities are usually classified into two stages:
  - Preliminary (or high-level) design (architecture design).
  - Detailed design.

Architecture

- Any complex system is composed of sub-systems that interact
- While designing systems, an approach is to identify sub-systems and how they interact with each other
- Software architecture tries to do this for software
- <u>Architecture is the system design at the highest level</u>
- Choices about technologies, products to use, servers, etc. are made at architecture level
  - Not possible to design system details and then accommodate these choices
  - Architecture must be created accommodating them
- Is the earliest place when properties like reliability/ performance can be evaluated

Architecture

- Software architecture is the structure or structures which comprise elements, their externally visible properties, and relationships among them
  - For elements only interested in external properties needed for relationship specification
  - Details on how the properties are supported is not important for architecture
  - The definition does not say anything about whether an architecture is good or not – analysis needed for it
- An architecture description describes the different structures of the system

Key Uses of Arch Descriptions

- **Understanding and communication**
  - By showing a system at a high level and hiding complexity of parts, arch descr facilitates communication
  - To get a common understanding between the diff stakeholders (users, clients, architect, designer,…)
  - Arch descr can also aid in understanding of existing systems
- **Reuse**
  - To reuse existing components, arch must be chosen such that these components fit together with other components
  - Hence, decision about using existing components is made at arch design time

Uses..

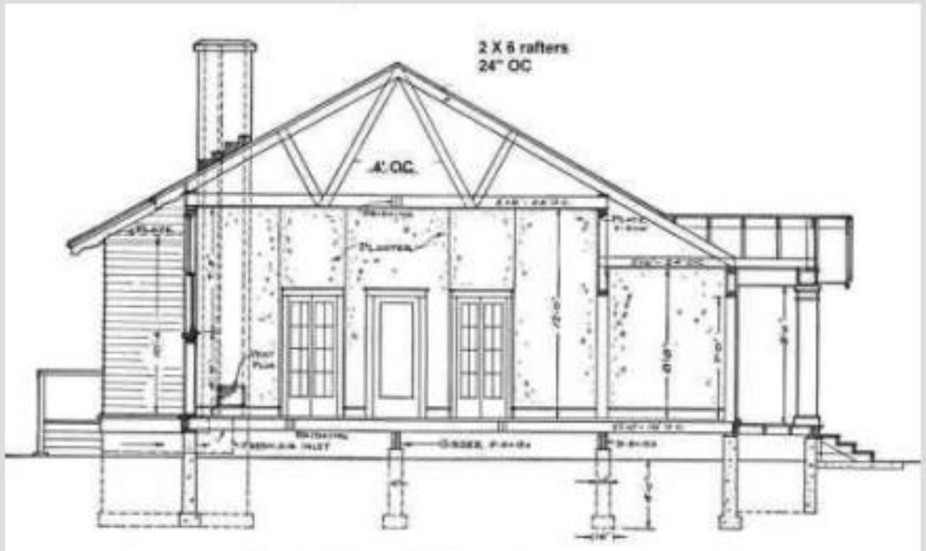
- **Construction and evolution**
  - Some structures in arch descr will be used to guide system development
  - Partitioning at arch level can also be used for work allocation to teams as parts are relatively independent
  - During sw evolution, arch helps decide what needs to be changed to incorporate the new changes/features
  - Arch can help decide what is the impact of changes to existing components on others

Uses…

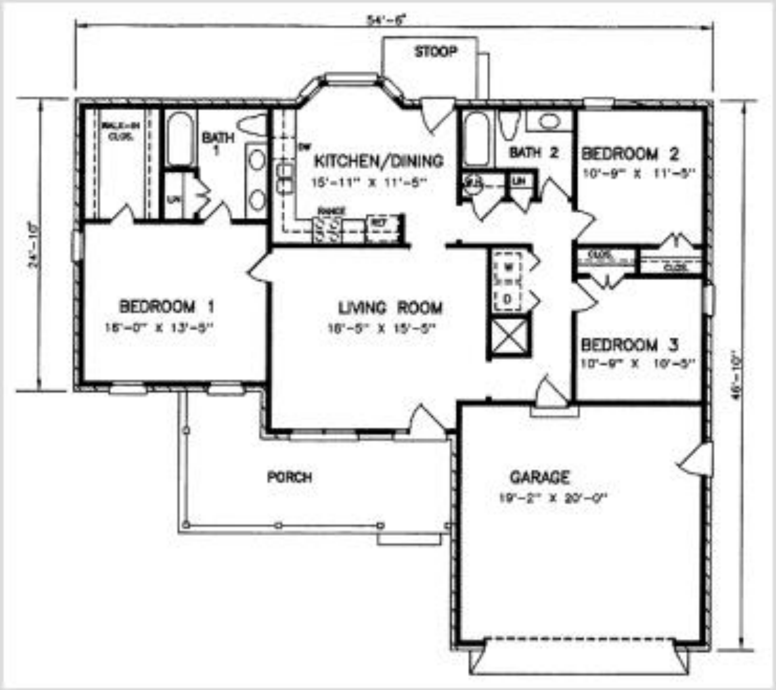
- **Analysis**
  - If properties like performance, reliability can be determined from design, alternatives can be considered during design to reach the desired performance levels
  - E.g. reliability and performance of a system can be predicted from its arch, if estimates for parameters like load etc. is provided
  - Will require precise description of arch, as well as properties of the elements in the description

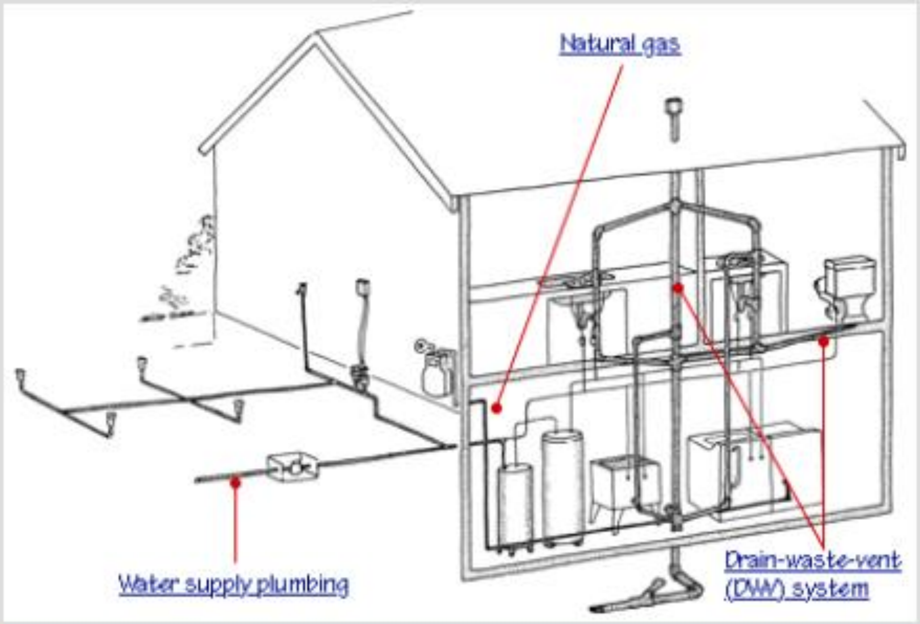
Architectural Views

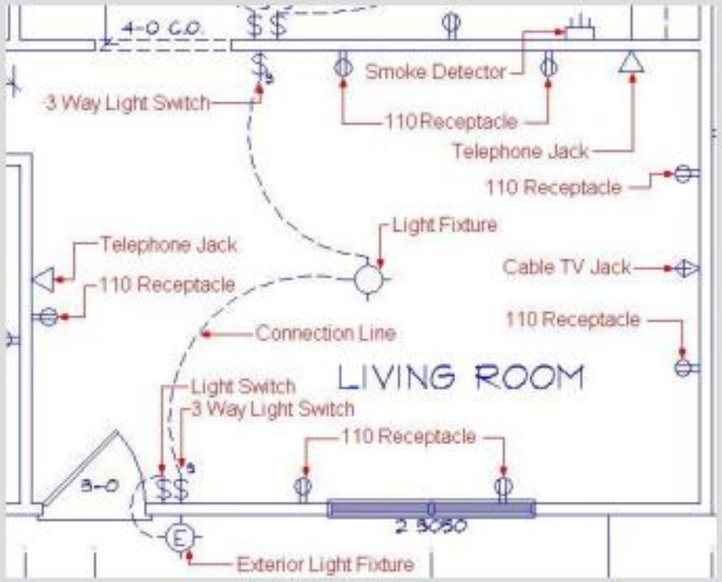
- There are different *views* of a software system
- A view consists of *elements* and *relationships* between them, and describes a *structure*
- The elements of a view depends on what the view wants to highlight, Diff views expose diff properties
- A view focusing on some aspects reduces its complexity
- Most views belong to one of these three types
  - Module
  - Component and Connector
  - Allocation
- The diff views are not unrelated – they all represent the same system



Sections View







Views…

- Module view
  - A sys is a collection of code units i.e. they do not represent runtime entities
  - I.e. elements are modules, eg. Class, package, function, procedure,…
  - Relationship between them is code based, e.g. part of, depends on, calls, generalization-specialization,..

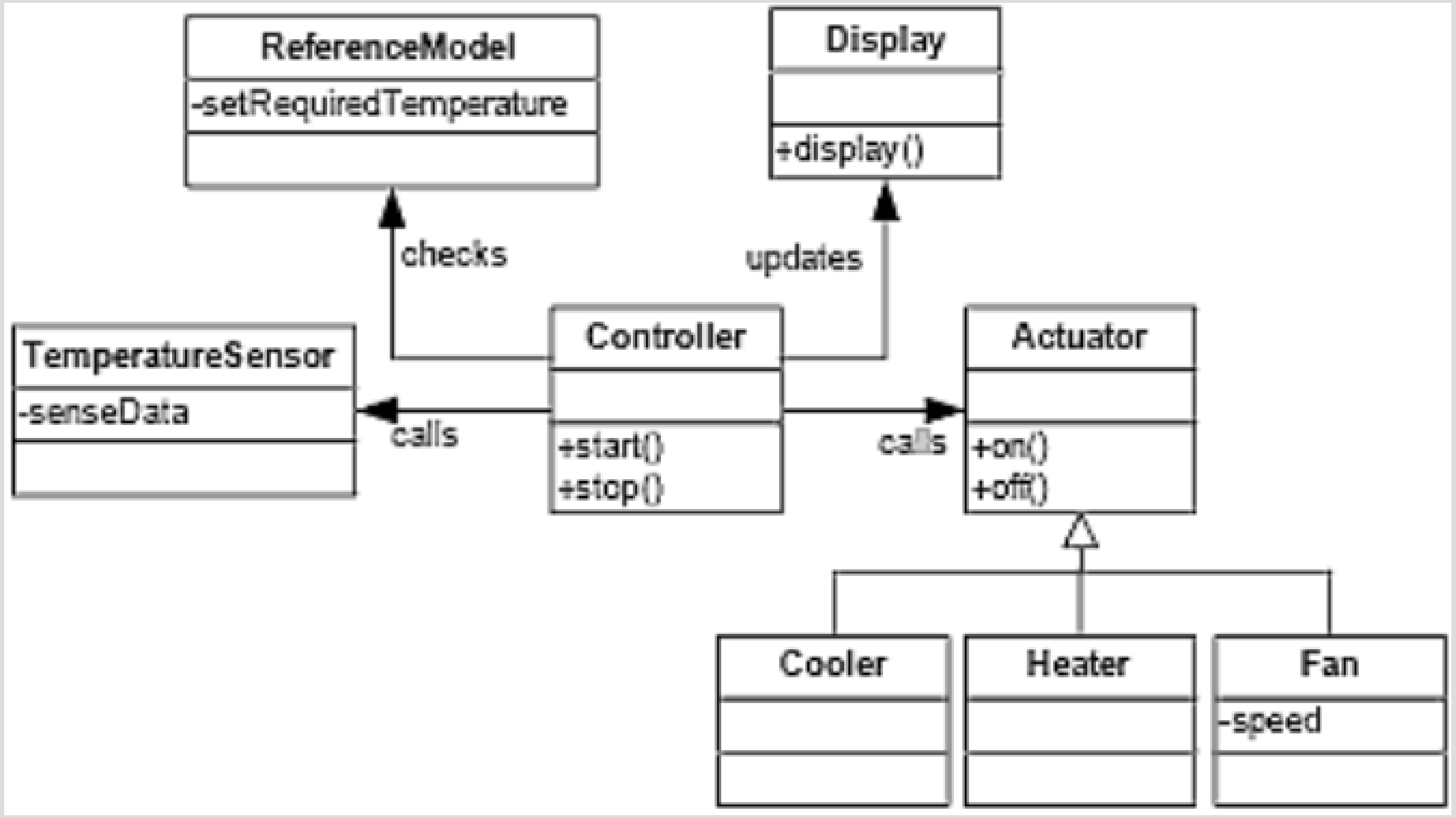
Module View

- Elements - modules, implementation units of software that provide a coherent set of responsibilities
- Relations
  - Object oriented
  - Is part of, a part/whole relationship
  - Depends on, a dependency relationship
  - between two modules
  - Is a, a generalization/specialization relationship
  - Layered – aggregation of modules into layers
- **UML: Package and class diagrams**

Module View

Module View Example

Climate control system in vehicles



Module View

- Static functional decomposition
- System information architecture
- Supports the definition of work assignments, development process and schedules
  - Blueprint for coding and testing
  - Change-impact analysis
  - Requirements traceability analysis

- "It is unlikely that the documentation of any software architecture can be complete without at least one module view."

Views…

- Component and Connector (C&C)
  - Elements are run time entities called components
  - I.e. a component is a unit that has identity in executing sys, e.g. objects, processes, .exe, .dll
  - Connectors provide means of interaction between components, e.g. pipes, shared memory, sockets

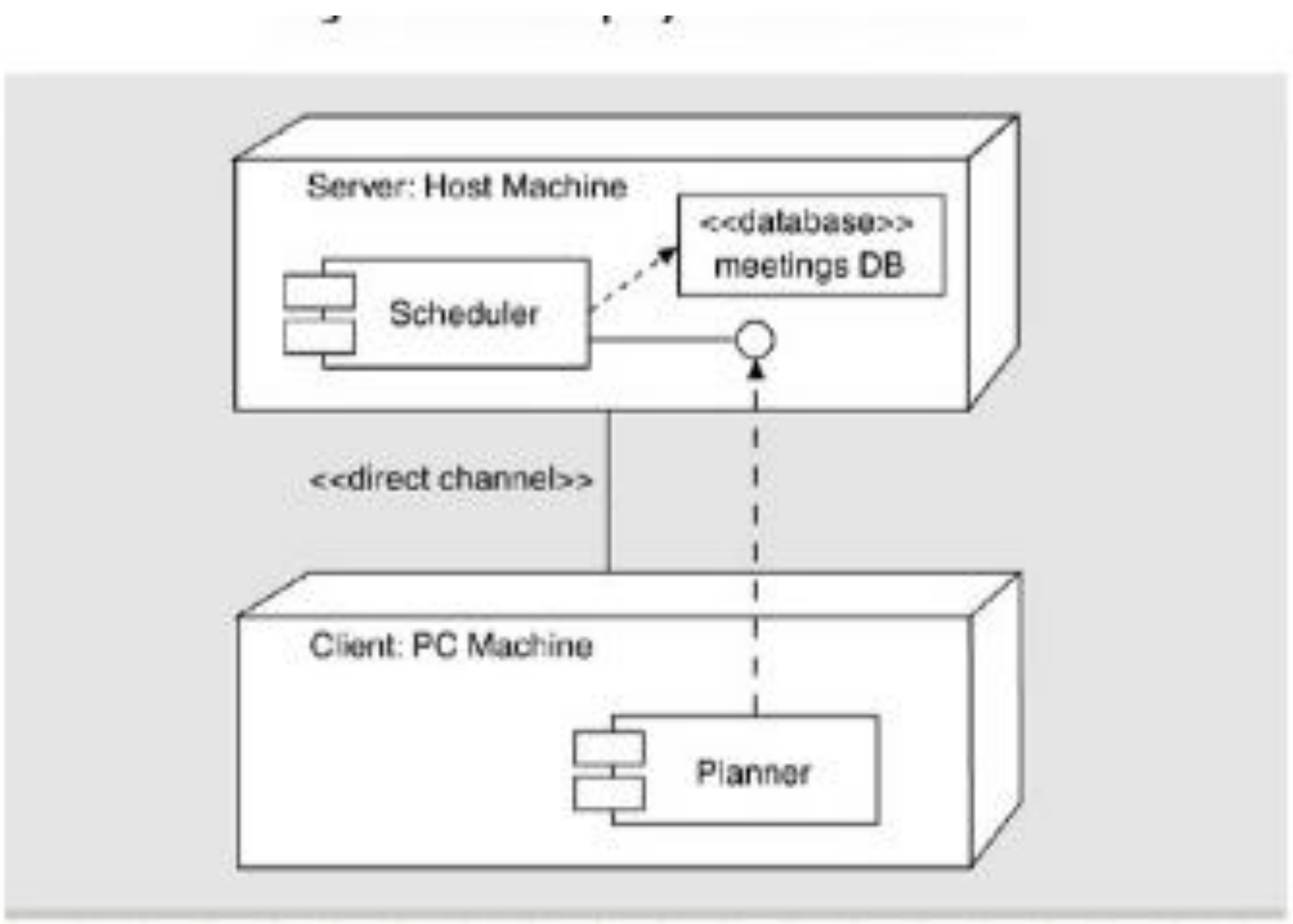
Views…

- Allocation view
  - Focuses on how sw units are allocated to resources like hw, file system, people
  - I.e. specifies relationship between sw elements and execution units in the env
  - Expose structural properties like which process runs on which processor, which file resides where, …

Allocation View

- Elements
  - Software element
  - Some runtime packaging of logical modules and components (e.g., processes)
  - Environmental element - execution (hardware, runtime operation) or development (file structure, deployment, development organization)
  - Properties that are provided to the software; e.g., bandwidth
- Relations
  - Allocated to - a software element is mapped (allocated to) an environmental element
  - Static or dynamic (e.g., resource allocation)
- **UML: Deployment diagrams**

Allocation View



Allocation View

- Usage of Allocation Views
- Specify structure and behavior of runtime elements such as processes, objects, servers, data stores
- Reasoning and decisions about …
  - What hardware and software is needed
  - Distributed development and allocation of work to teams.
  - Builds, integration testing, version control
  - System installation

Views…

- An arch description consists of views of diff types, each showing a diff structure
  - Diff sys need diff types of views depending on the needs
  - E.g. for perf analysis, allocation view is necessary; for planning, module view helps
- The C&C view is almost always done, and has become the primary view
  - We focus primarily on the C&C view
  - Module view is covered in high level design, whose focus is on identifying modules

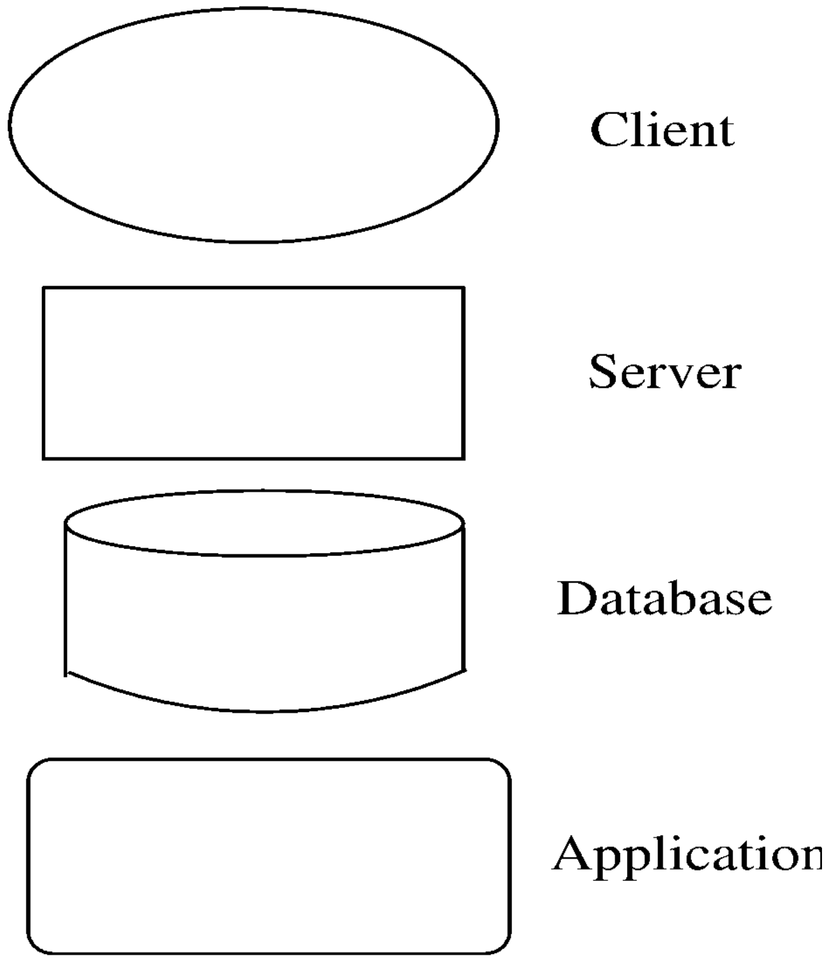
Component and Connector View

- Two main elements – components and connectors
- Components: Computational elements or data stores
- Connectors: Means of interaction between comps
- A C&C view defines the comps, and which comps are connected through which connector
- The C&C view describes a runtime structure of the system – what comps exist at runtime and how they interact during execution
- Is a graph; often shown as a box-and-line drawing
- Most commonly used structure

Components

- Units of computations or data stores
- Has a name, which represents its role, and provides it identity
- A comp may have a type; diff types rep by diff symbols in C&C view
- Comps use ports (or interfaces) to communicate with others
- An arch can use any symbols to rep components; some common ones are shown

