Parul® University

Software Engineering (203105303)

Prof. Yoothika Patel, Assistant Professor
Computer Science & Engineering

Parul®
University

DIGITAL LEARNING CONTENT

Unit-4

Architectural Design

Architectural Design

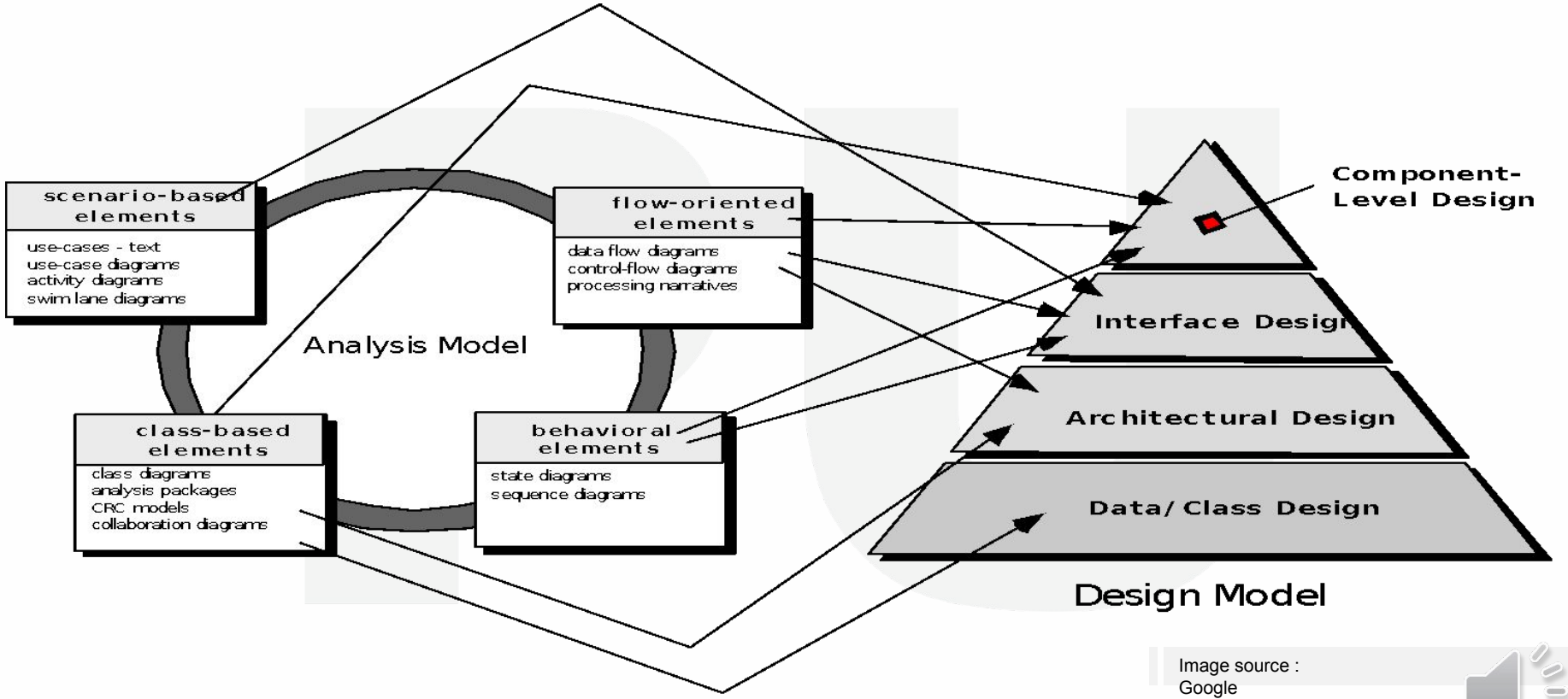


Image source :
Google

Architecture?

The architecture is not the operational software. Rather, it is a representation that enables a software engineer to:

(1) analyze the effectiveness of the design in meeting its stated requirements,

(2) consider architectural alternatives at a stage when making design changes is still relatively easy, and

(3) reduce the risks associated with the construction of the software.

Why is Architecture Important?

- Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.

The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.

Architectural Genres

- *Genre* implies a specific category within the overall software domain.
- Within each category, you encounter a number of subcategories.
  - For example, within the genre of *buildings*, you would encounter the following general styles: houses, condos, apartment buildings, office buildings, industrial building, warehouses, and so on.
  - Within each general style, more specific styles might apply. Each style would have a structure that can be described using a set of predictable patterns.

Architectural Styles

Each style describes a system category that encompasses:

(1)A **set of components** (e.G., A database, computational modules) that perform a function required by a system,

(2)A **set of connectors** that enable “communication, coordination and cooperation” among components,

(3) **Constraints** that define how components can be integrated to form the system, and