Some Component examples…



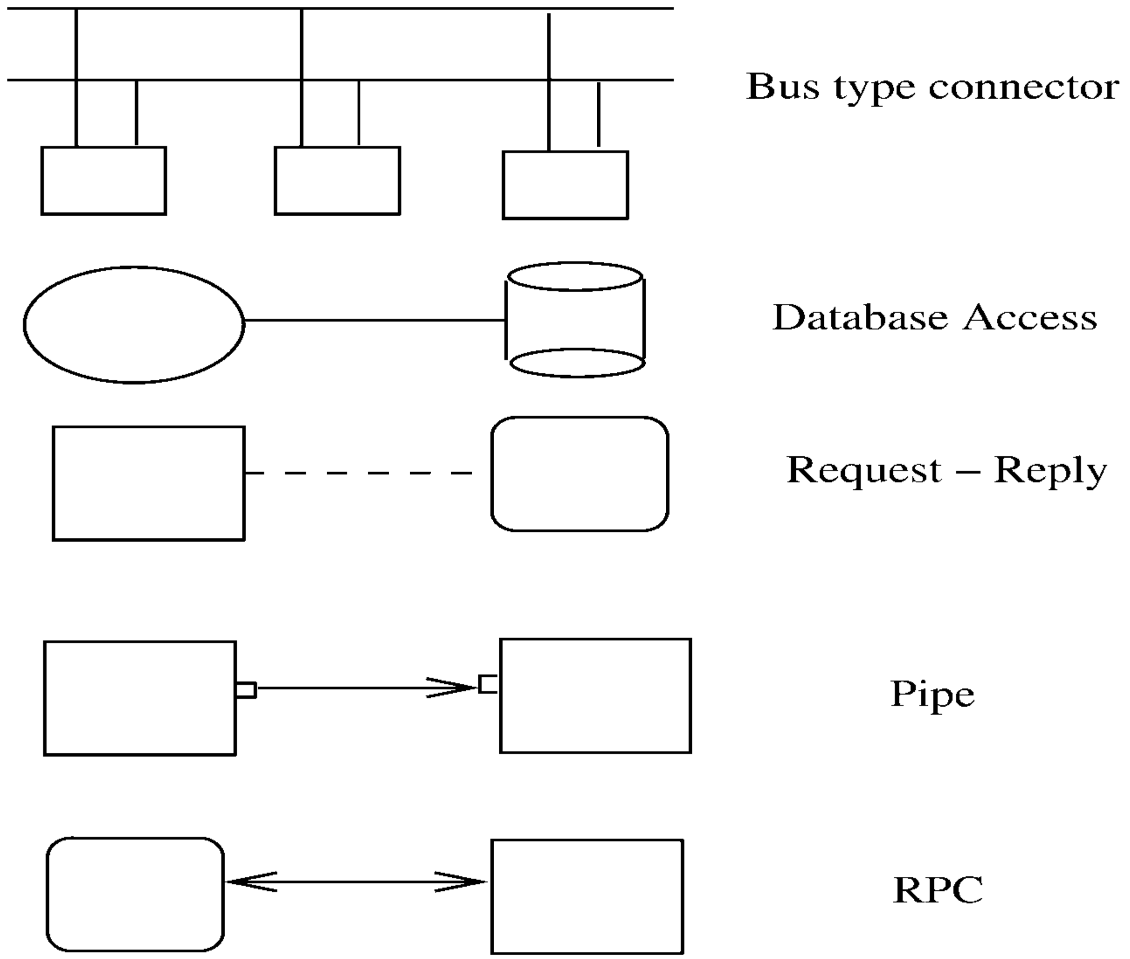
Connectors

- Interaction between components happen through connectors
- A connector may be provided by the runtime environment, e.g. procedure call
- But there may be complex mechanisms for interaction, e.g http, tcp/ip, ports,…; a lot of sw needed to support them
- Important to identify them explicitly; also needed for programming comps properly

Connectors…

- Connectors need not be binary, e.g. a broadcast bus
- Connector has a name (and a type)
- Often connectors represented as protocol – i.e. comps need to follow some conventions when using the connector
- Best to use diff notation for diff types of connectors; all connectors should not be shown by simple lines

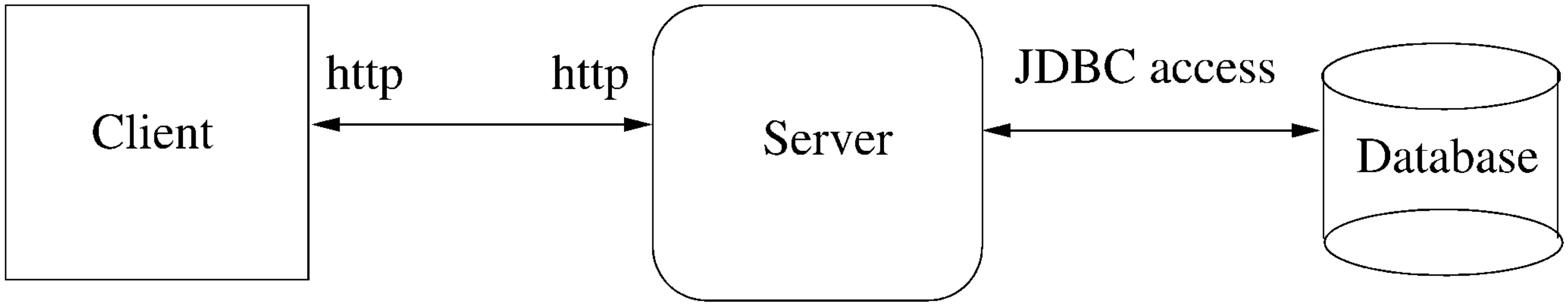
Connector examples



An Example

- Design a system for taking online survey of students on campus
  - Multiple choice questions, students submit online
  - When a student submits, current result of the survey is shown
- Is best built using web; a 3-tier architecture is proposed
  - Has a client, server, and a database components (each of a diff type)
  - Connector between them are also of diff types

Example…

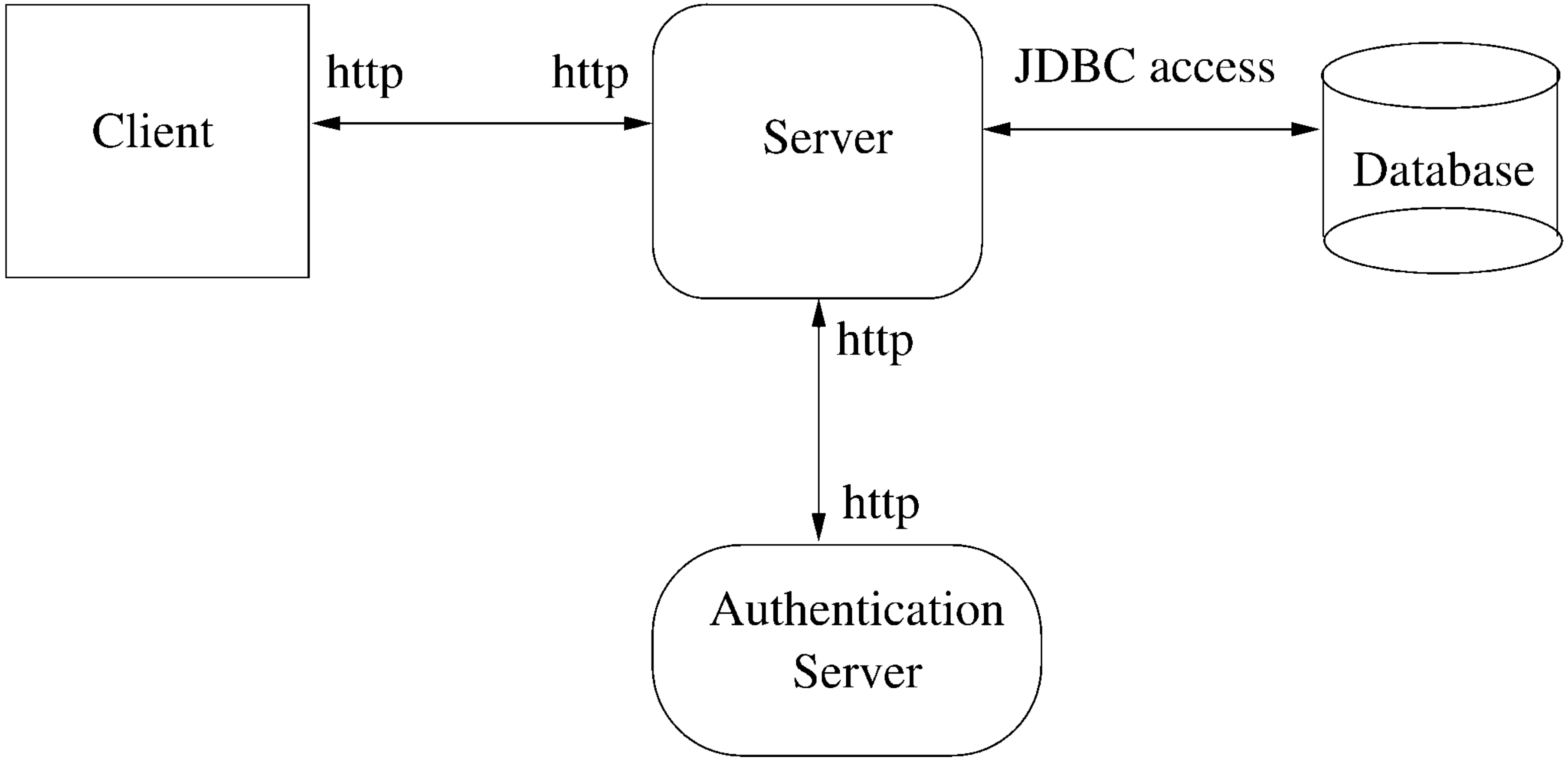


- At arch level, details are not needed
- The connectors are explicitly stated, which implies that the infrastructure should provide http, browser, etc.
- The choice of connectors imposes constraints on how the components are finally designed and built

Extension 1

- This arch has no security – anyone can take the survey
- We want that only registered students can take the survey (at most once)
  - To identify students and check for one-only submission, need a authentication server
  - Need to use cookies, and server has to be built accordingly (the connector between server and authentication server is http with cookies)

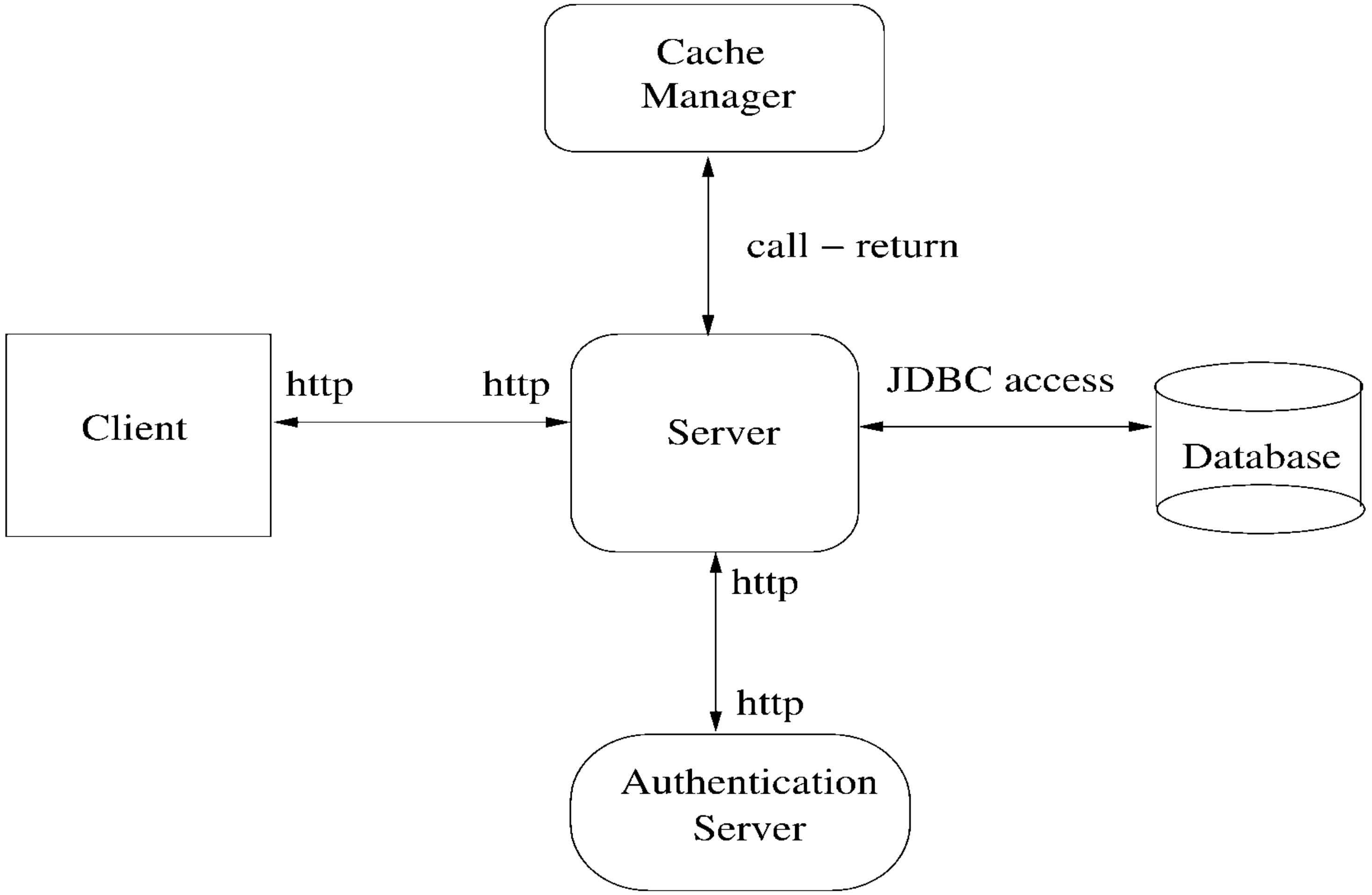
Extension 1…


Client
http
http
Server
JDBC access
Database
http
http
Authentication Server

Extension 2

- It was found that DB is frequently down
- For improving reliability, want that if DB is down, student is given an older survey result and survey data stored
- The survey data given can be outdated by at most 5 survey data points
- For this, will add a cache comp, which will store data as well as results

Extension 2…


Cache Manager
call – return
Client
http
http
Server
JDBC access
Database
http
http
Authentication Server

C&C View

- Usage
  - Major executing components
  - Major shared data stores
  - Runtime interaction; e.g., control and data flow, parallelism
  - Connector mechanisms – e.g., service invocation, asynchronous messaging, event subscription, …
- Constraints
  - All attachments are only between components and connectors
  - Attachments must be between compatible ports and roles

Views..

- **Relating Structures to Each Other**
- Each structure provides a different perspective and design handle on a system
- Each is valid and useful on its own The structures are not independent, just the opposite
- Elements of one will be related to elements of another
- Relationships should be consistent and rational

Element names: meaningful and consistent across views!!

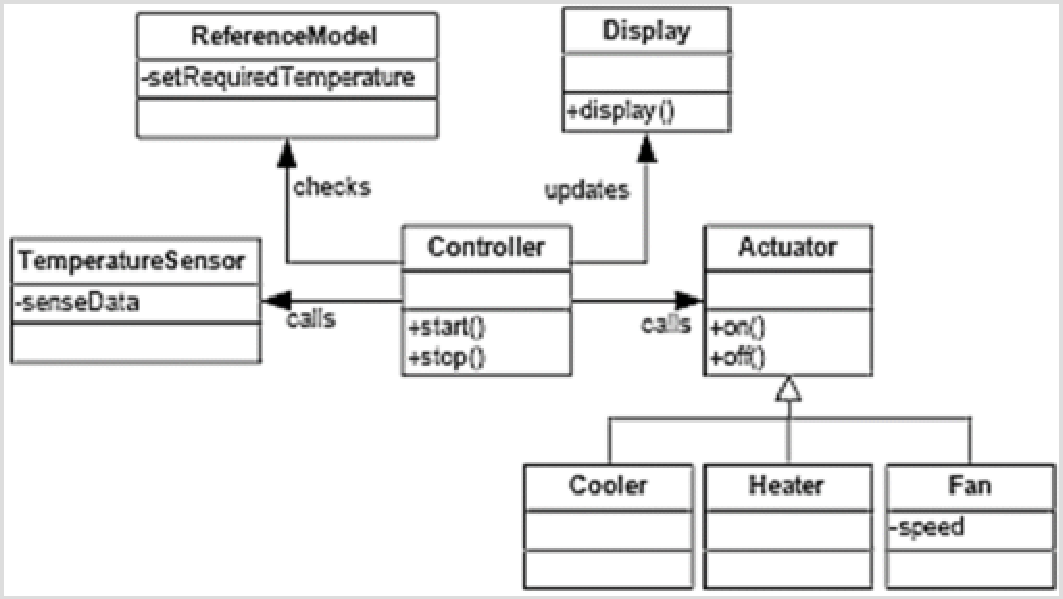
Views..

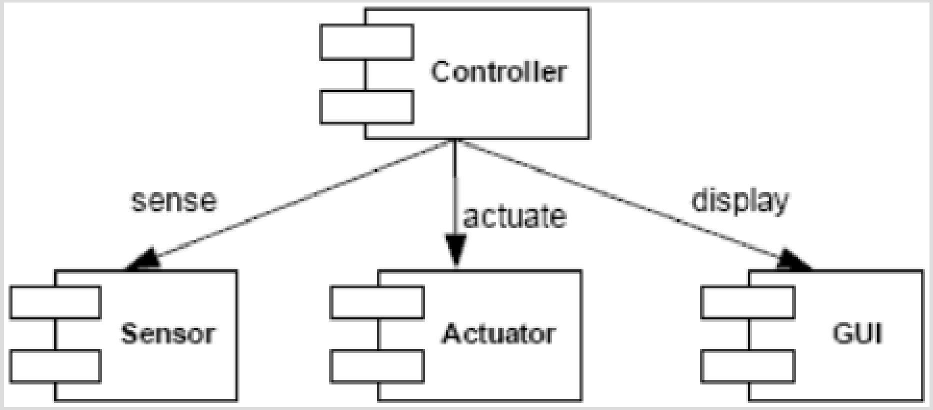
- **Relating Structures to Each Other**

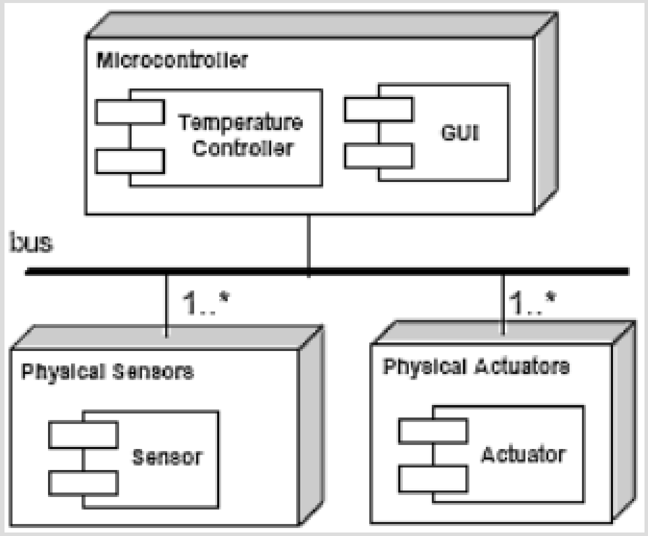
- Example: a code module in a decomposition structure may map to one, part of one, or several run-time components in a componentand-connector structure
- Sometimes, one structure dominates (usually decomposition structure)
- For some systems, some structures may be irrelevant or trivial, such as a single node, single process application

Views..

Relating Structures to Each Other







Views..

Which Views? The Ones You Need!

- Different views support different goals and uses
- The views you document depend on the stakeholders and uses of the documentation.
- Each view has a cost and a benefit; the benefits of maintaining a view should outweigh its costs
- At a minimum, at least on module view and one component and connector view

Pipe and filter style

- Well suited for systems that mainly do data transformations
- A system using this style uses a network of transforms to achieve the desired result
- Has one component type – filter
- Has one connector type – pipe
- A filter does some transformation and passes data to other filters through pipes

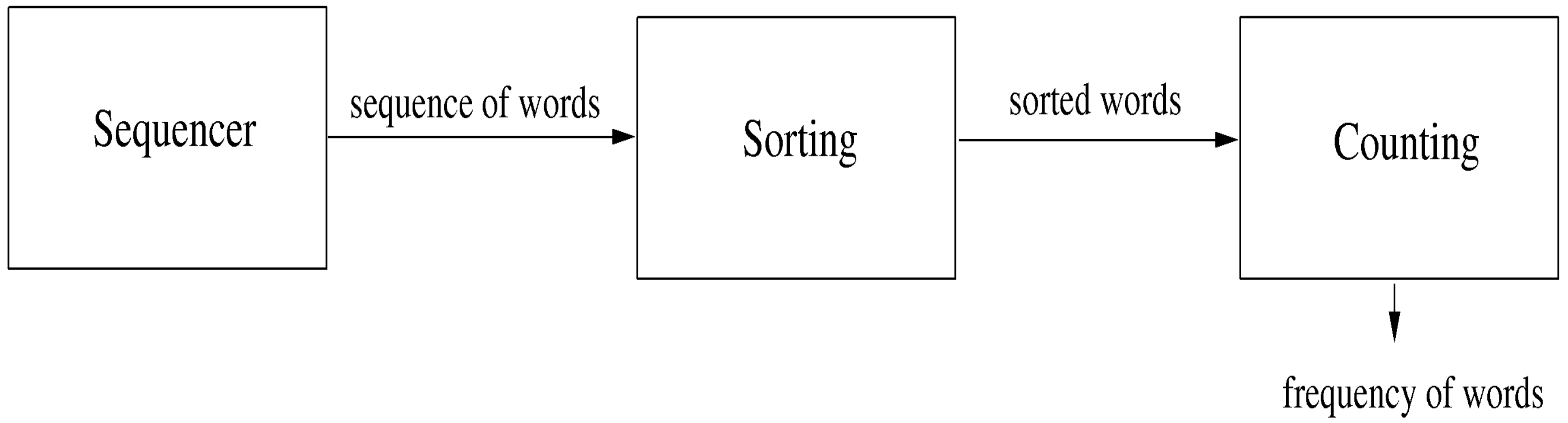
Pipe and Filter style

- A filter is independent; need not know the id of filters sending/receiving data
- Filters can be asynchronous and are producers or consumers of data
- A pipe is unidirectional channel which moves streams of data from one filter to another
- A pipe is a 2-way connector
- Pipes have to perform buffering, and synchronization between filters
- Pipes should work without knowing the identify of producers/consumers
- A pipe must connect the output port of one filter to input port of another
- Filters may have independent thread of control

Example

- A system needed to count the frequency of different words in a file
- One approach: first split the file into a sequence of words, sort them, then count the #of occurrences
- The arch of this system can naturally use the pipe and filter style

Example..