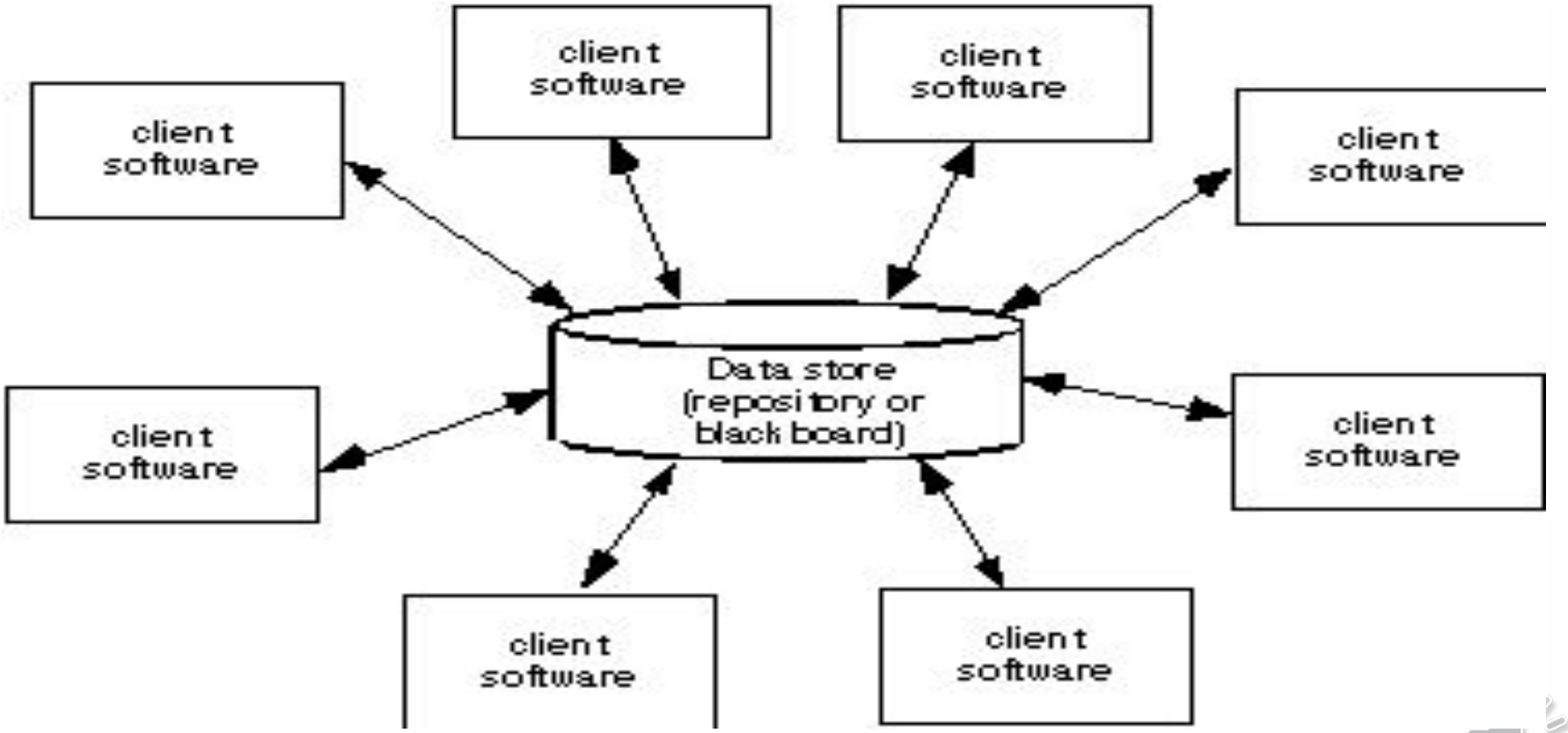
(4) **Semantic models** that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

Parul®
University

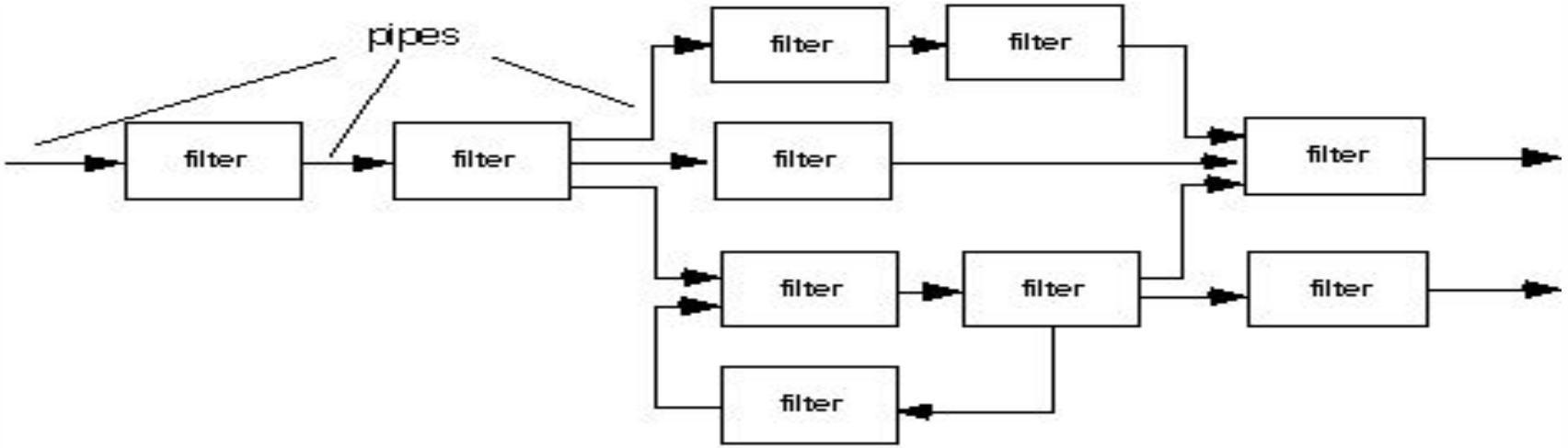
Various Architectures

- Data-centered architectures
- Data flow architectures
- Call and return architectures
- Object-oriented architectures
- Layered architectures