Sequencer
sequence of words
Sorting
sorted words
Counting
frequency of words

Shared-data style

- Two component types – data repository and data accessor
- Data repository – provides reliable permanent storage
- Data accessors – access data in repositories, perform computations, and may put the results back also
- Communication between data accessors is only through the repository

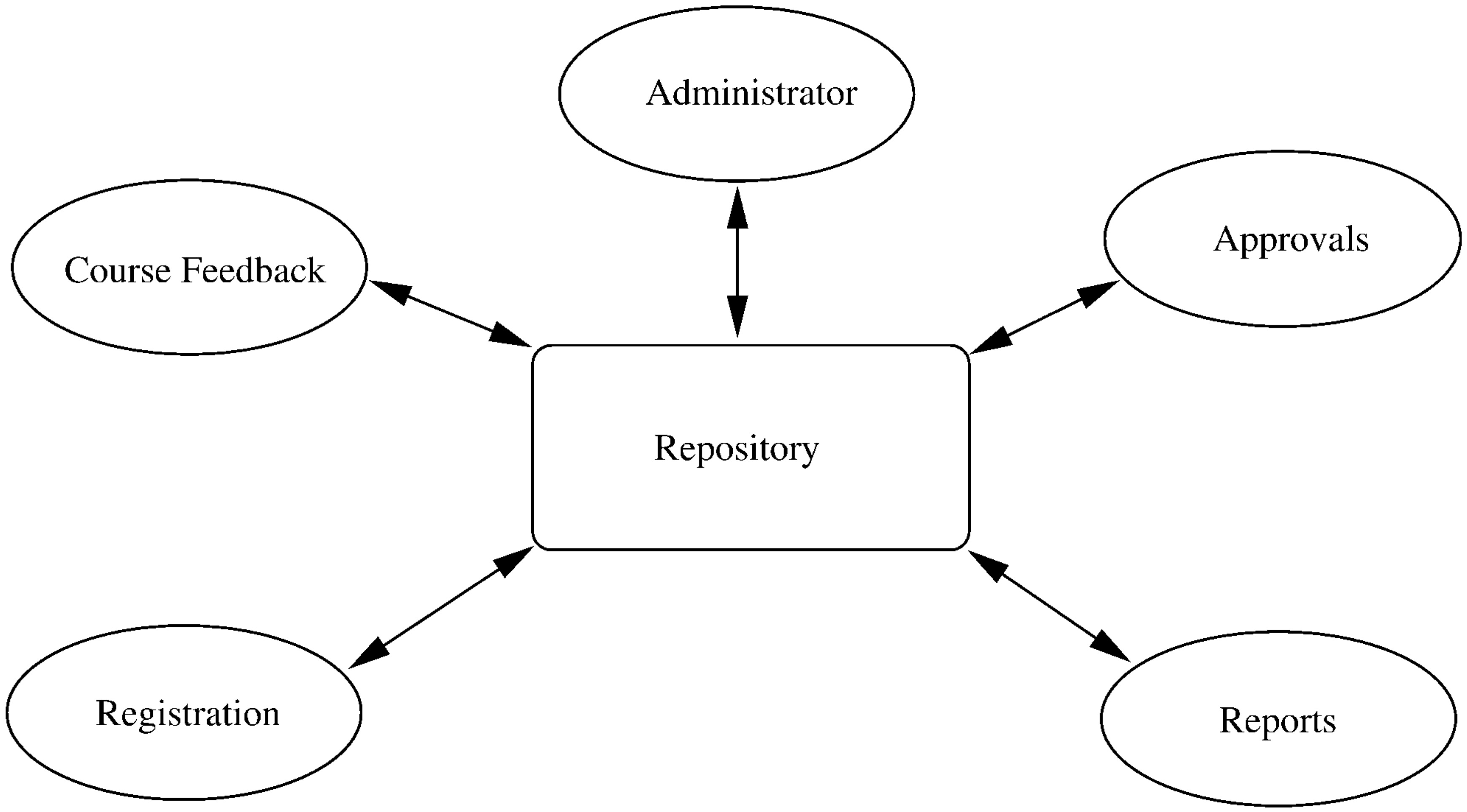
Shared-data style…

- Two variations possible
  - Black board style: if data is posted in a repository, all accessors are informed; i.e. shared data source is an active agent
  - Repository style: passive repository
- Eg. database oriented systems; web systems; programming environments,..

Example

- A student registration system of a university
- Repository contains all the data about students, courses, schedules,…
- Accessors like admin, approvals, registration, reports which perform operations on the data

Example…


Administrator
Course Feedback
Approvals
Repository
Registration
Reports

Example..

- Components do not directly communicate with each other
- Easy to extend – if a scheduler is needed, it is added as a new accessor
  - No existing component needs to be changed
- Only one connector style in this – read/write

Client-Server Style

- Two component types – clients and servers
- Clients can only communicate with the server, but not with other clients
- Communication is initiated by a client which sends request and server responds
- One connector type – request/reply, which is asymmetric
- Often the client and the servers reside on different machines

Client-server style…

- A general form of this style is the n-tier structure
- A 3-tier structure is commonly used by many application and web systems
  - Client-tier contains the clients
  - Middle-tier contains the business rules
  - Database tier has the information

Some other styles

- Publish-subscribe style
  - Some components generate events, and others subscribe to them
  - On an event, those component that subscribe to it are invoked
- Peer-to-peer style
  - Like object oriented systems; components use services from each other through methods
- Communication processes style
  - Processes which execute and communicate with each other through message passing

Architecture and Design

- Both arch and design partition the system into parts and their org
- What is the relationship between design and arch?
  - Arch is a design; it is about the solution domain, and not problem domain
  - Can view arch as a very high level design focusing on main components
  - Design is about modules in these components that have to be coded
  - Design can be considered as providing the module view of the system

Contd…

- Boundaries between architecture and design are not clear or hard
- It is for designer and architect to decide where arch ends and design begins
- In arch, issues like files, data structure etc are not considered, while they are important in design
- Arch does impose constraints on design in that the design must be consistent with arch

Preserving the Integrity of Architecture

- What is the role of arch during the rest of the development process
- Many designers and developers use it for understanding but nothing more
- Arch imposes constraints; the implementation must preserve the arch
- I.e. the arch of the final system should be same as the arch that was conceived
- It is very easy to ignore the arch design and go ahead and do the development
- Example – impl of the word frequency problem

Documenting Arch Design

- While designing and brainstorming, diagrams are a good means
- Diagrams are not sufficient for documenting arch design
- An arch design document will need to precisely specify the views, and the relationship between them

Documenting…

- An arch document should contain
  - System and architecture context
  - Description of architecture views
  - Across view documentation
- A context diagram that establishes the sys scope, key actors, and data sources/sinks can provide the overall context
- A view description will generally have a pictorial representation, as discussed earlier

Documenting…

- Pictures should be supported by
  - Element catalog: Info about behavior, interfaces of the elements in the arch
  - Architectural rationale: Reasons for making the choices that were made
  - Behavior: Of the system in different scenarios (e.g. collaboration diagram)
  - Other Information: Decisions which are to be taken, choices still to be made,..

Documenting...

- Inter-view documentation
  - Views are related, but the relationship is not clear in the view
  - This part of the doc describes how the views are related (eg. How modules are related to components)
  - Rationale for choosing the views
  - Any info that cuts across views
- Sometimes views may be combined in one diagram for this – should be done if the resulting diagram is still easy to understand

Evaluating Architectures

- Arch impacts non-functional attributes like modifiability, performance, reliability, portability, etc
  - Attr. like usability etc are not impacted
- Arch plays a much bigger impact on these than later decisions
- So should evaluate a proposed arch for these properties
- Q: How should this evaluation be done?
  - Many different ways

Evaluating Architectures…

- Procedural approach – follow a sequence of steps
  - Identify the attributes of interest to different stakeholders
  - List them in a table
  - For each attribute, evaluate the architectures under consideration
  - Evaluation can be subjective based on experience
  - Based on this table, then select some arch or improve some existing arch for some attribute

Software Design

- Design activity begins with a set of requirements, and maybe an architecture
- Design done before the system is implemented
- Design focuses on module view – i.e. what modules should be in the system
- Module view may have easy or complex relationship with the C&C view
- Design of a system is a blue print for implementation
- Often has two levels – high level (modules are defined), and detailed design (logic specified)

Design Concepts

- Design is correct, if it will satisfy all the requirements and is consistent with architecture
- Of the correct designs, we want *best design*
- We focus on modularity as the main criteria (besides correctness)

A Classification of Design Methodologies

- Procedural (Function-oriented)
- Object-oriented
- More recent:
  - Aspect-oriented
  - Component-based (Client-Server)

Analysis versus Design

- An analysis technique helps elaborate the customer requirements through careful thinking:
  - And at the same time consciously avoids making any decisions regarding implementation.
- The design model is obtained from the analysis model through transformations over a series of steps:
  - Decisions regarding implementation are consciously made.

Does a Design Technique Lead to a Unique Solution?

- No:
  - Several subjective decisions need to be made to trade off among different parameters.
  - Even the same designer can come up with several alternate design solutions.

Good and Bad Designs

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

Which of Two is a Better Design?

- Should implement all functionalities of the system correctly.
- Should be easily understandable.
- Should be efficient.
- Should be easily amenable to change,
  - i.e. easily maintainable.

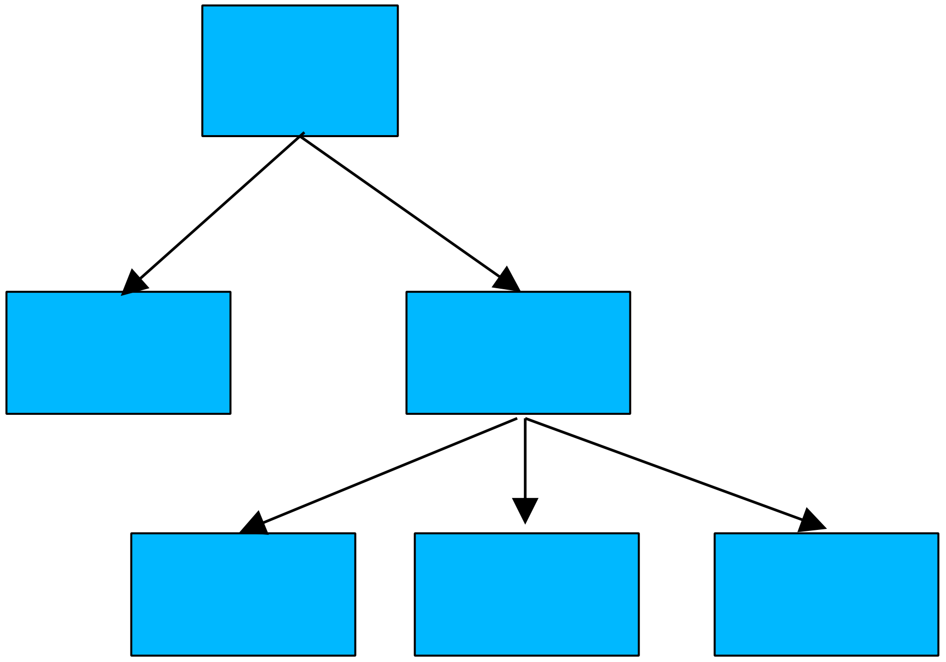
Which of Two is a Better Design?

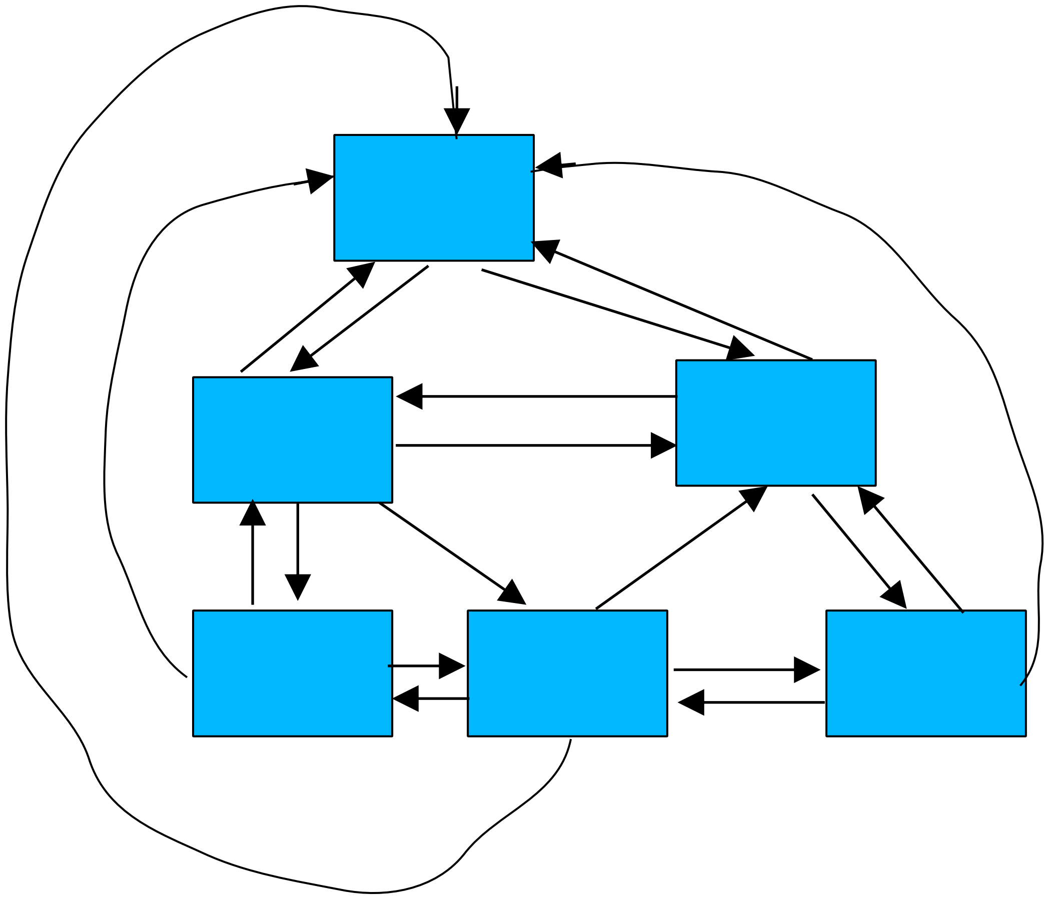
- Understandability of a design is a major issue:
  - Determines goodness of design:
  - A design that is easy to understand:
    - Also easy to maintain and change.
- Unless a design is easy to understand,
  - Tremendous effort needed to maintain it
  - We already know that about 60% effort is spent in maintenance.
- If the software is not easy to understand:
  - Maintenance effort would increase many times.

How are Abstraction and Decomposition Principles Used in Design?

- Two principal ways:
  - Modular Design
  - Layered Design
- **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.
- If modules are independent:
  - Modules can be understood separately,
    - Reduces the complexity greatly.

Layered Design





Neat arrangement of modules in a hierarchy means:

Low fan-out

Control abstraction

Cohesion and Coupling

- In technical terms, modules should display:
  - High cohesion
  - Low 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 the 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:**
- 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.
- 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.

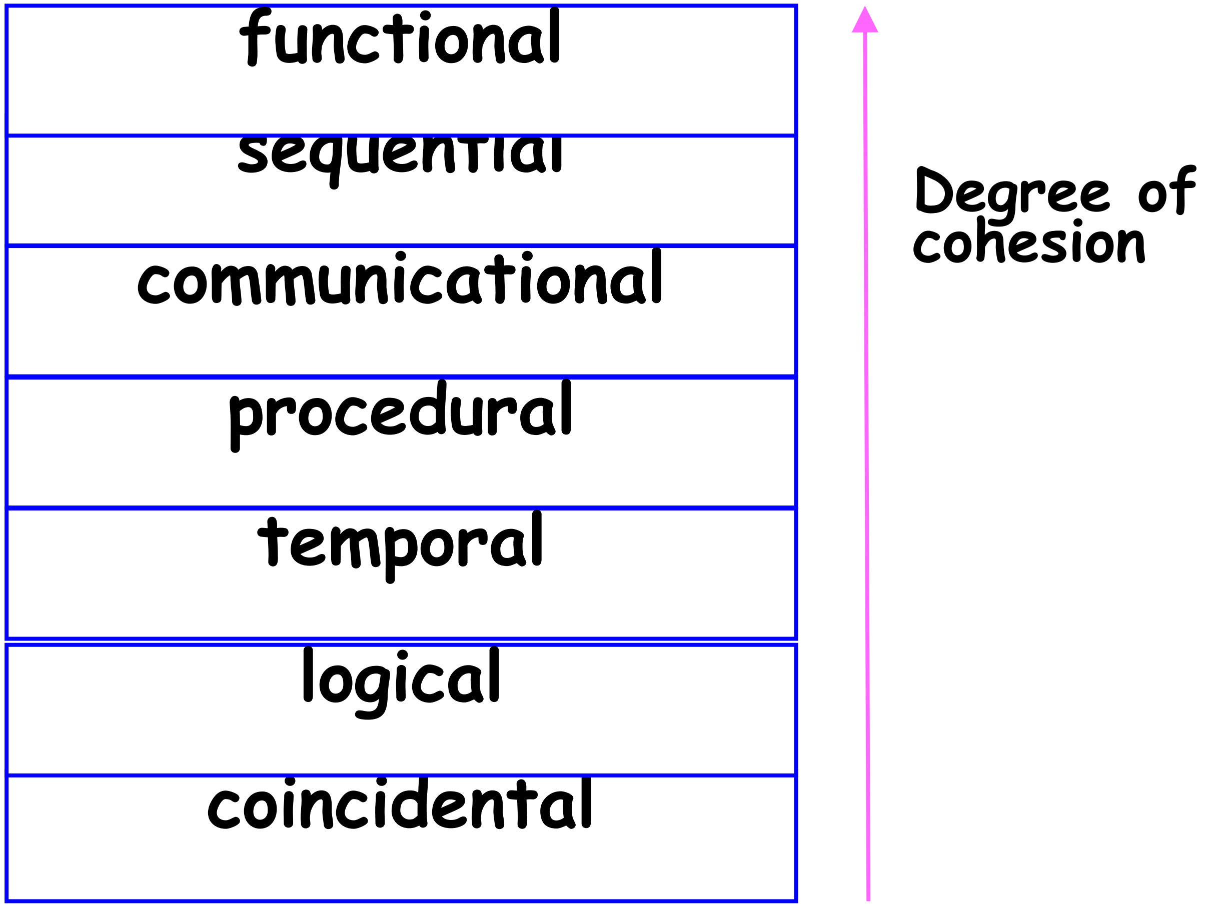
Functional Independence

- Unfortunately, there are no ways:
  - To quantitatively measure the degree of cohesion and coupling.
  - Classification of different kinds of cohesion and coupling:
    - Can 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.

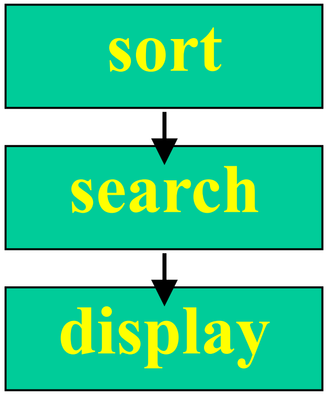
Temporal Cohesion

- 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.



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.

Determining Cohesiveness

- Write down a sentence to describe the function of the module
  - If the sentence has to be a compound sentence, contains more than one verbs, the module is probably performing more than one function. Probably has **sequential or communicational cohesion.**
  - If the sentence contains words relating to time, like "first", "next", "after", "start" etc., the module probably has **sequential or temporal cohesion**.
- If it has words like initialize, It probably has **temporal cohesion**.
- If the predicate of the sentence does not contain a single specific object following the verb, the module is probably logically cohesive. Eg "edit all data", while "edit source data" may have **functional cohesion**.
- Functionally cohesive module can always be described by a simple statement.

Cohesion in OO Systems

- In OO, different types of cohesion is possible as classes are the modules
  - Method cohesion
  - Class cohesion
  - Inheritance cohesion
- Method cohesion – why diff code elements are together in a method
  - Like cohesion in functional modules; highest form is if each method implements a clearly defined function with all elements contributing to implementing this function

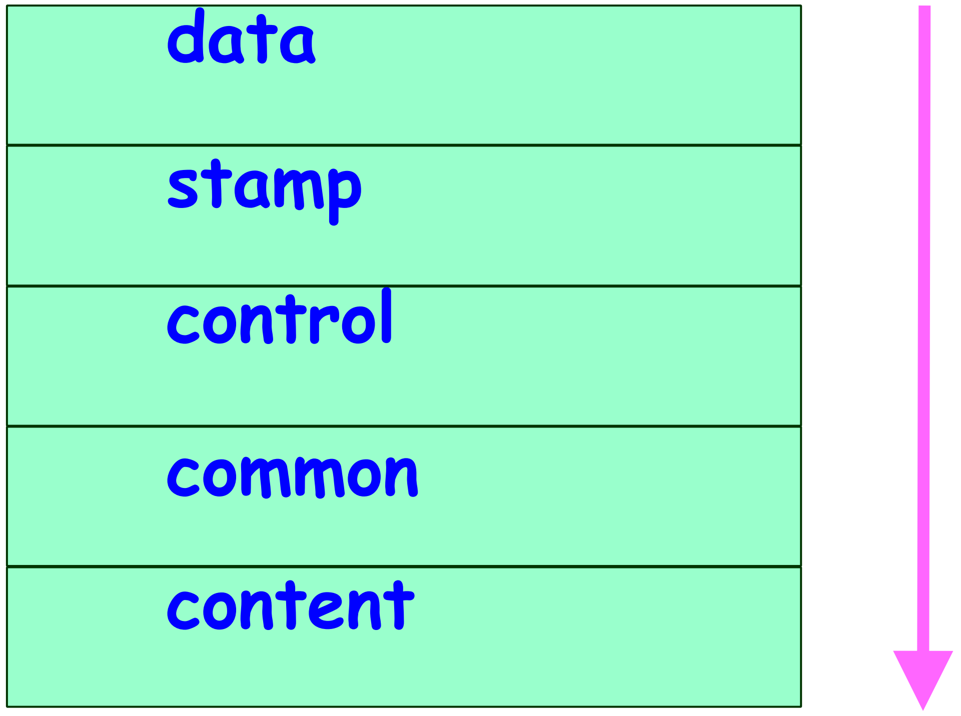
Cohesion in OO…

- Class cohesion – why diff attributes and methods are together in a class
  - A class should represent a single concept with all elts contributing towards it
  - Whenever multiple concepts encapsulated, cohesion is not as high
  - A symptom of multiple concepts – diff gps of methods accessing diff subsets of attributes
- Inheritance cohesion – focuses on why classes are together in a hierarchy
  - Two reasons for subclassing – generalization-specialization and reuse
  - Cohesion is higher if the hierarchy is for providing generalization-specialization

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.
- 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



Degree 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 stamp coupled,
  - 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 common coupled,
  - 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.