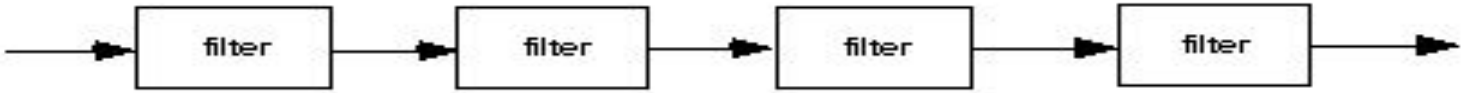
Data-Centered Architecture



Data Flow Architecture

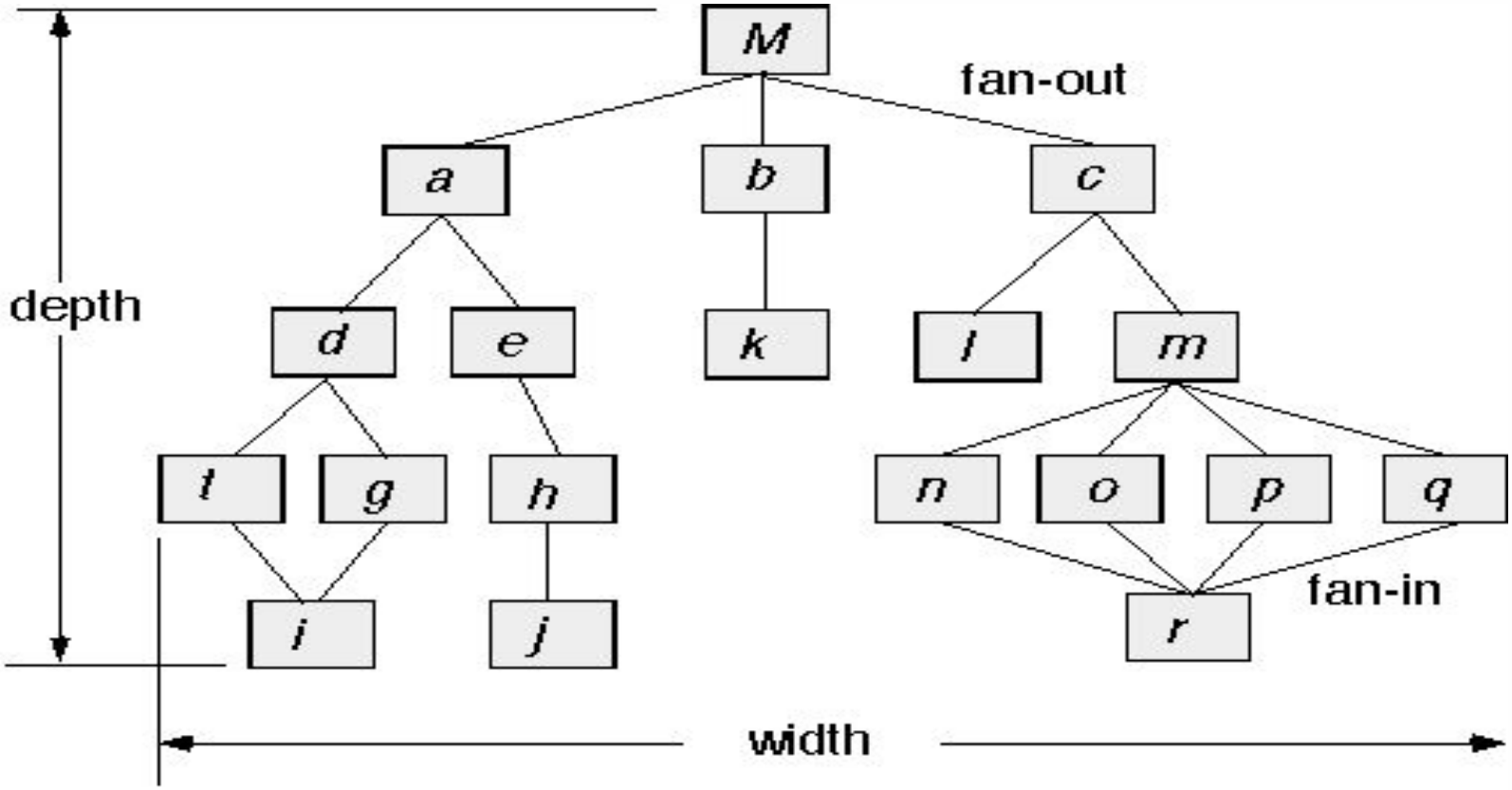


(a) pipes and filters

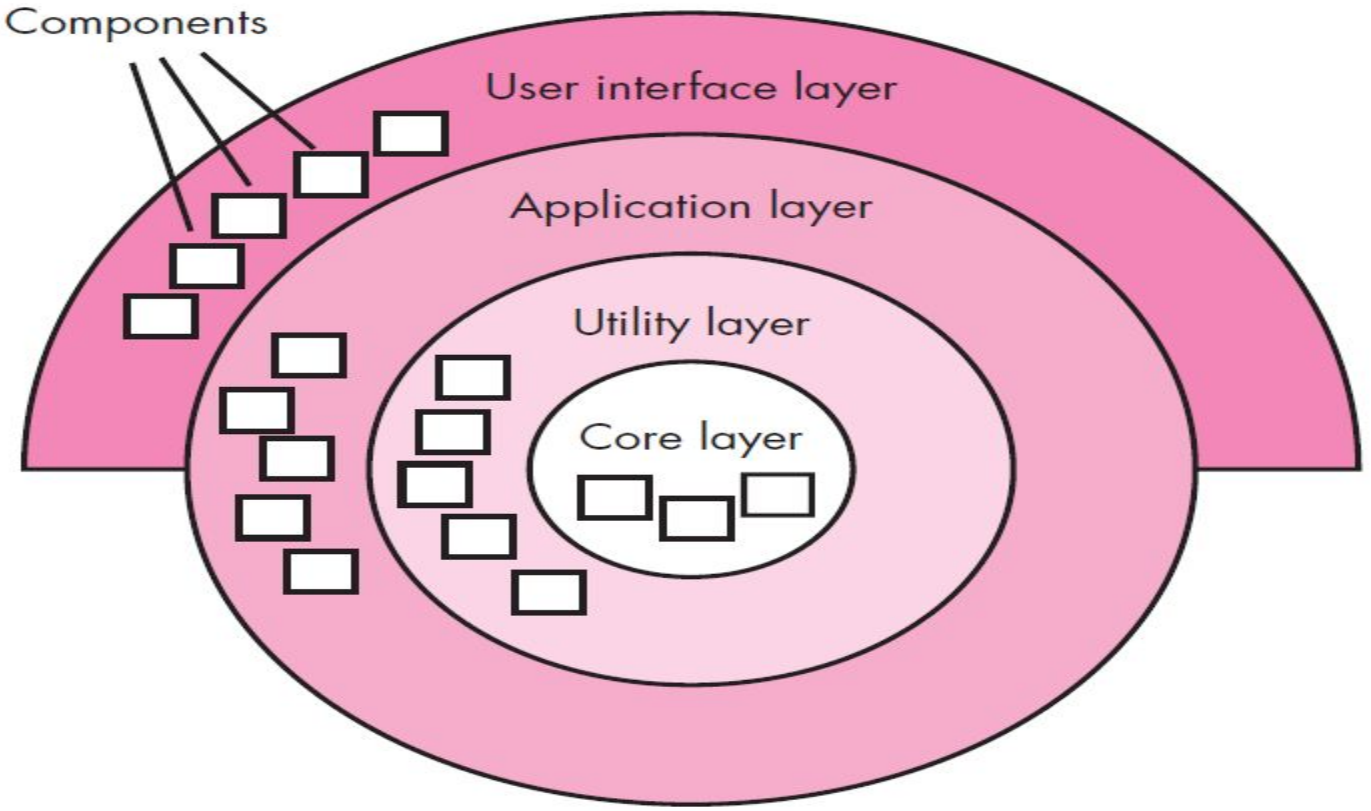


(b) batch sequential

Call and Return Architecture



Layered Architecture



Architectural Patterns

Concurrency—applications must handle multiple tasks in a manner that simulates parallelism

operating system process management pattern

task scheduler pattern

Architectural Patterns (Cont.)

Persistence—Data persists if it survives past the execution of the process that created it. Two patterns are common:

a *database management system* pattern that applies the storage and retrieval capability of a DBMS to the application architecture

an *application level persistence* pattern that builds persistence features into the application architecture

Architectural Patterns (cont.)

Distribution— the manner in which systems or components within systems communicate with one another in a distributed environment

A *broker* acts as a ‘middle-man’ between the client component and a server component.

Architectural Design

The software must be placed into context

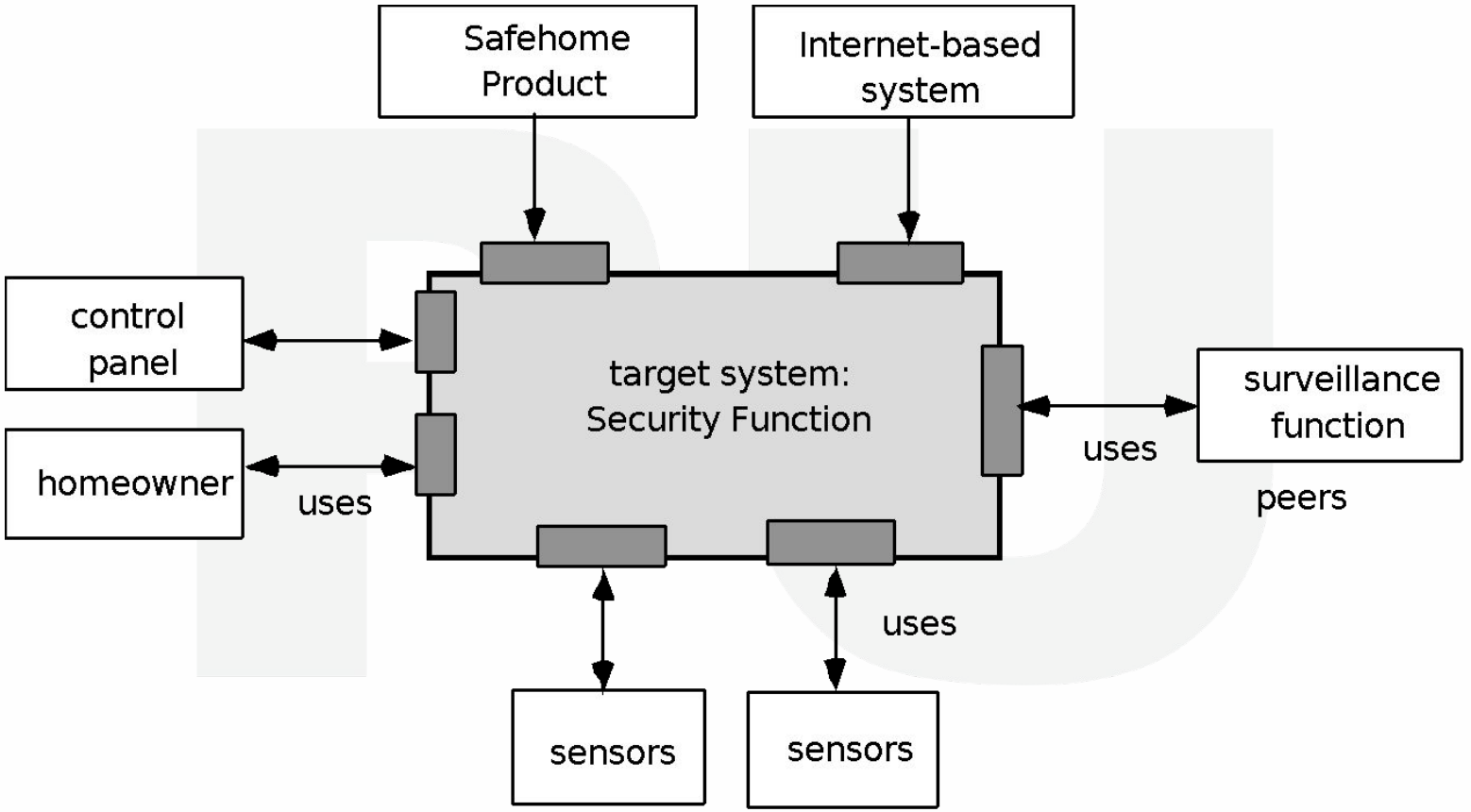
The design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction

A set of architectural archetypes should be identified

An *archetype* is an abstraction (similar to a class) that represents one element of system behavior

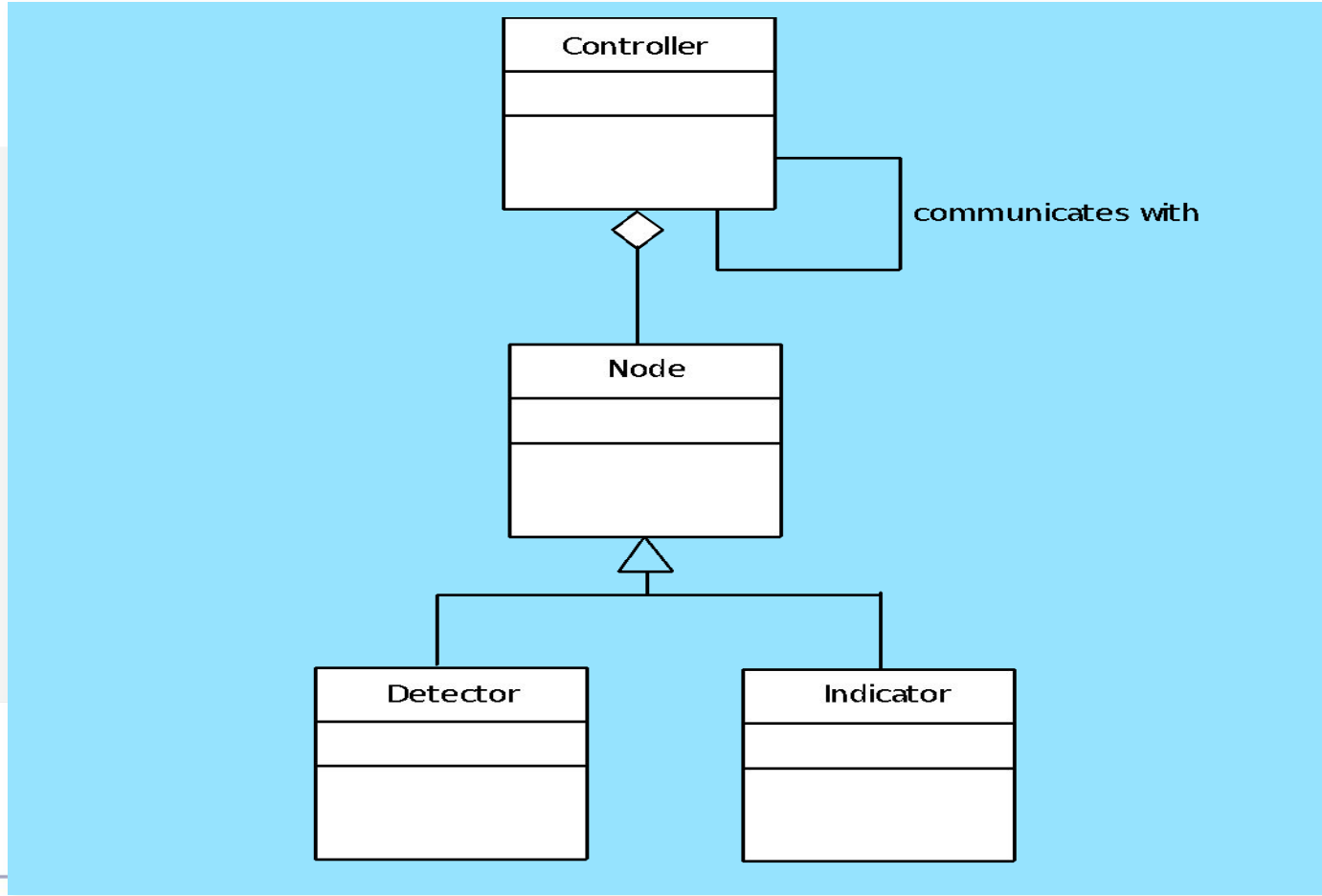
The designer specifies the structure of the system by defining and refining software components that implement each archetype