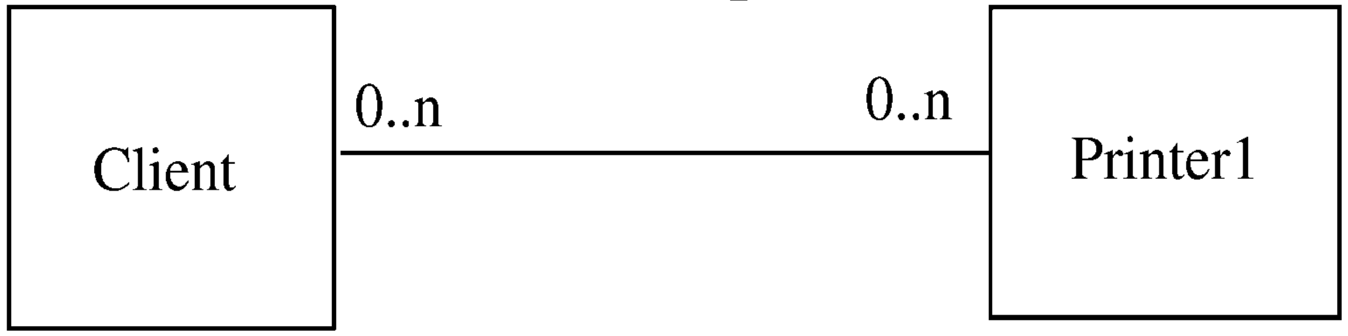
Open-closed Principle

- Besides cohesion and coupling, open closed principle also helps in achieving modularity
- Principle: A module should be open for extension but closed for modification
  - Behavior can be extended to accommodate new requirements, but existing code is not modified
  - I.e. allows addition of code, but not modification of existing code
  - Minimizes risk of having existing functionality stop working due to changes – a very important consideration while changing code
  - Good for programmers as they like writing new code

Open-closed Principle…

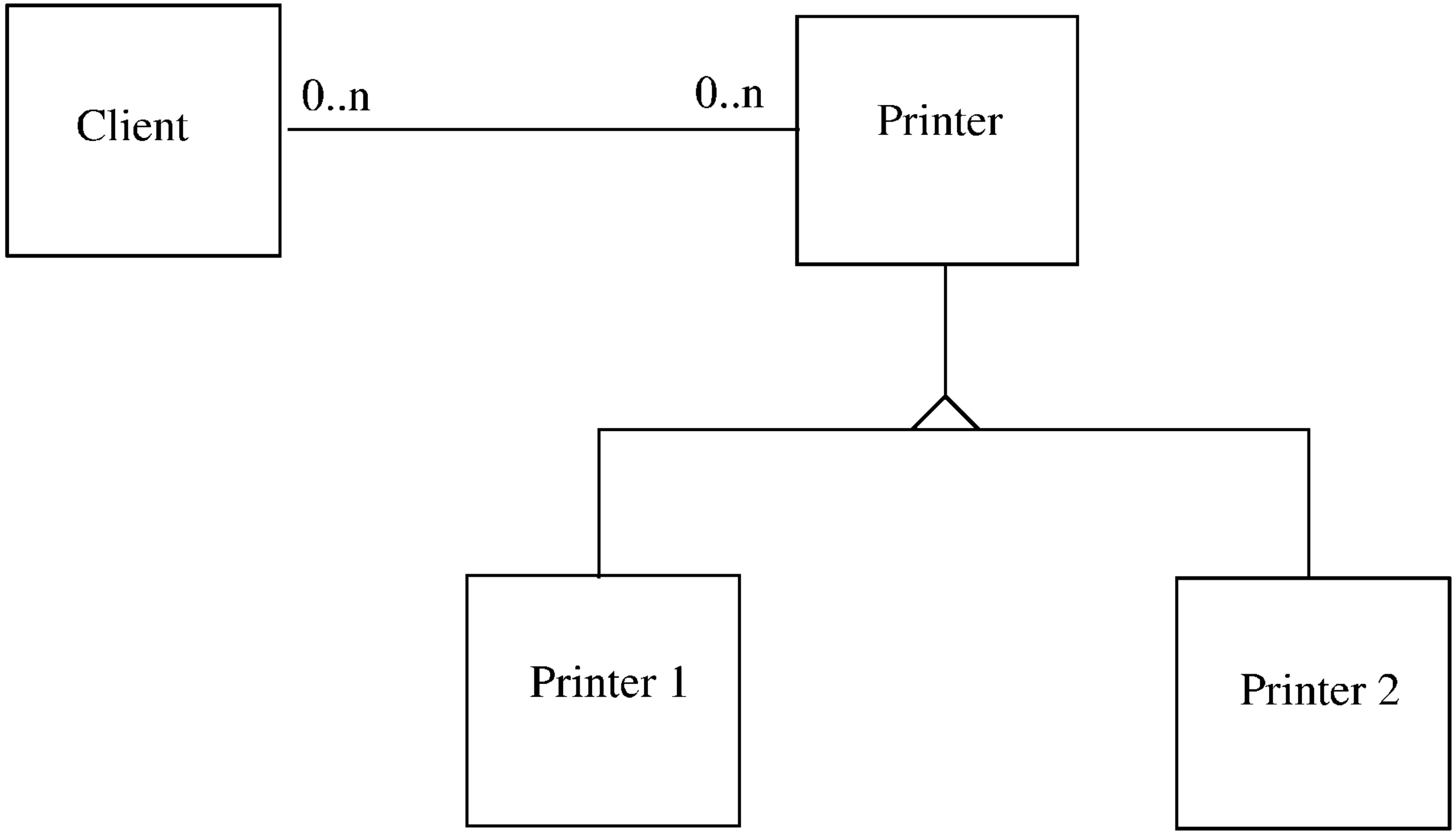
- In OO this principle is satisfied by using inheritance and polymorphism
- Inheritance allows creating a new class to extend behavior without changing the original class
- This can be used to support the open-closed principle
- Consider example of a client object which interacts with a printer object for printing

Example..



- Client directly calls methods on Printer1
- If another printer is to be allowed
  - A new class Printer2 will be created
  - But the client will have to be changed if it wants to use Printer 2
- Alternative approach
  - Have Printer1 a subclass of a general Printer
  - For modification, add another subclass Printer 2
  - Client does not need to be changed

Example…



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.
- Essentially means:
  - Low fan-out
  - Control abstraction

Characteristics of Module Hierarchy

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

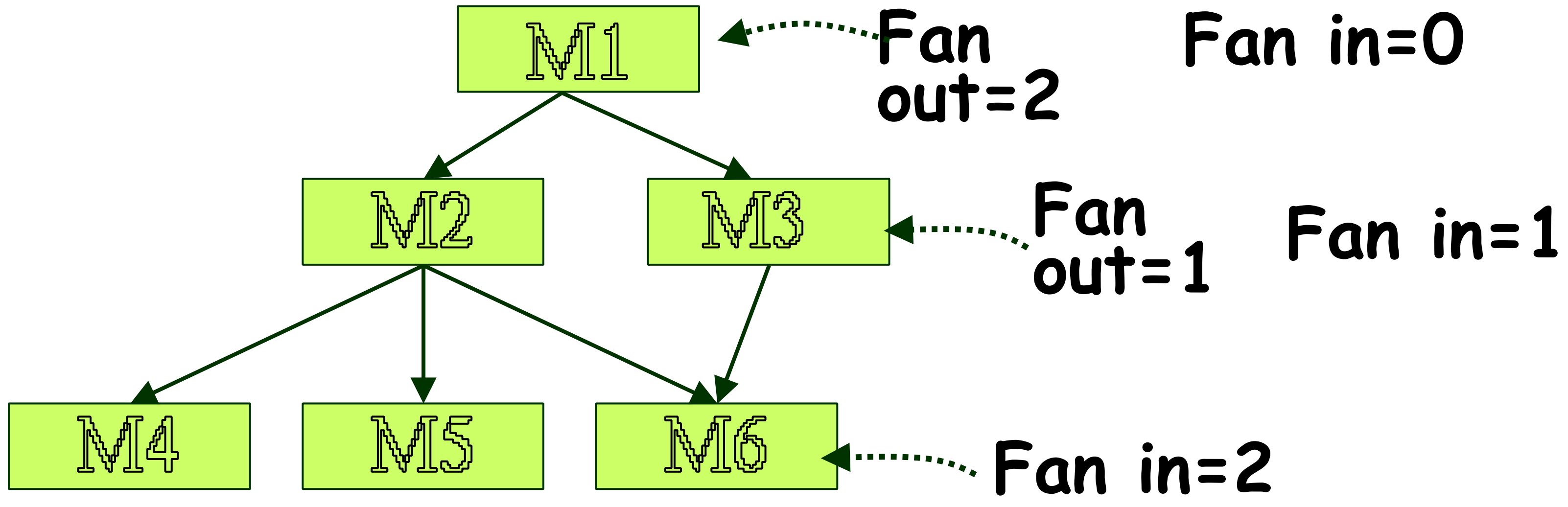
Control Relationships

- 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.

Characteristics of Module Structure

- The **fan-out** of a module is the number of its immediately subordinate modules.
- As a rule of thumb, the optimum fan-out is seven, plus or minus 2.
- The **fan-in** of a module is the number of its immediately superordinate (i.e., parent or boss) modules.
- The designer should strive for high fan-in at the lower levels of the hierarchy.
- High fan-in can also increase portability.

Module Structure



Layered Design

- A design having modules:
  - With high fan-out numbers is not a good design:
  - A module having high fan-out lacks cohesion.
- A module that invokes a large number of other modules:
  - Likely to implement several different functions:
  - Not likely to perform a single cohesive function.

Goodness of Design

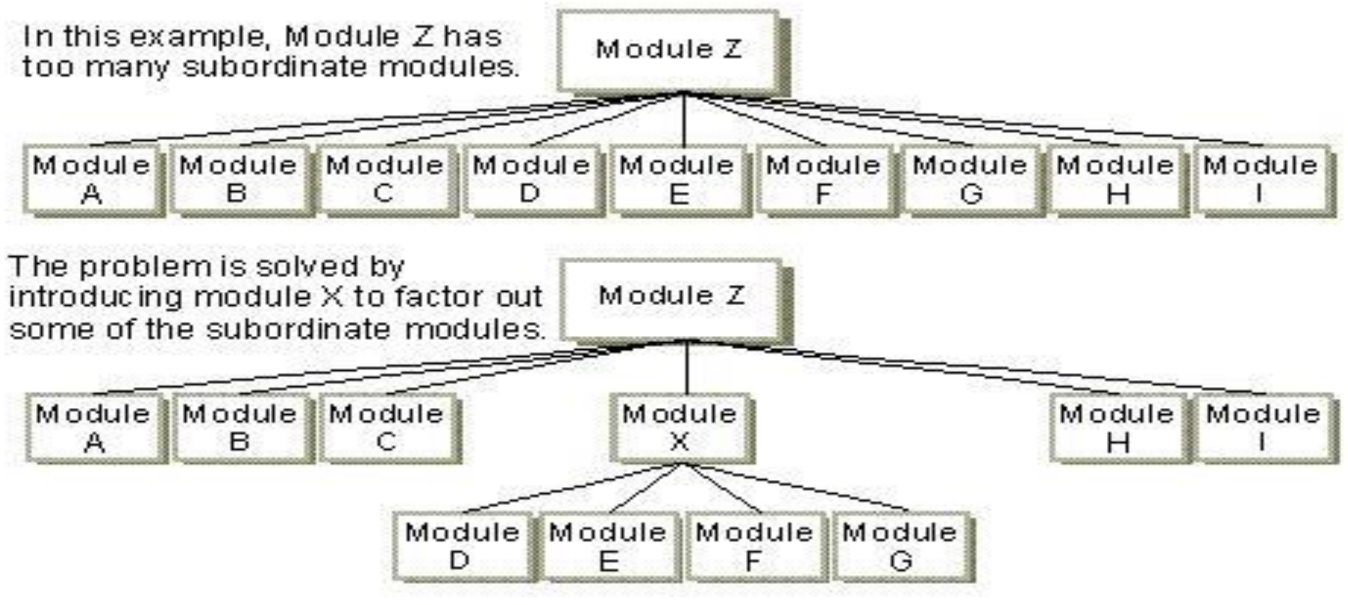
- **Object-Oriented Considerations**
- In object-oriented systems, fan-in and fan-out relate to interactions between objects.
- In object-oriented design, high fan-in generally contributes to a better design of the overall system.
- High fan-in shows that an object is being used extensively by other objects, and is indicative of re-use.
- High fan-out in object-oriented design is indicated when an object must deal directly with a large number of other objects.
- This is indicative of a high degree of class interdependency.
- In general, the higher the fan-out of an object, the poorer is the overall system design.

Goodness of Design

- **Designing Modules That Consider Fan-In/Fan-Out**

- The designer should strive for software structure with **moderate fan-out in the upper levels** of the hierarchy and **high fan-in in the lower levels of the hierarchy**.

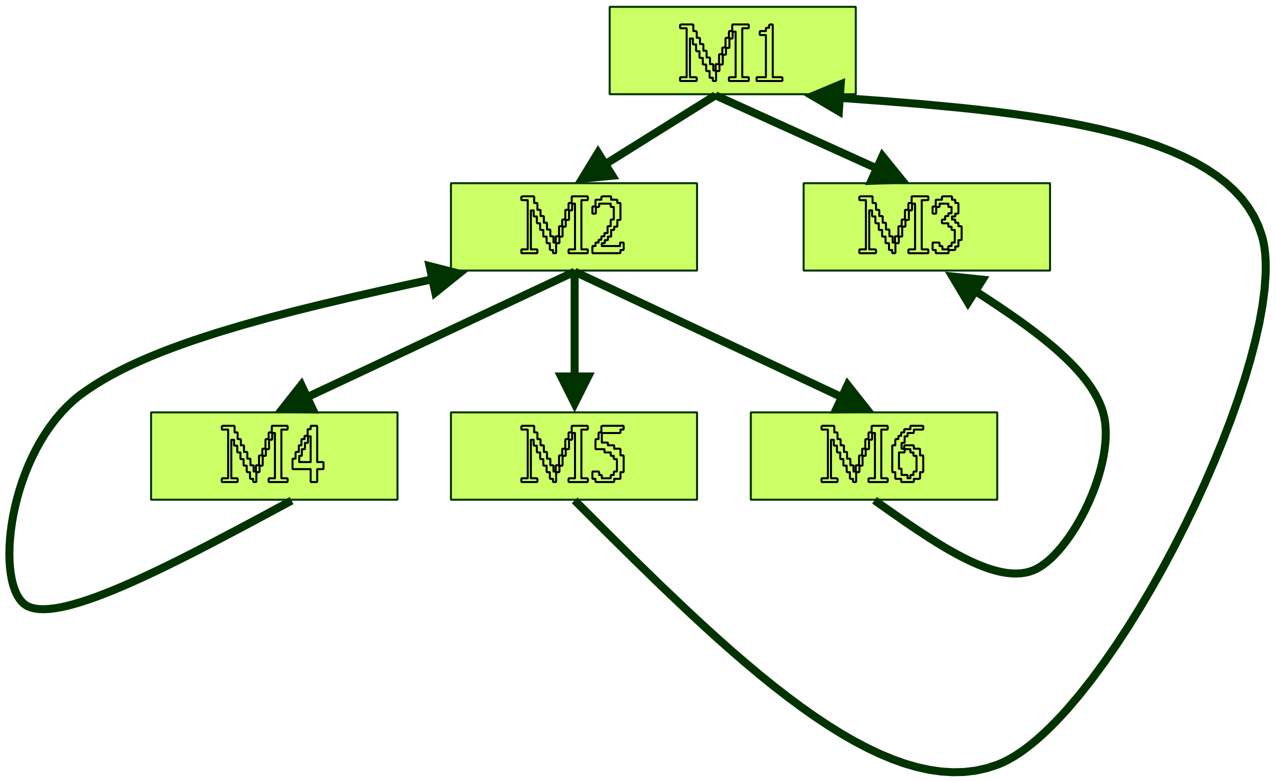
Example of a Solution to Excessively High Fan-Out



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

- A module is unaware (how to invoke etc.) of the higher level modules.
- Lower-level modules:
  - Do input/output and other low-level functions.
- Upper-level modules:
  - Do more managerial functions.
- The principle of abstraction requires:
  - Lower-level modules do not invoke functions of higher level modules.
  - Also known as layered design.

Design Approaches

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

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.
- During the design process:
  - High-level functions are successively decomposed:
    - Into more detailed functions. (Top-Down approach)
  - Finally the detailed functions are mapped to a module structure.

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

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.

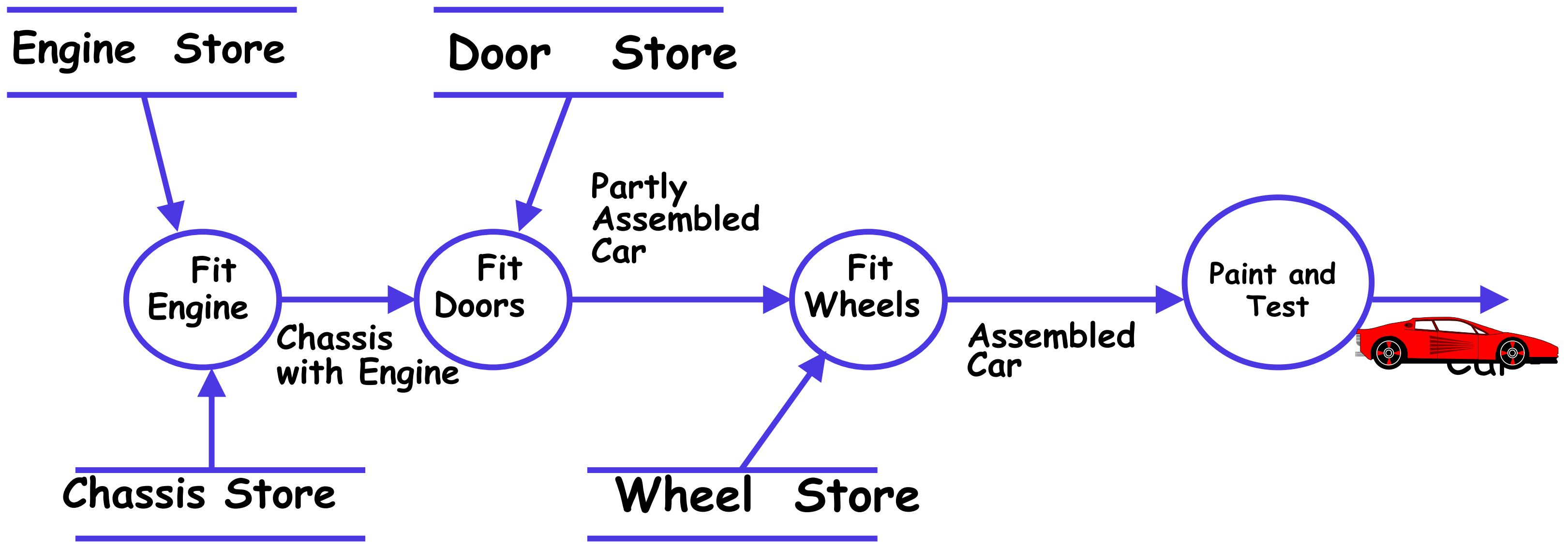
Structured Analysis

- 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 Diagram

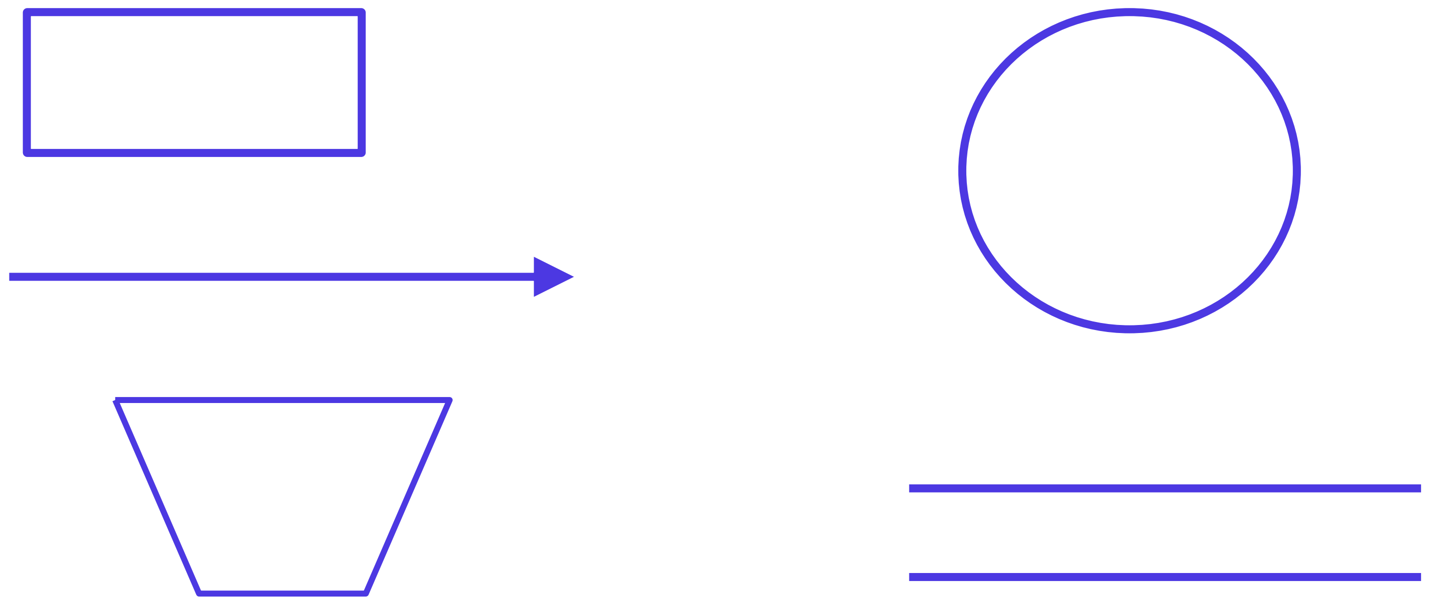
- DFD is a hierarchical graphical model:
  - Shows the different functions (or processes) of the system and
  - Data interchange among the processes.
- It is useful to consider each function as a processing station:
  - Each function consumes some input data.
  - Produces some output data.

Data Flow Model of a Car Assembly Unit (Example)



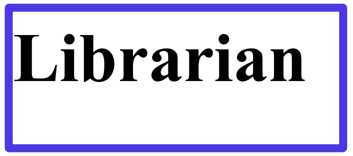
Data Flow Diagrams (DFDs)

- Primitive Symbols Used for Constructing DFDs:



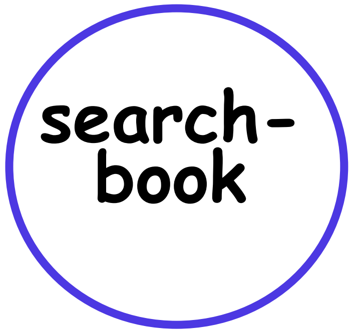
External Entity Symbol

- 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.