Architectural Context

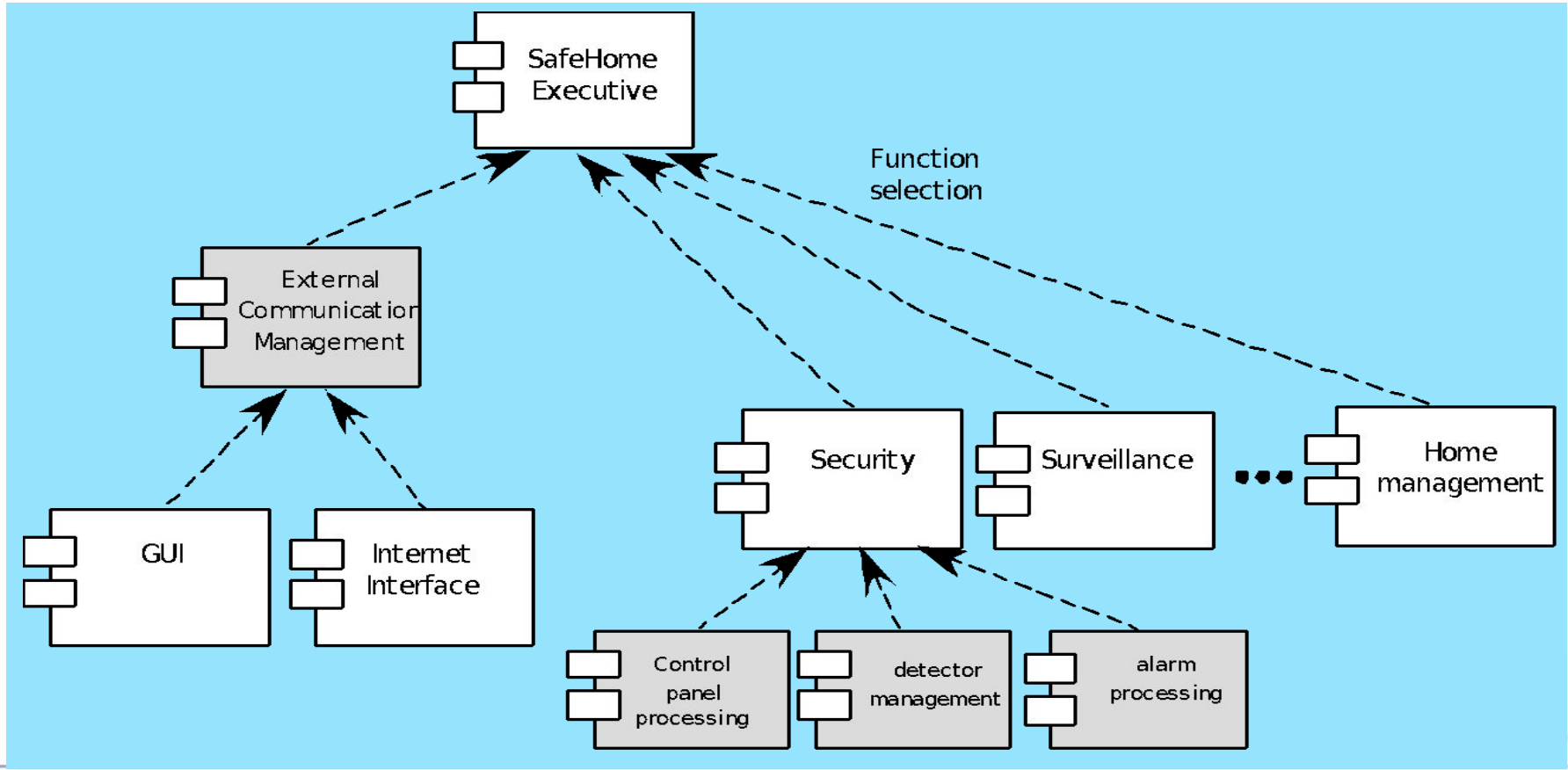


Archetypes

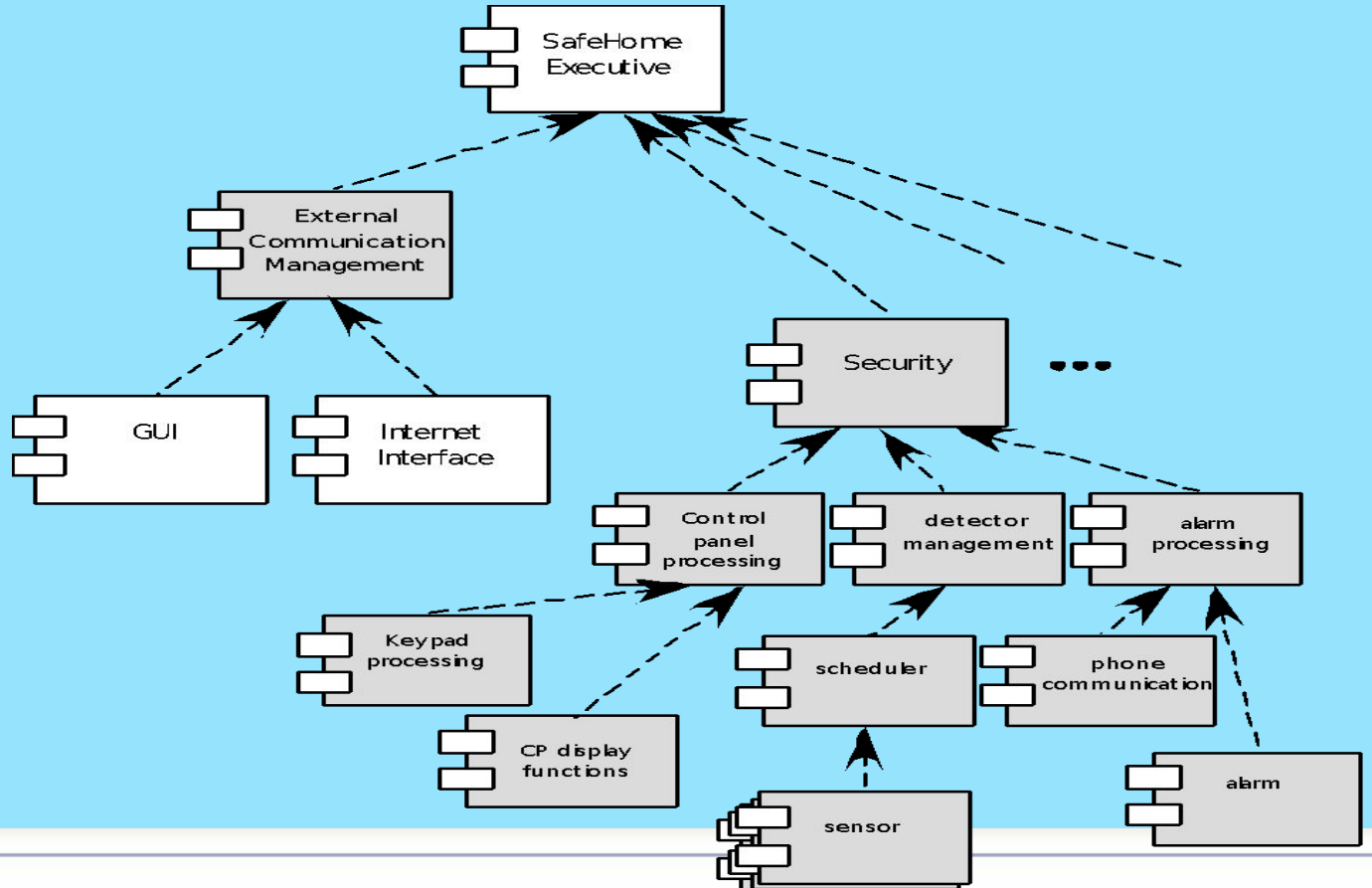
UML relationships for *SafeHome* security function archetypes



Component Structure



Refined Component Structure



Architectural Complexity

The overall complexity of a proposed architecture is assessed by considering the dependencies between components within the architecture.