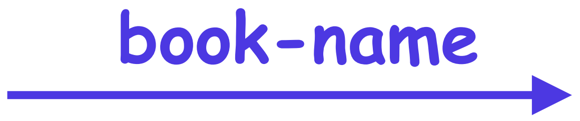
Function Symbol

- A function such as "search-book" is represented using a circle:
  - This symbol is called a process or bubble or transform.
  - Bubbles are annotated with corresponding function names.
  - Functions represent some activity:
    - Function names should be verbs.



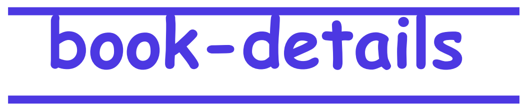
Data Flow Symbol

- 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.



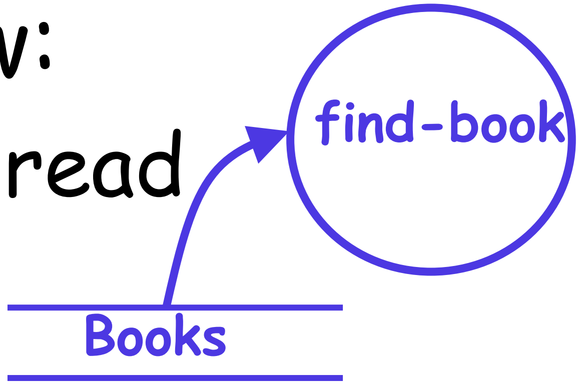
Data Store Symbol

- 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

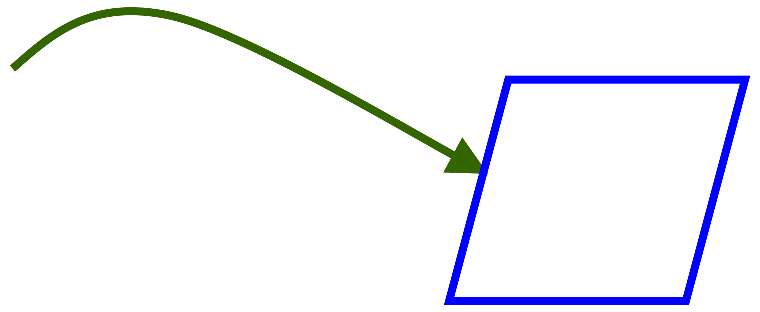
- Direction of data flow arrow:
  - Shows whether data is being read from or written into it.



- 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.

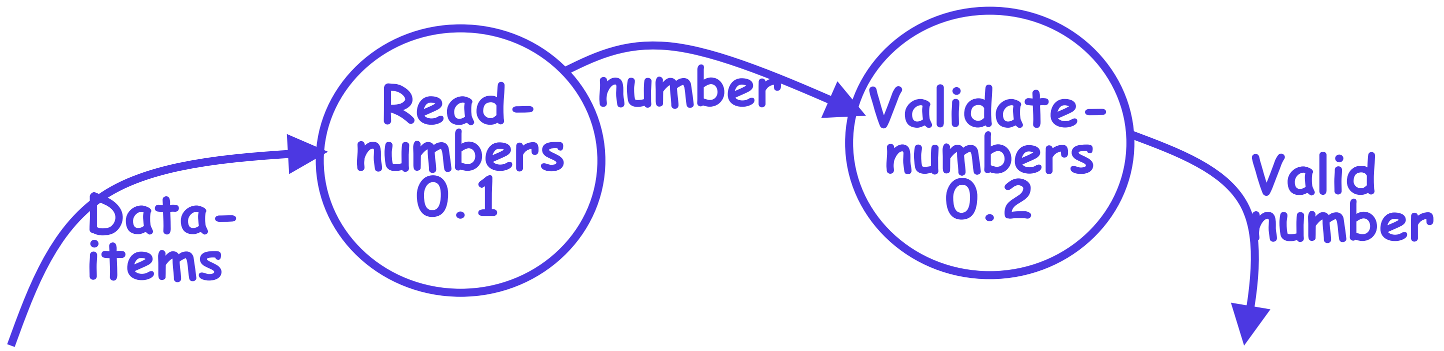
Output Symbol

- Output produced by the system



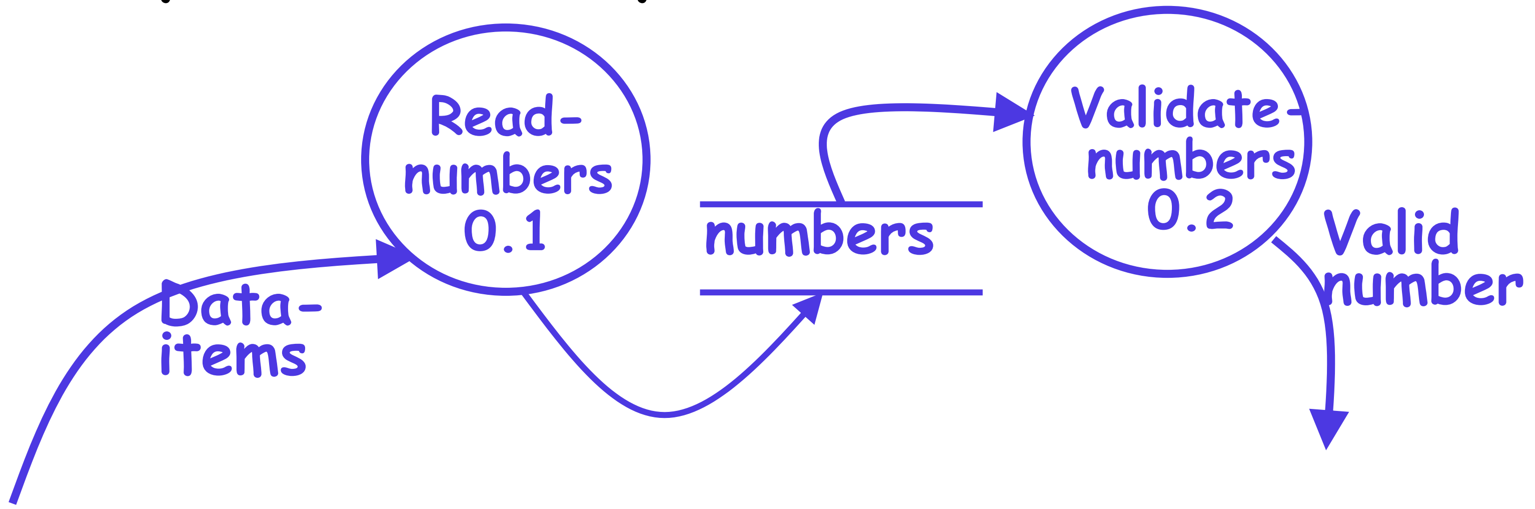
Synchronous Operation

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



Asynchronous Operation

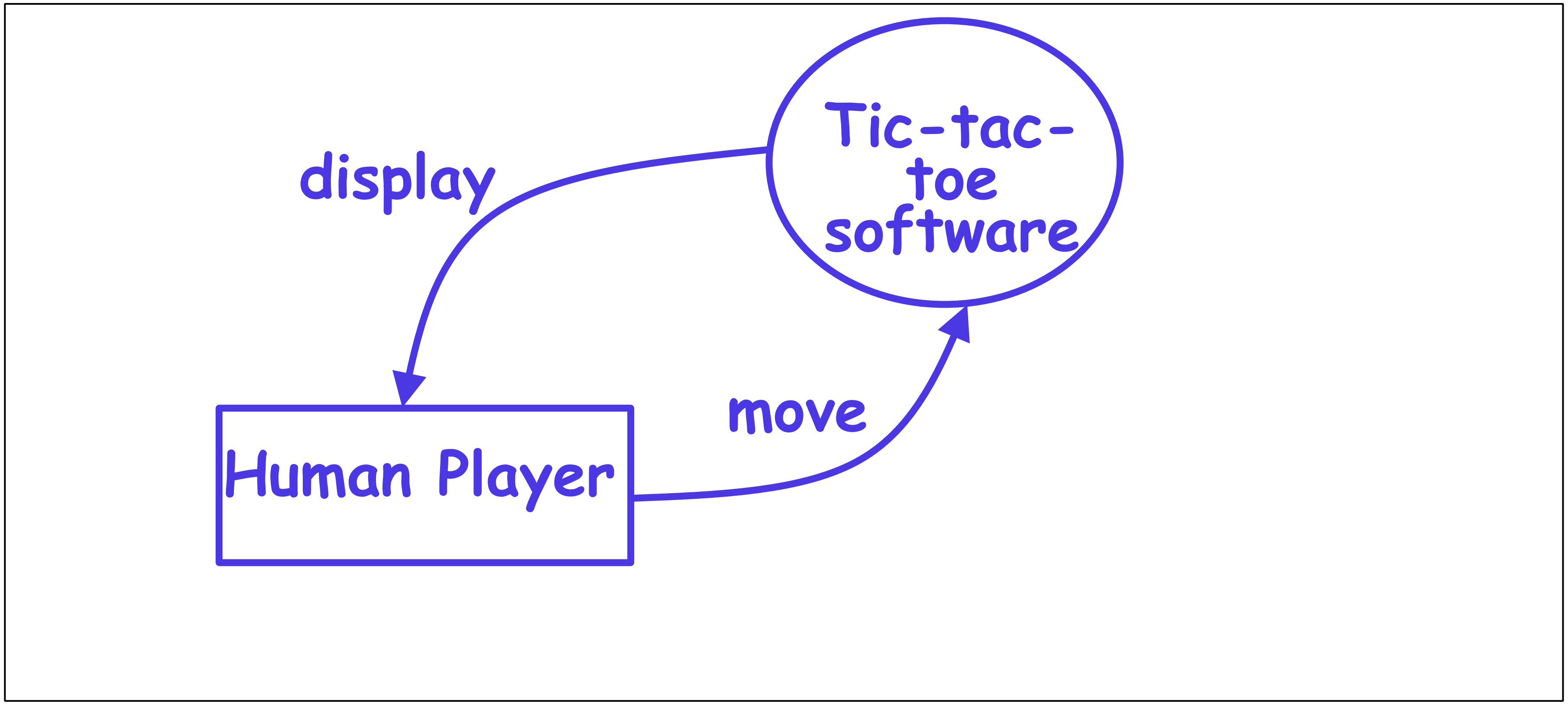
- If two bubbles are connected via a data store:
  - They are not synchronous.



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



- 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

- 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.
- 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.
- 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.

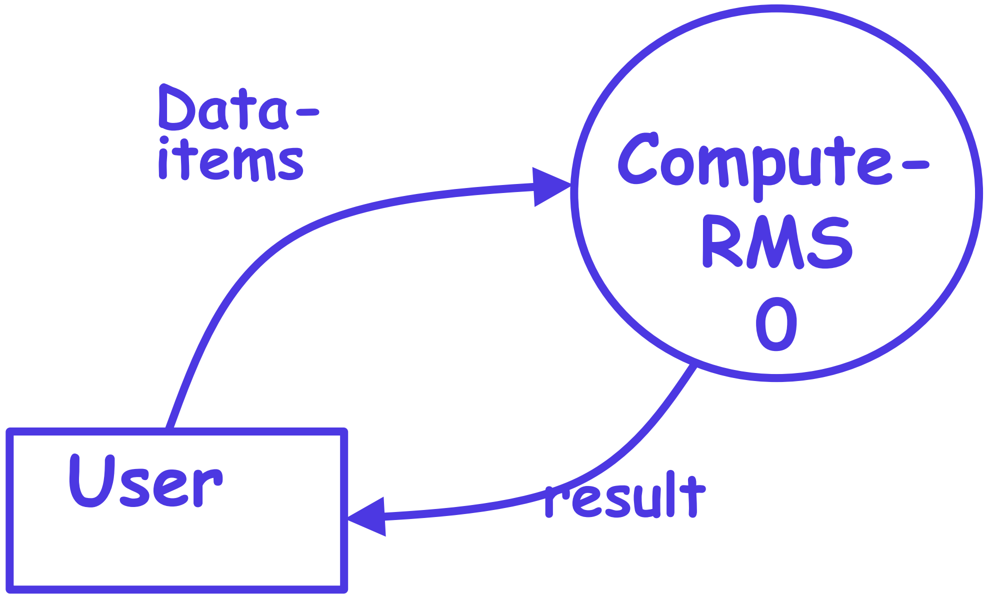
Example 1: RMS Calculating Software

- 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

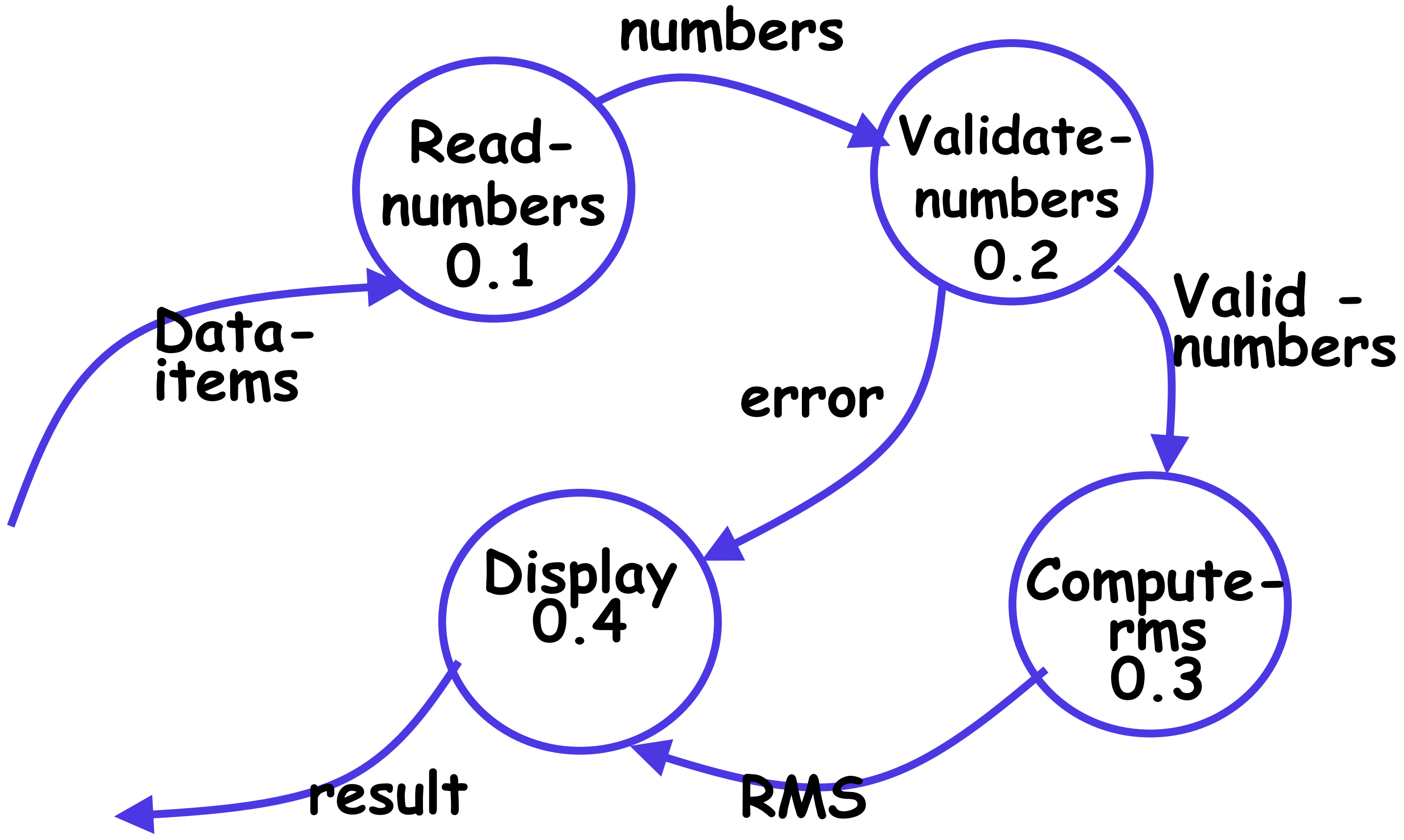
Example 1: RMS Calculating Software

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

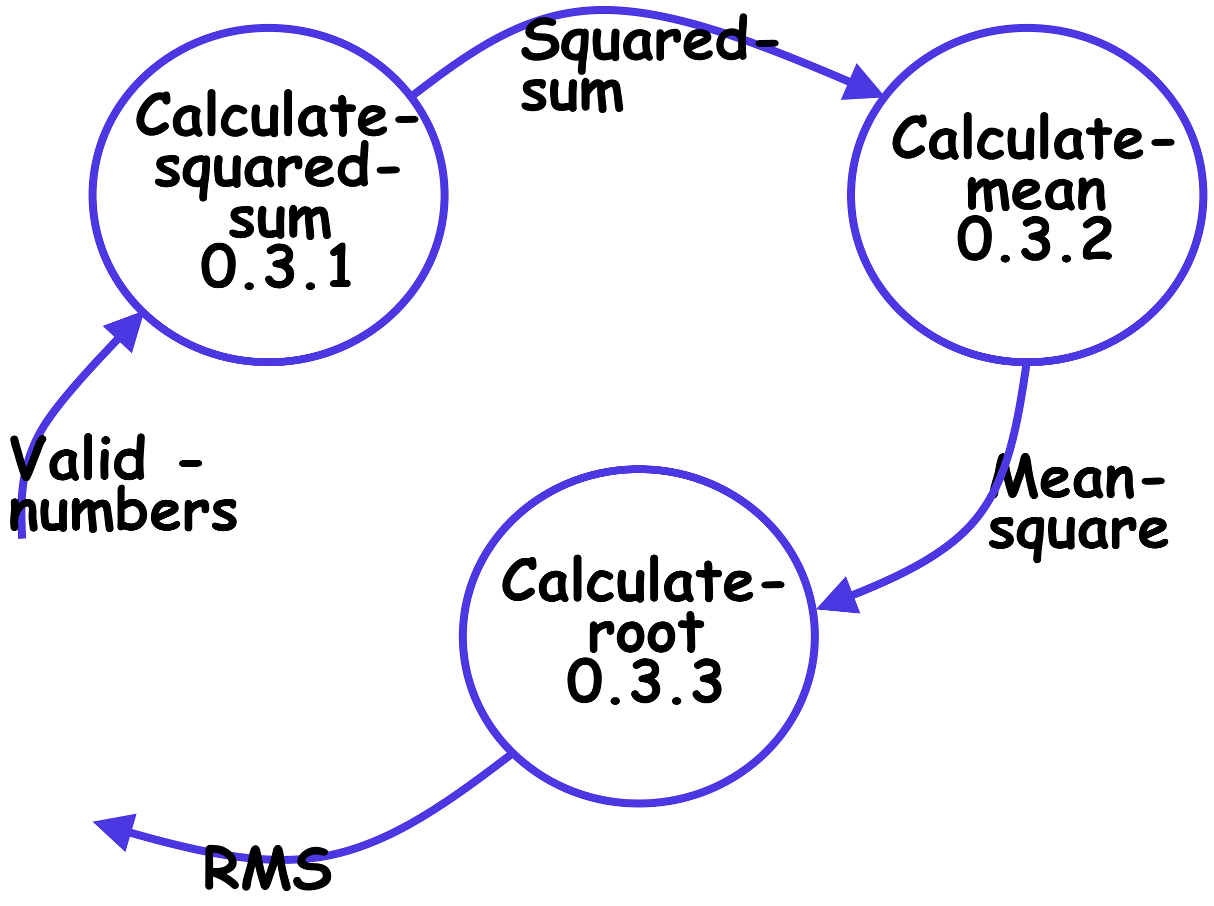
Example 1: RMS Calculating Software

- Accept input numbers from the user:
  - Validate the numbers,
  - Calculate the root mean square of the input numbers
  - Display the result.

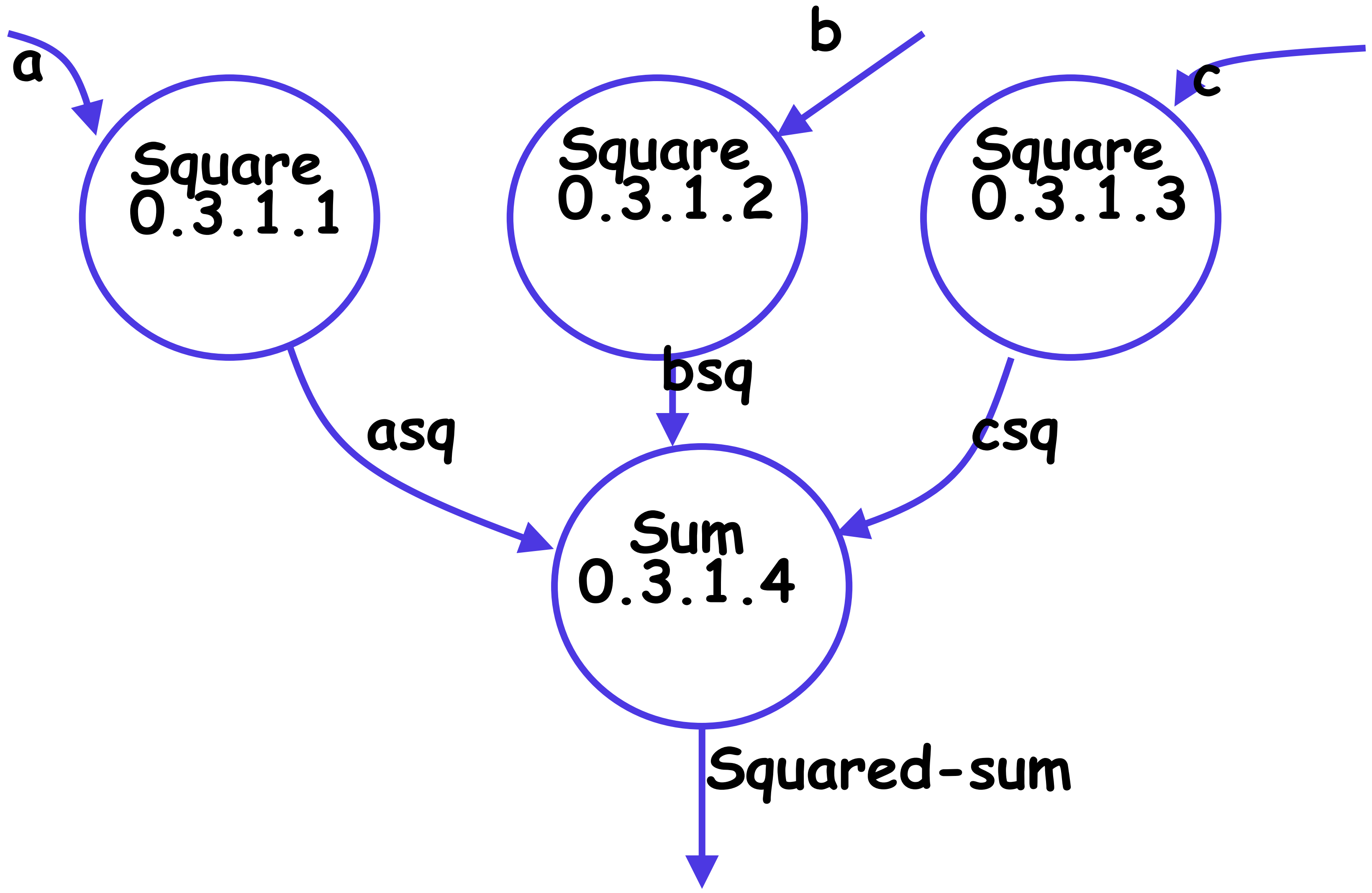
Example 1: RMS Calculating Software



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 the 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.
  - CASE tools capture the data items appearing in a DFD automatically to generate the data dictionary.

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.

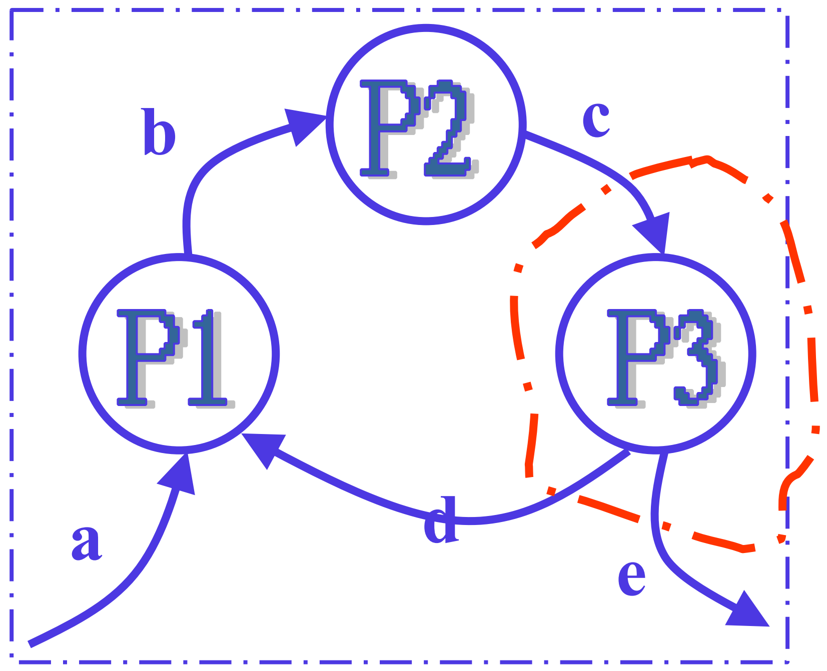
Data Dictionary for RMS Software

- 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*

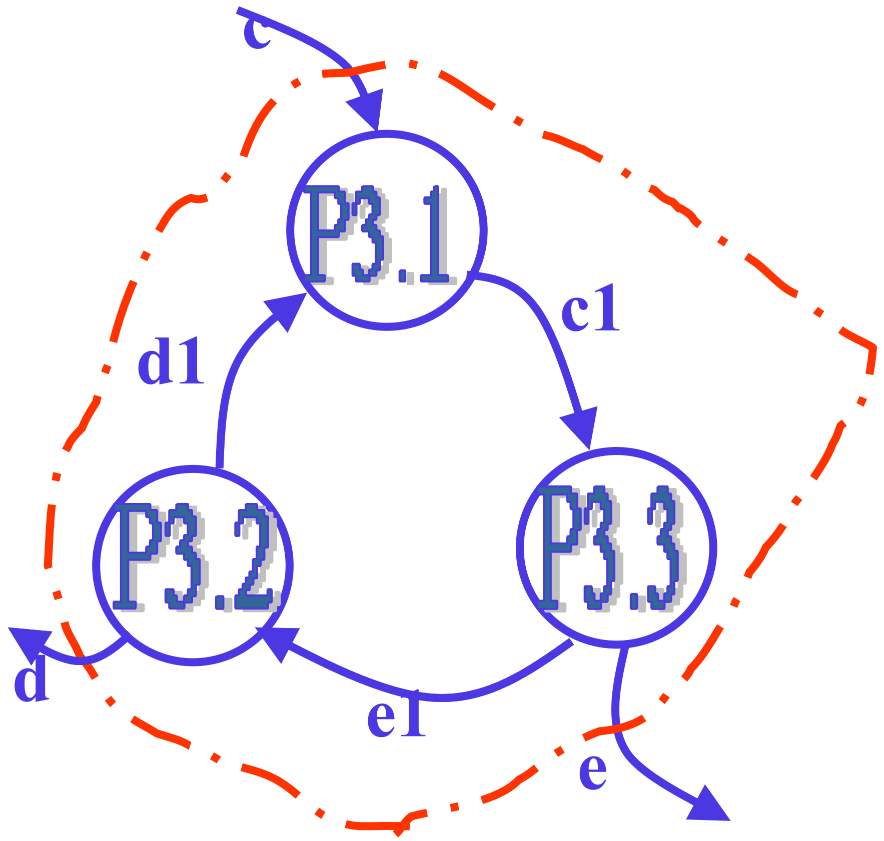
Balancing a DFD

- Data flowing into or out of a bubble:
  - Must match the data flows at the next level of DFD.
- 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



Level 1



Level 2

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.

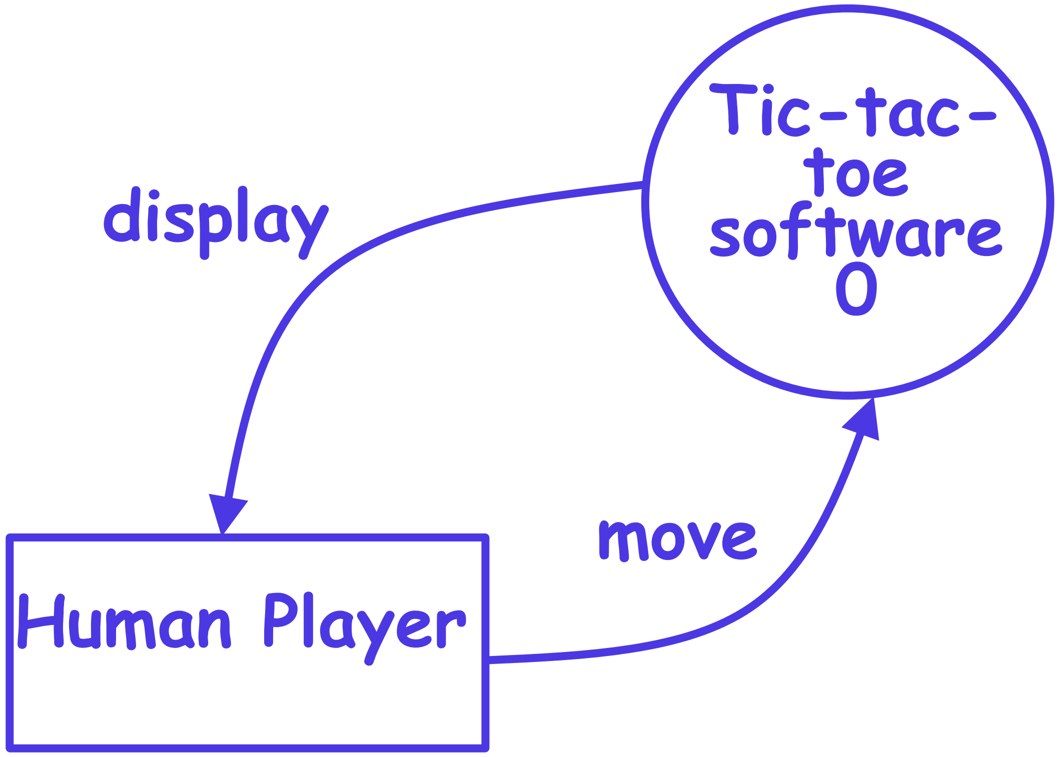
Example 2: Tic-Tac-Toe Computer Game

- A human player and the computer make alternate moves on a 3 X 3 square.
- A move consists of marking a previously unmarked square.
- The user inputs a number between 1 and 9 to mark 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.

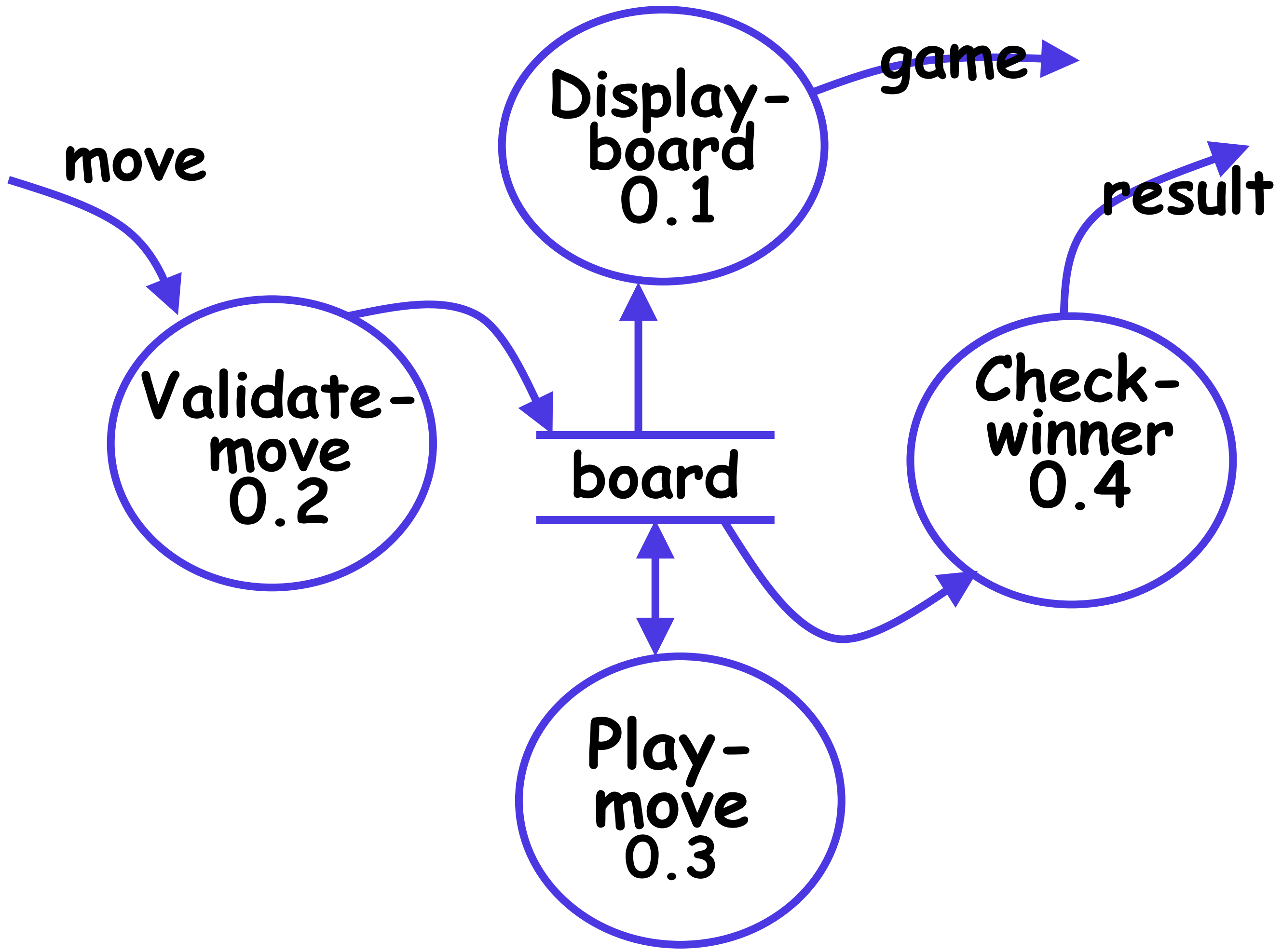
Example: Tic-Tac-Toe Computer Game

- 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


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

Data Dictionary

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

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.
- 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 3: 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.
- 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.

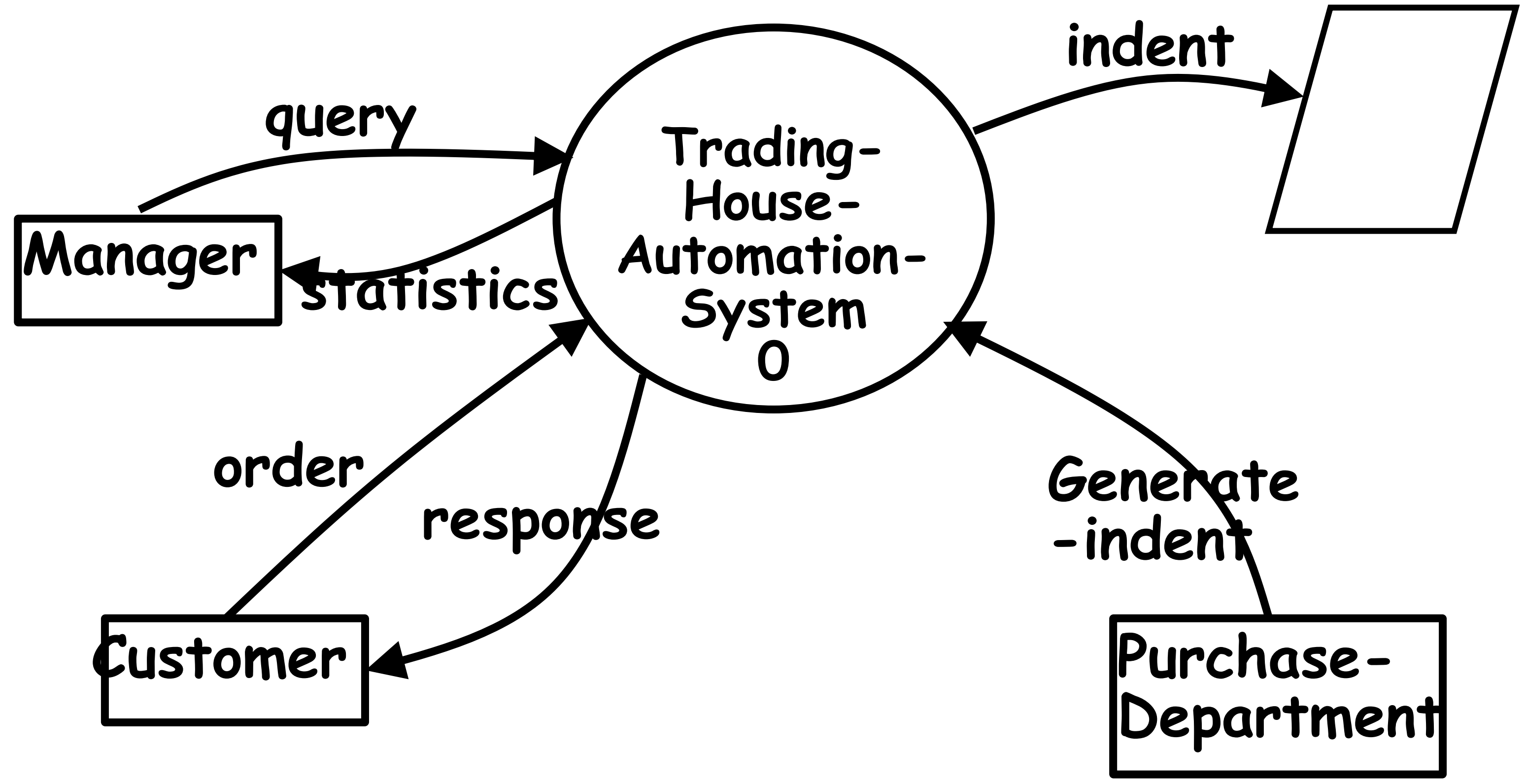
Example: Trading-House Automation System (TAS)

- The items in a customer's order that the trading house deals with:
  - Are checked for availability in 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.
- 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.

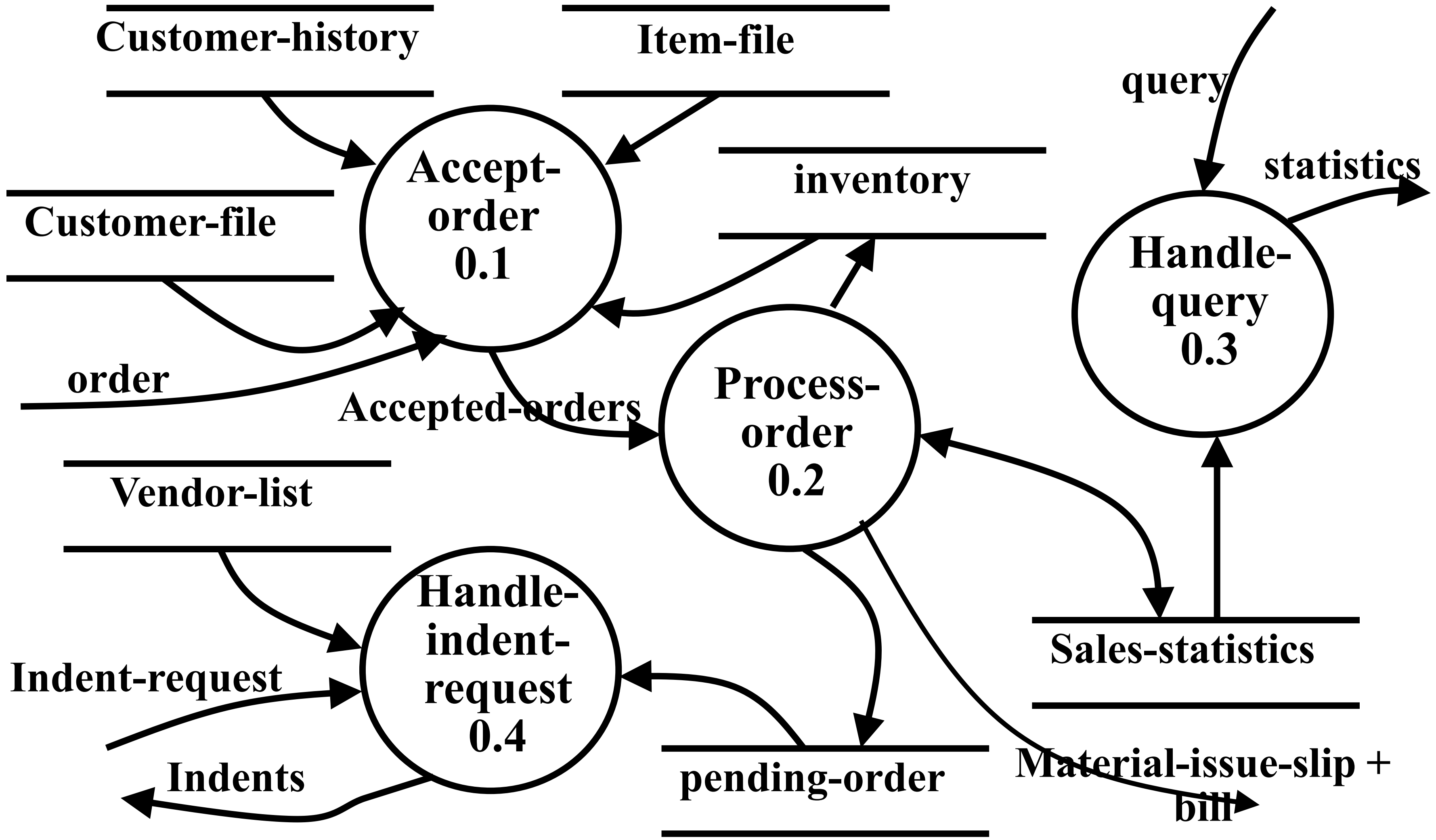
Example: Trading-House Automation System (TAS)

- 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.
- 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.
  - 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


Customer-history
Item-file
query
statistics
Accept-order 0.1
inventory
Customer-file
Handle-query 0.3
order
Accepted-orders
Process-order 0.2
Vendor-list
Sales-statistics
Handle-indent-request 0.4
Indent-request
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

Commonly Made Errors: DFD

- 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

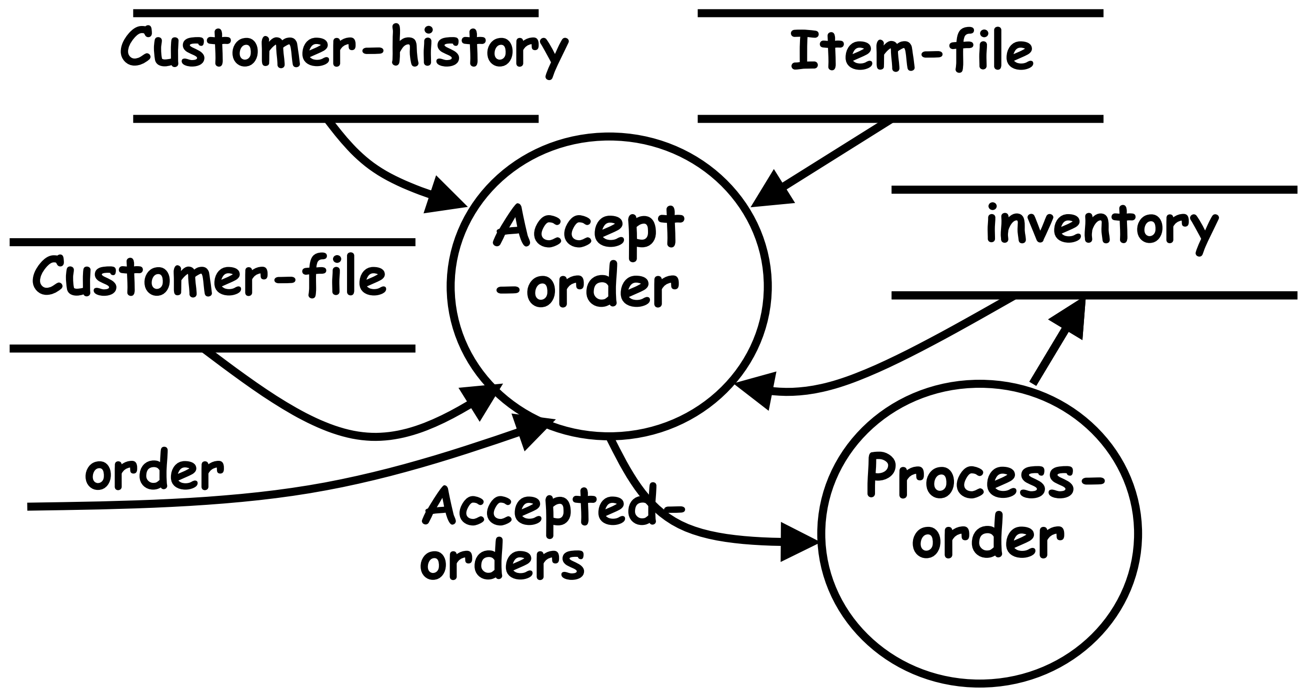
- DFD models suffer from several shortcomings:
- DFDs leave ample scope to be imprecise.
  - In a DFD model, we infer about the function performed by a bubble from its label.
  - A label may not capture all the functionality of a bubble.