Sharing dependencies represent dependence relationships among consumers who use the same resource or producers who produce for the same consumers.

Flow dependencies represent dependence relationships between producers and consumers of resources.

Constrained dependencies represent constraints on the relative flow of control among a set of activities.

ADL

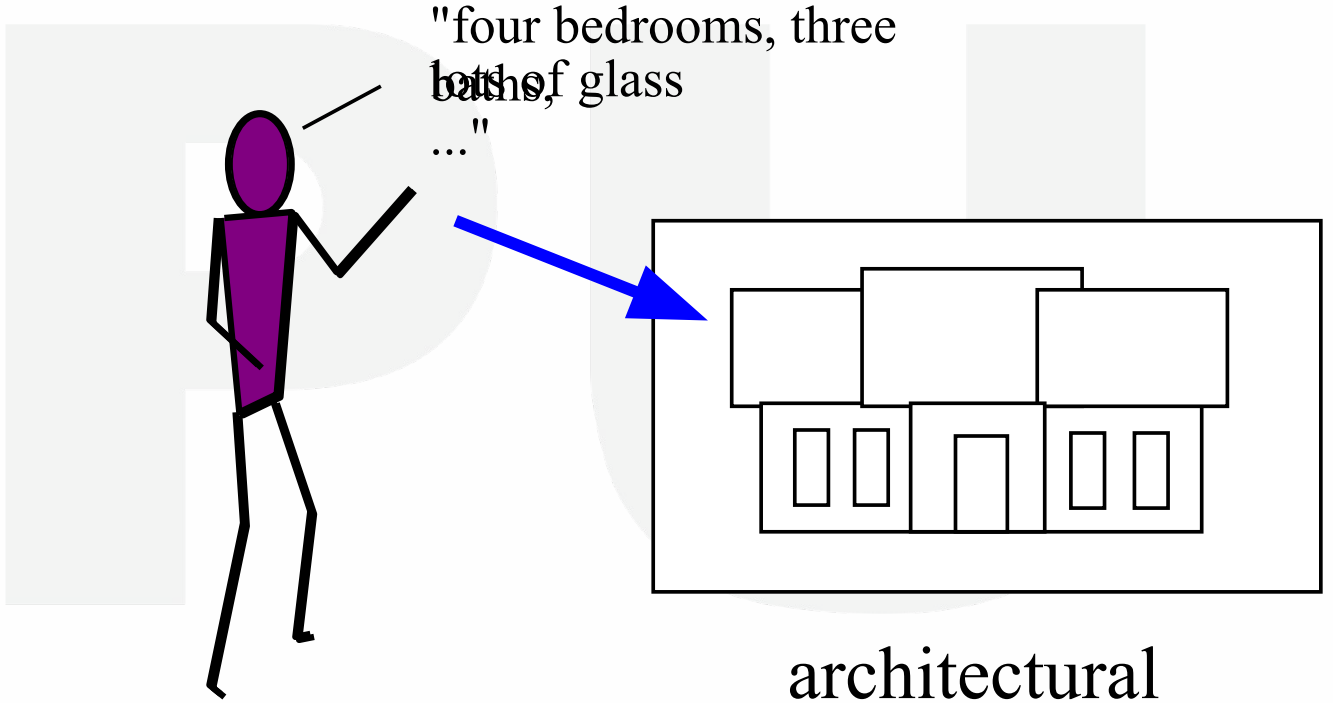
Architectural description language (ADL) provides a semantics and syntax for describing a software architecture