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.

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.

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.

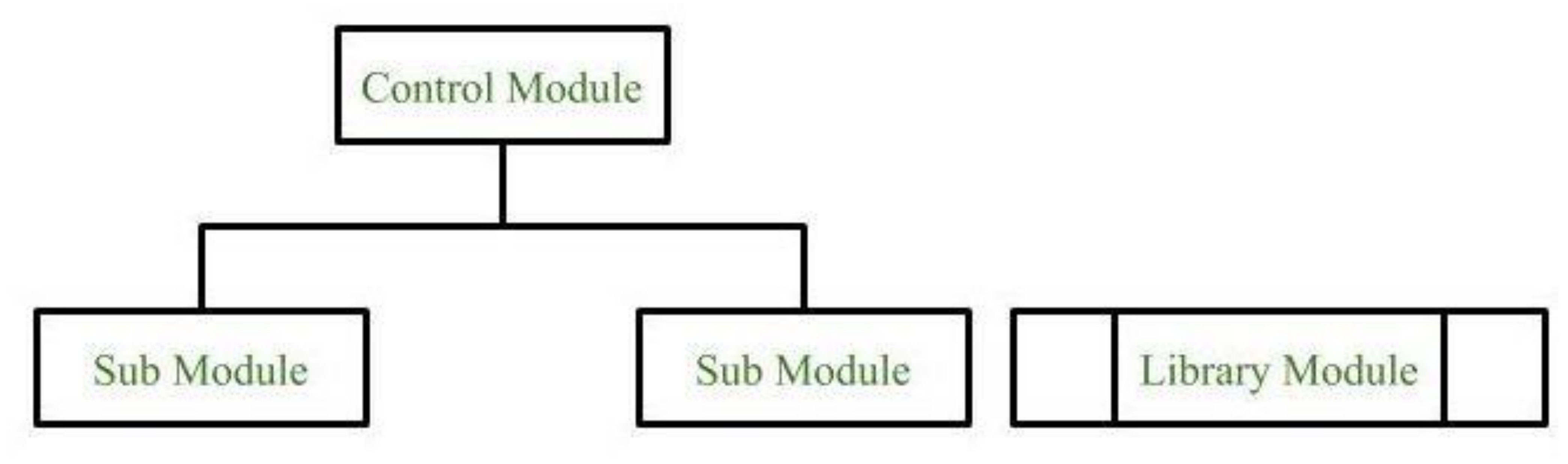
Structure Chart

- Structure chart representation
  - Easily implementable using programming languages.
- **Structure Chart** represent hierarchical structure of modules.
- It breaks down the entire system into lowest functional modules, describe functions and sub-functions of each module of a system.
- Structure Chart partitions the system into black boxes (functionality of the system is known to the users but inner details are unknown).
- Inputs are given to the black boxes and appropriate outputs are generated.

Basic Building Blocks of Structure Chart

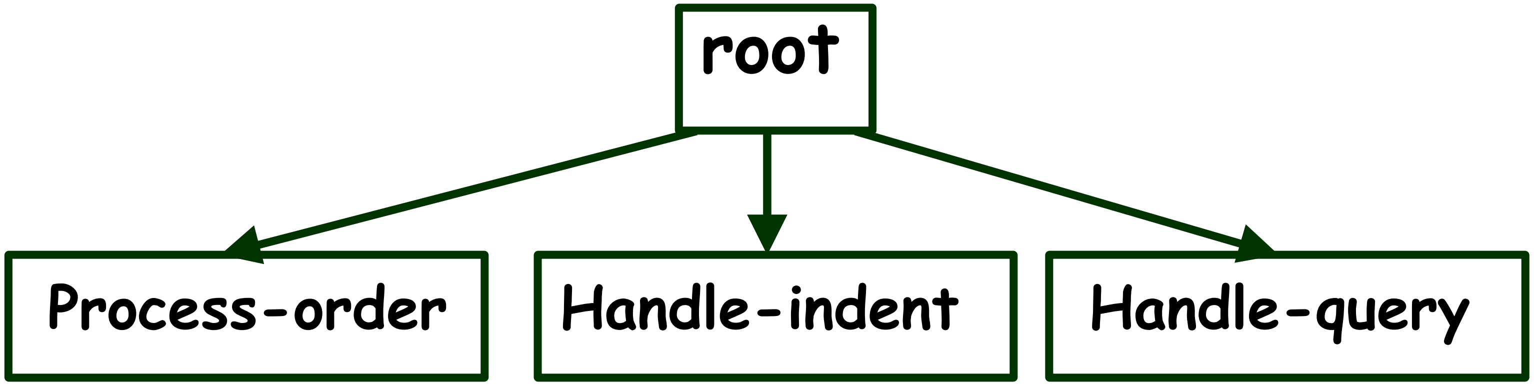
- **Module**
  It represents the process or task of the system. It is of three types.
- **Control Module**
  A control module branches to more than one sub module.
- **Sub Module**
  Sub Module is a module which is the part (Child) of another module.
- **Library Module**
  Library Module are reusable and invokable from any module.

Basic Building Blocks of Structure Chart



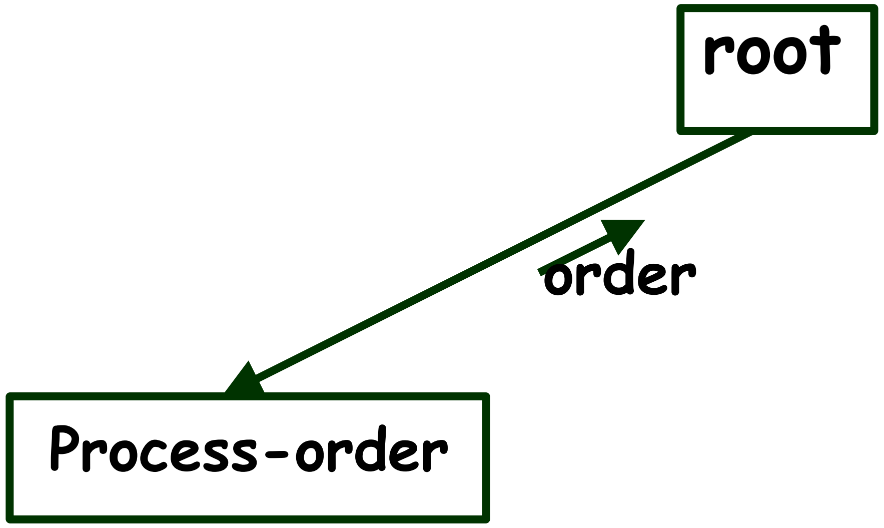
Arrows

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



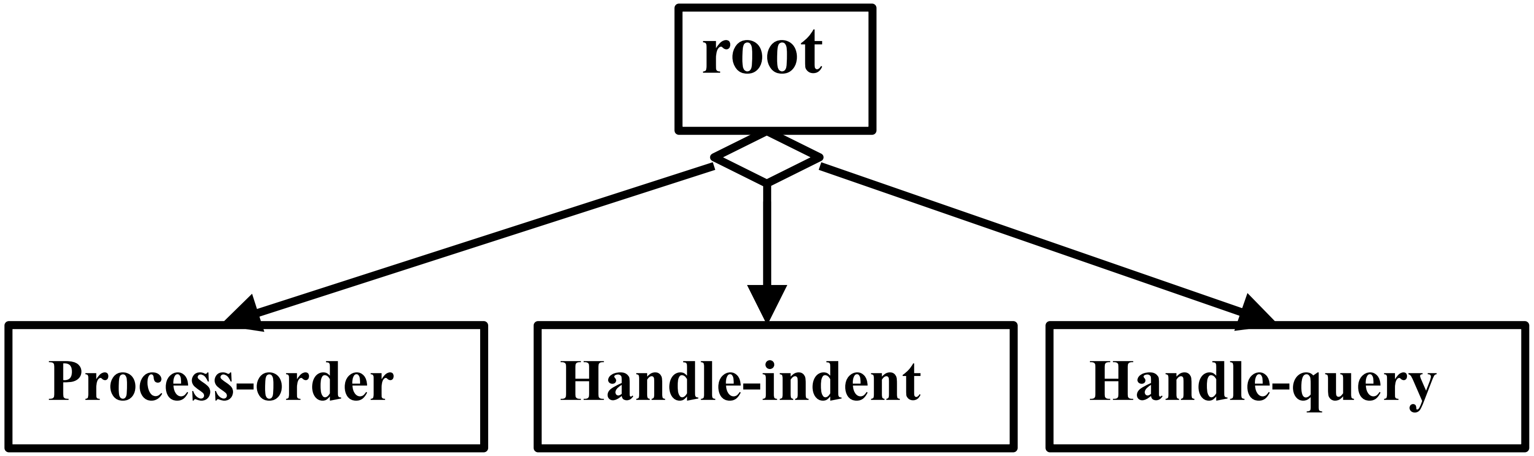
Data Flow Arrows

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



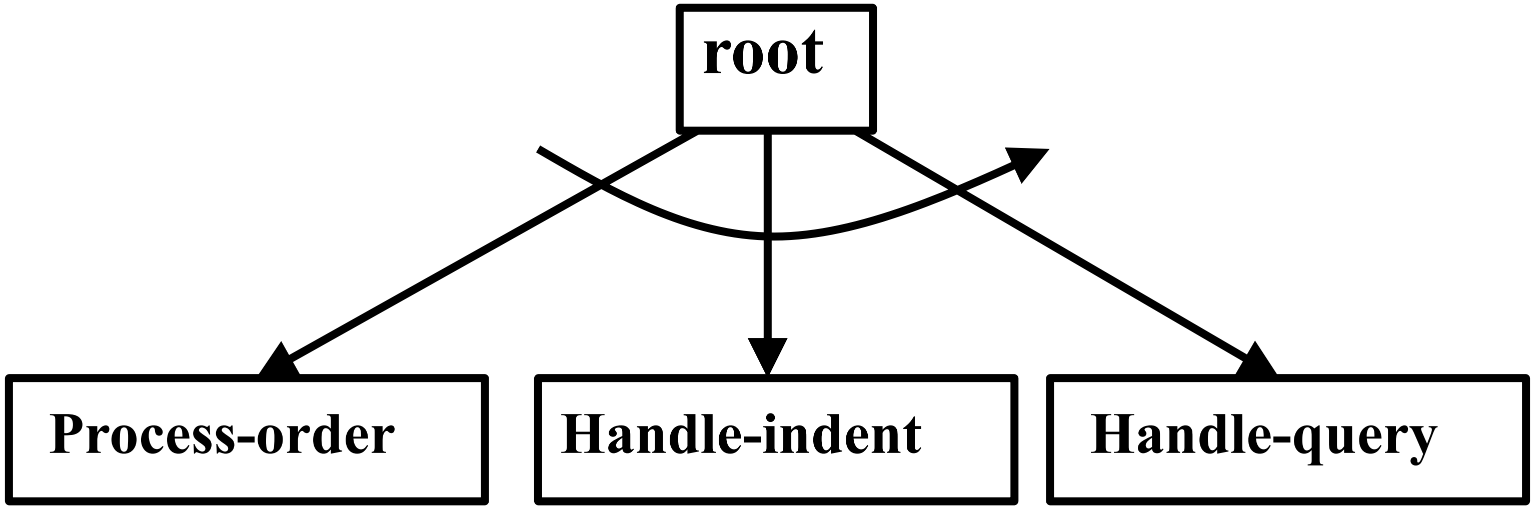
Selection

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



Repetition

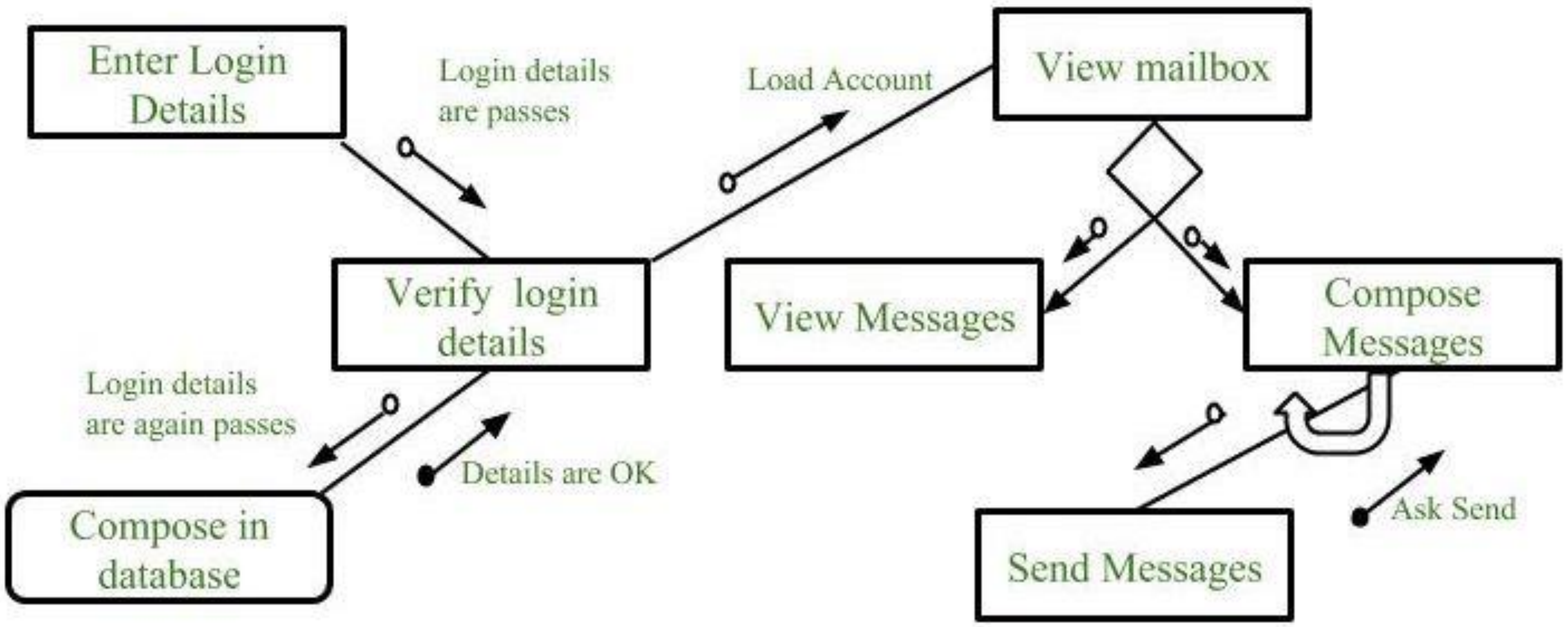
- A loop around control flow arrows denotes that the concerned modules are invoked repeatedly.



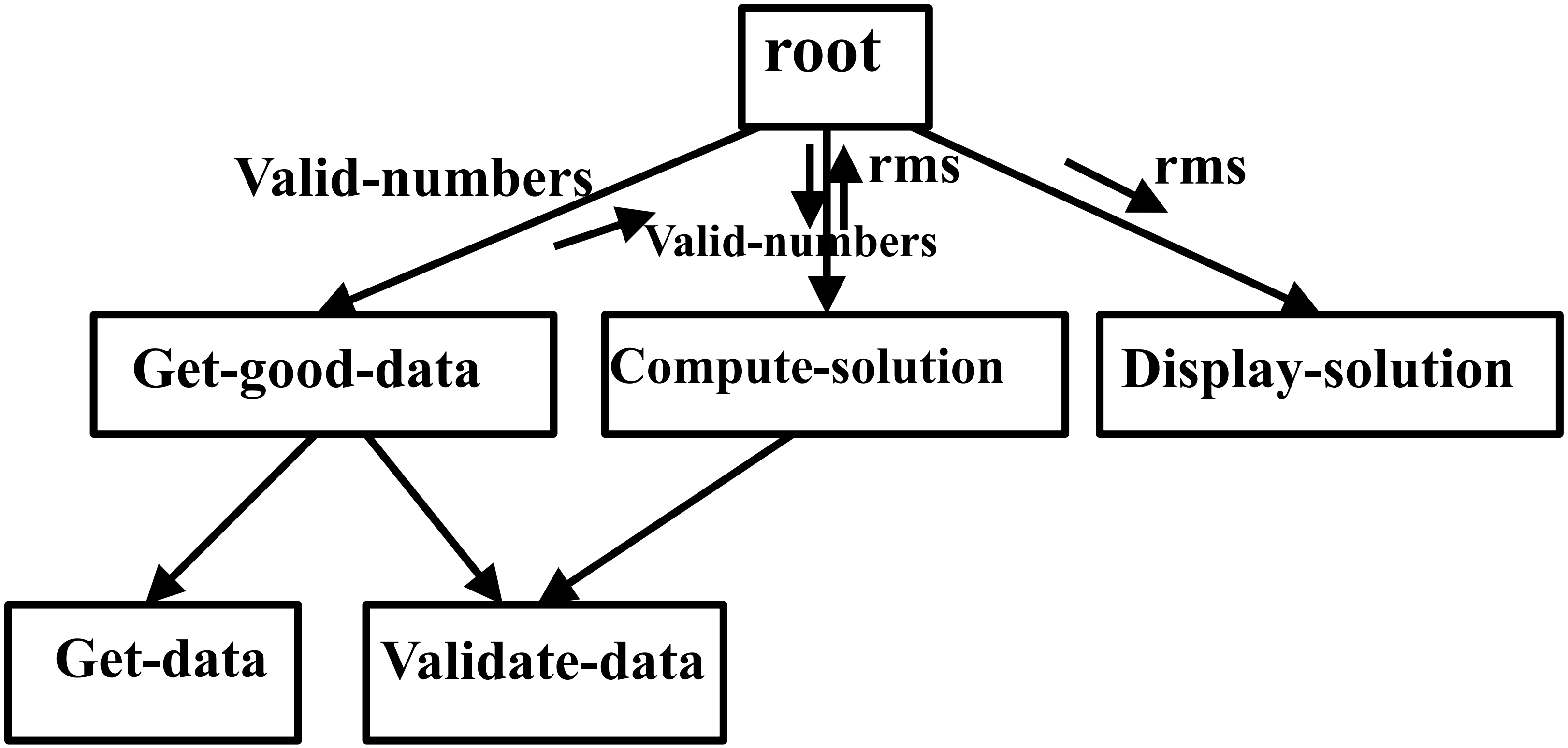
Structure Chart

- 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.

Structure Chart



Example



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

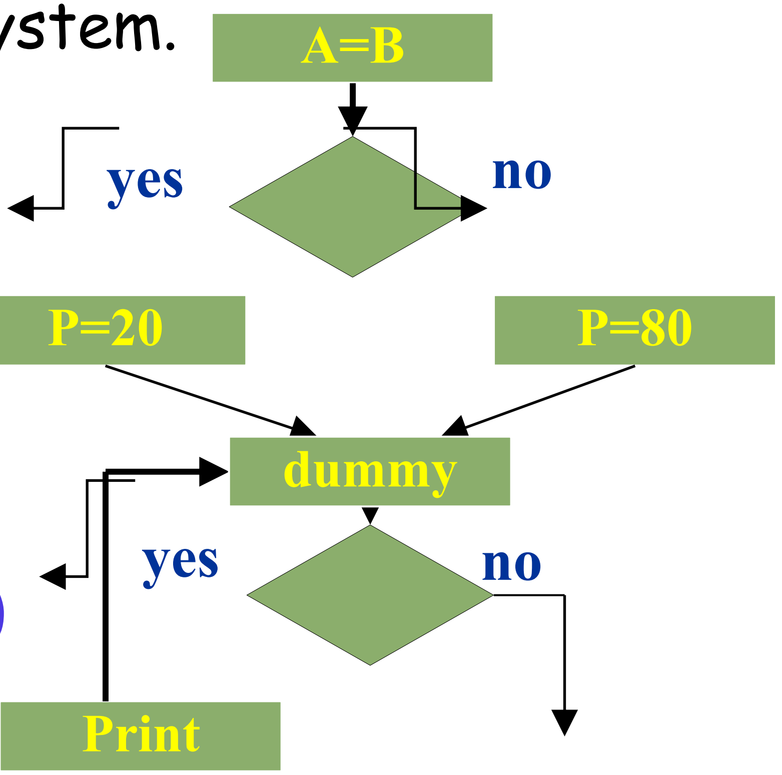
Shortcomings of Structure Chart

- 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.

Flow Chart (Aside)

- We are all familiar with the flow chart representations:
  - Flow chart is a convenient technique to represent the flow of control in a system.

```
• A=B
• if(c == 100)
•       P=20
• else  p= 80
• while(p>20)
•       print(student mark)
```



Flow Chart versus Structure Chart

- A structure chart differs from a flow chart in three principal ways:
  - It is difficult to identify modules of a software from its flow chart representation.
  - Data interchange among the modules is not represented in a flow chart.
  - Sequential ordering of tasks inherent in a flow chart is suppressed in a structure chart.

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 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.
- Some guidelines:
  - Trace the inputs until a bubble is found whose output cannot be deduced from the inputs alone.
  - Processes which validate input are not central transforms.
  - Processes which sort input or filter data from it are.

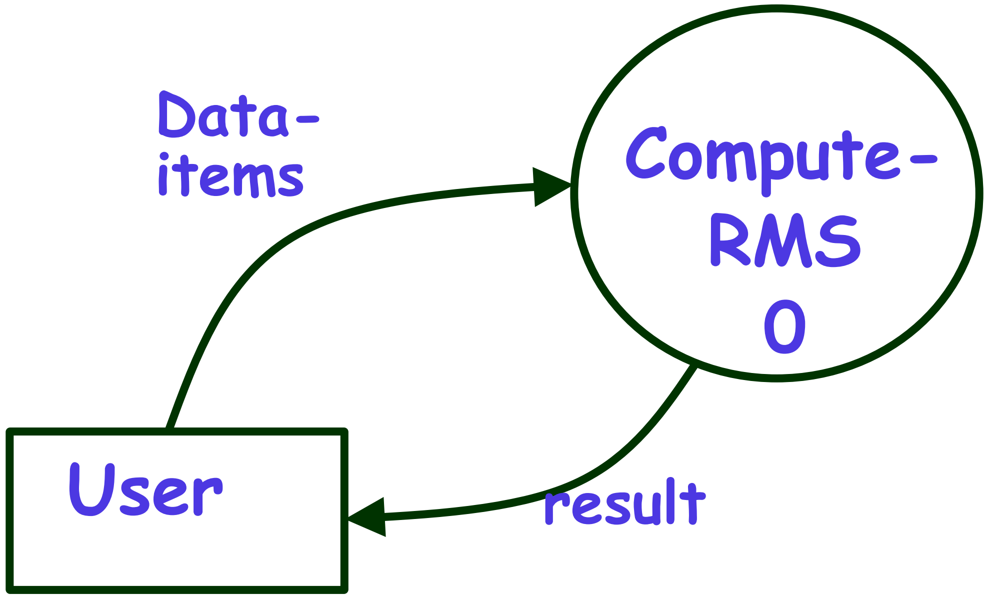
Transform Analysis

- 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

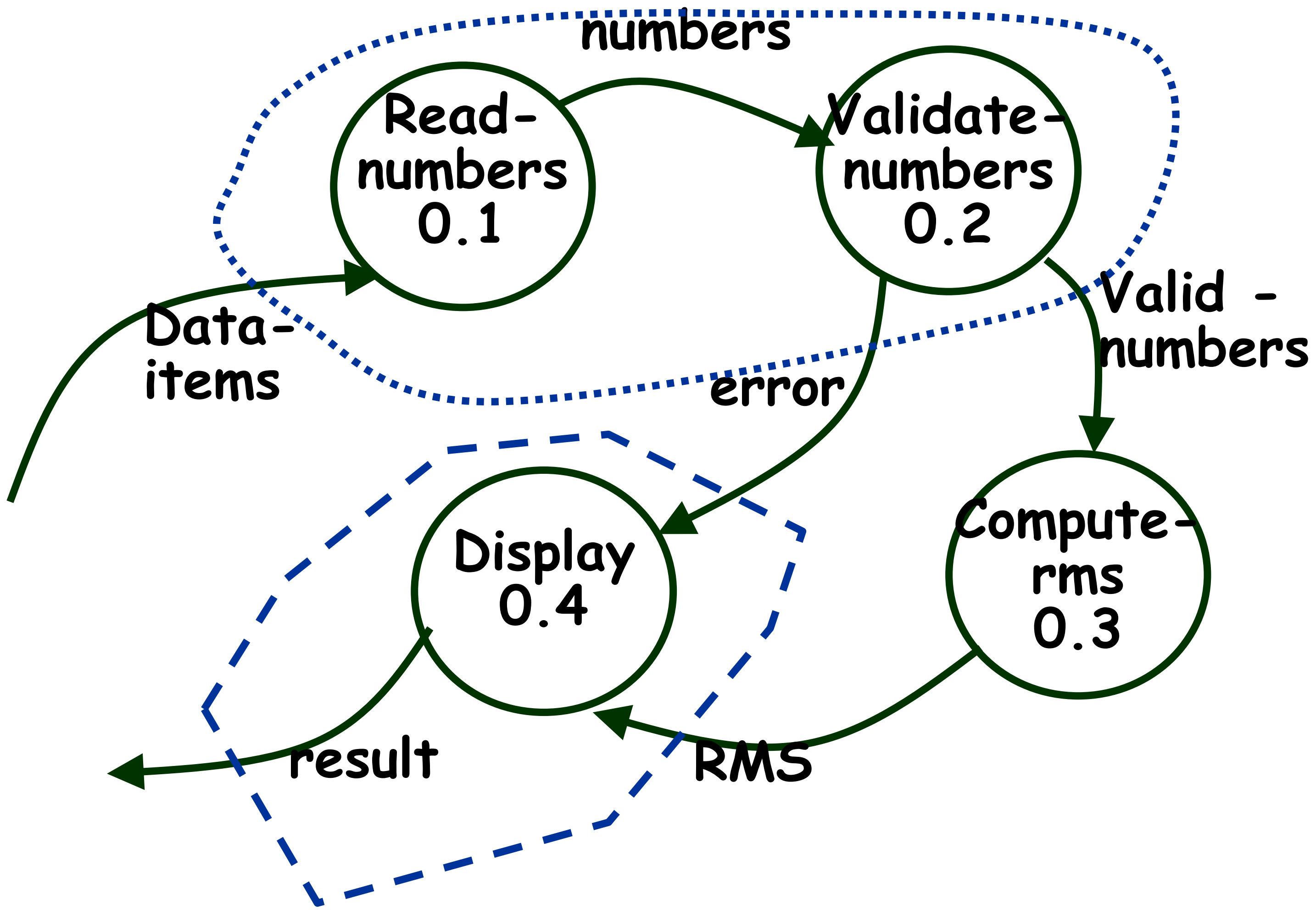


Context Diagram

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.

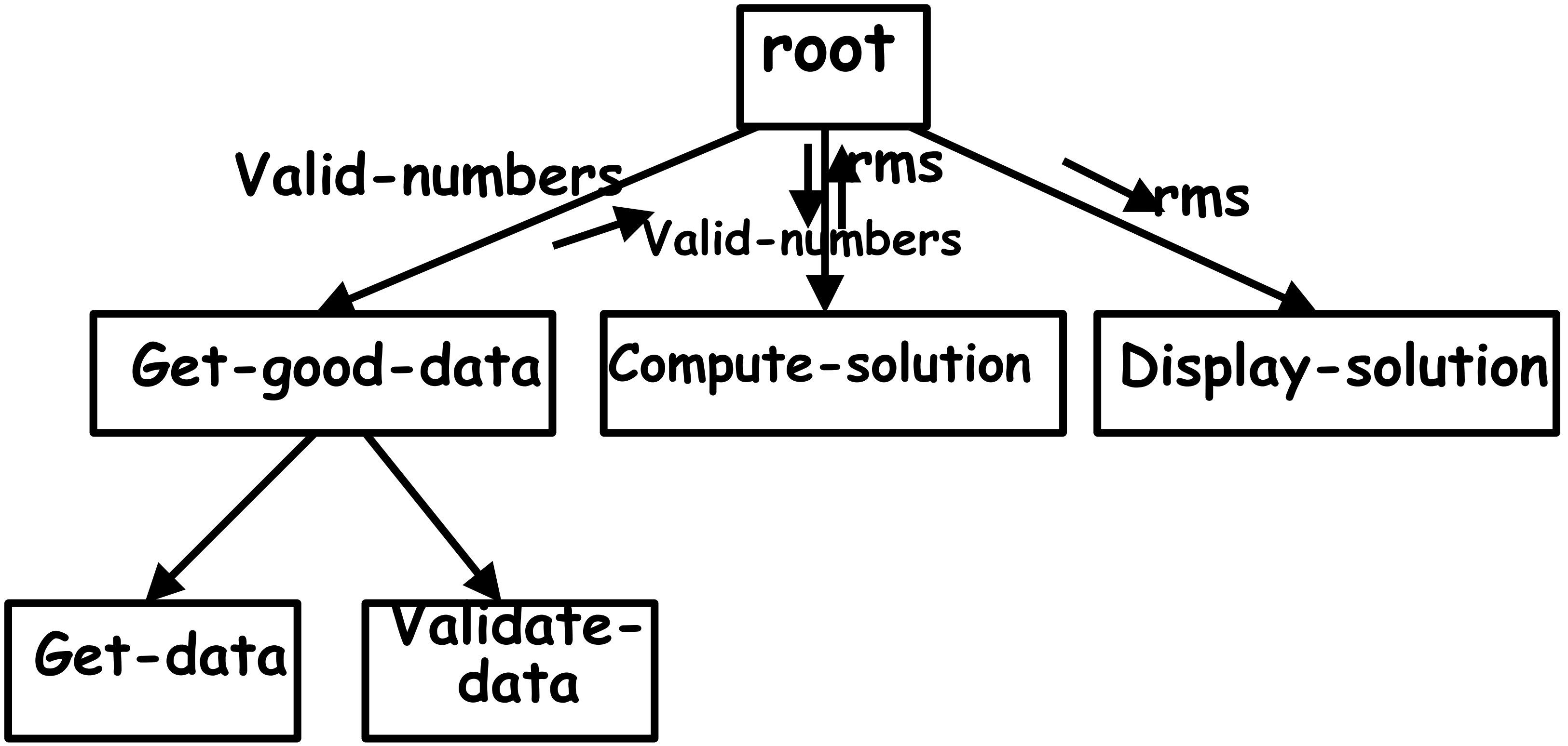
Example 1: RMS Calculating Software



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.

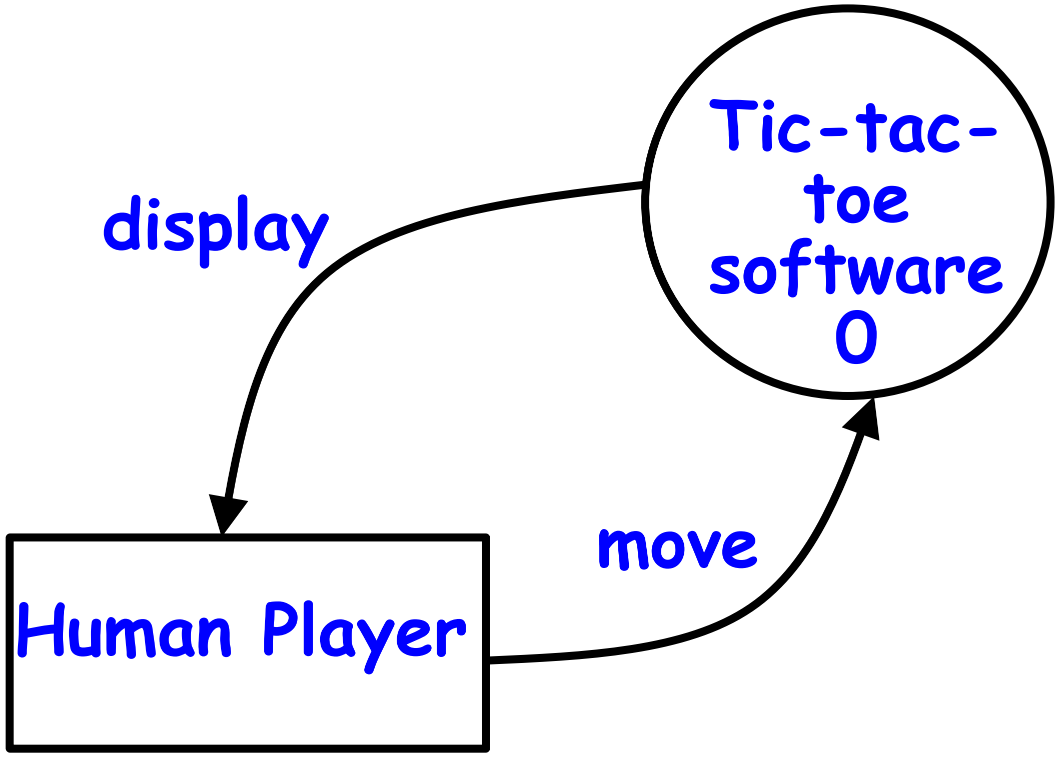
Example 1: RMS Calculating Software



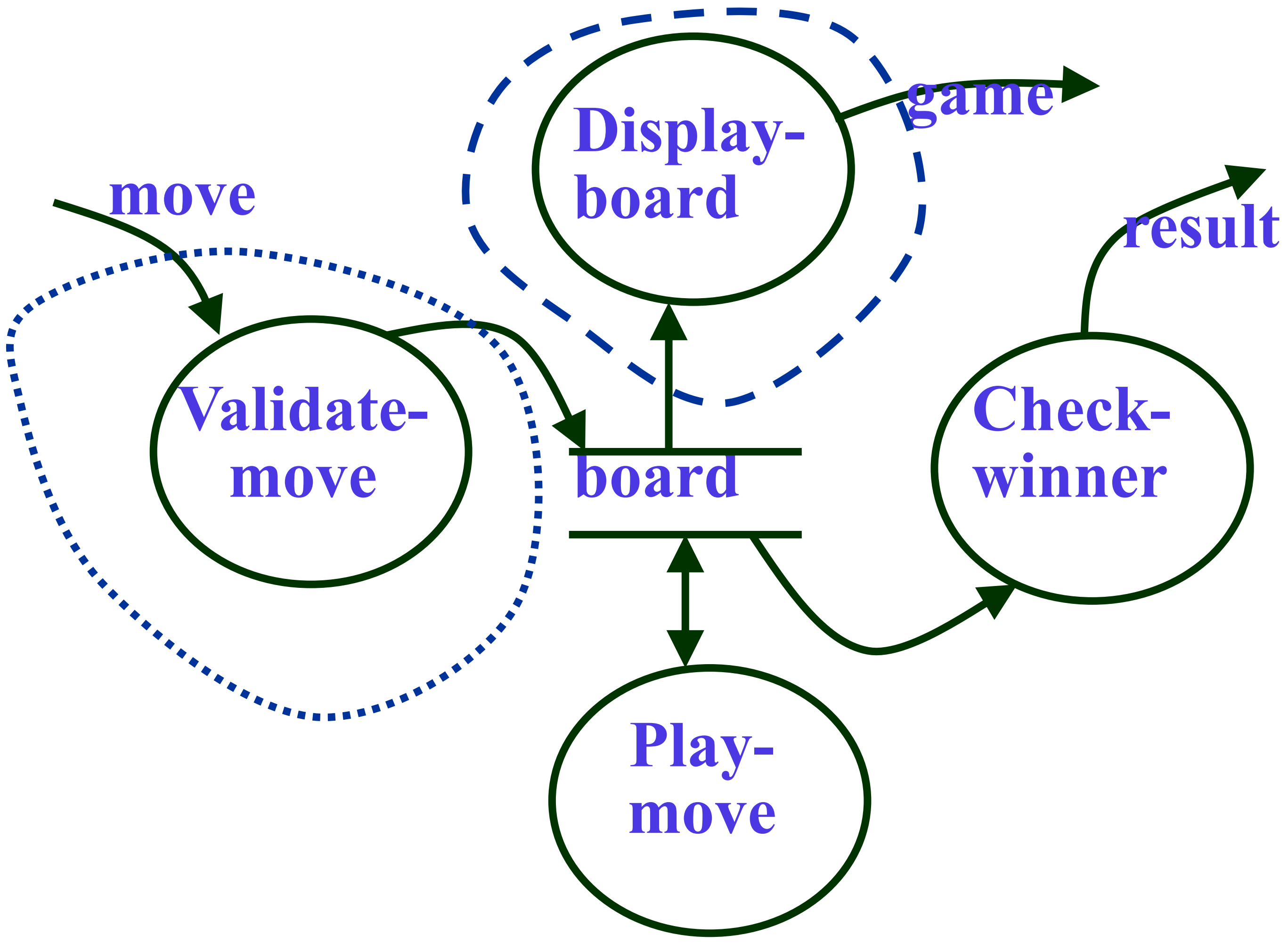
Example 2: Tic-Tac-Toe Computer Game

- 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.

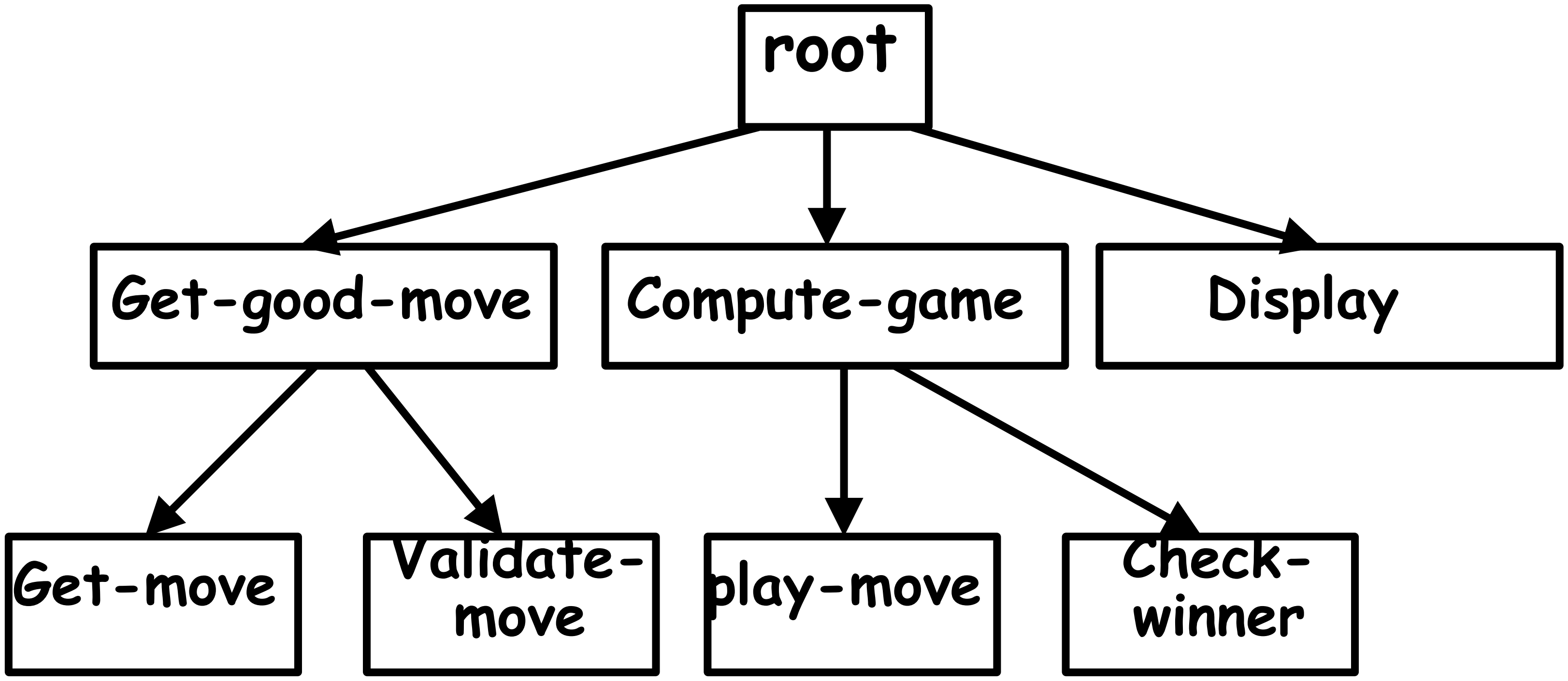
Context Diagram for Example 2



Level 1 DFD


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

Structure Chart



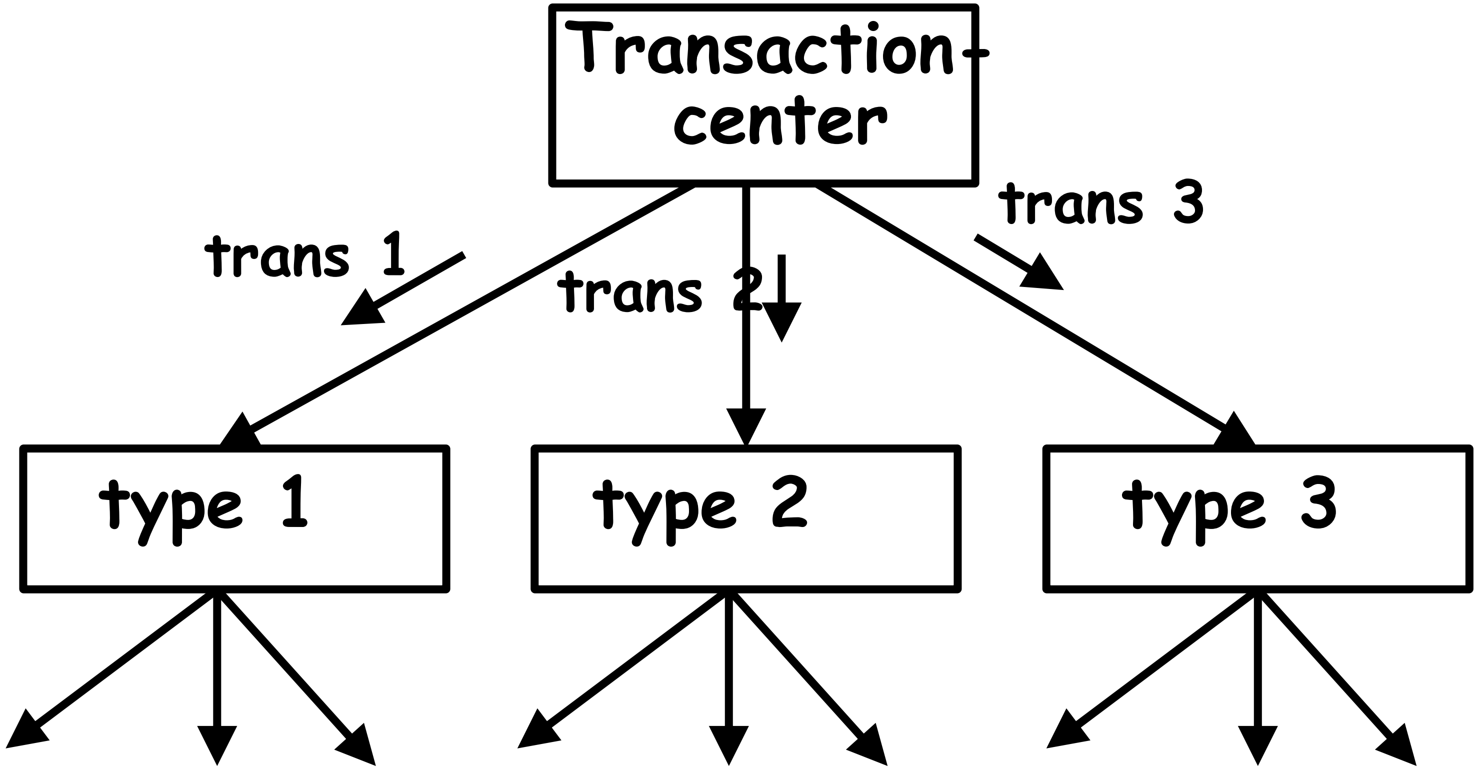
Transaction Analysis

- Useful for designing transaction processing programs.
  - Transform-centered systems:
    - Characterized by similar processing steps for every data item processed by input, process, and output bubbles.
  - Transaction-driven systems,
    - One of several possible paths 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



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 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:
  - employee names,
  - code numbers,
  - basic salaries, etc.
- Whereas, in object oriented design:
  - 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.
- 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.
- 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.