Provide the designer with the ability to:

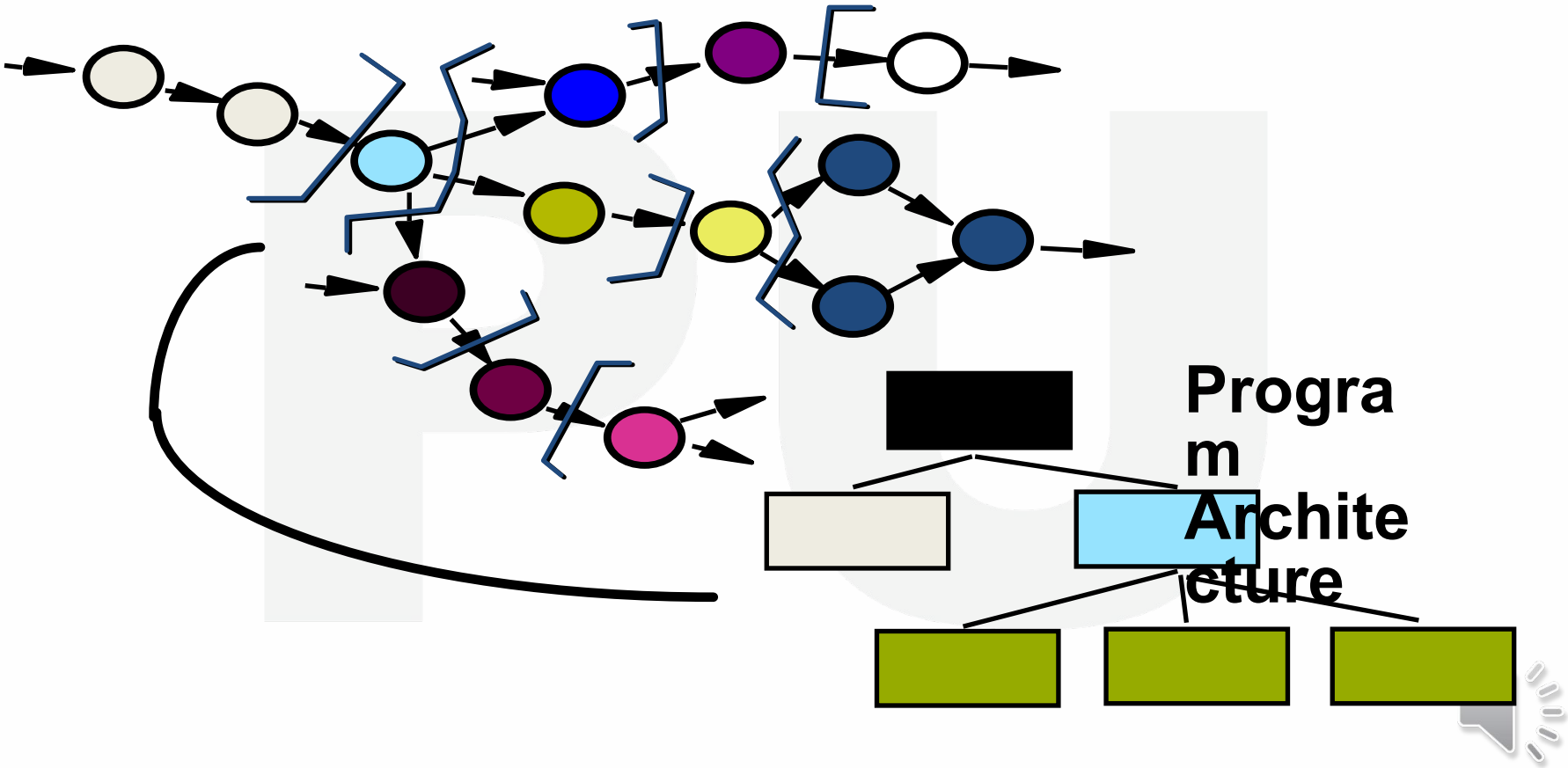
- decompose architectural components
- compose individual components into larger architectural blocks and
- represent interfaces (connection mechanisms) between components.

An Architectural Design Method

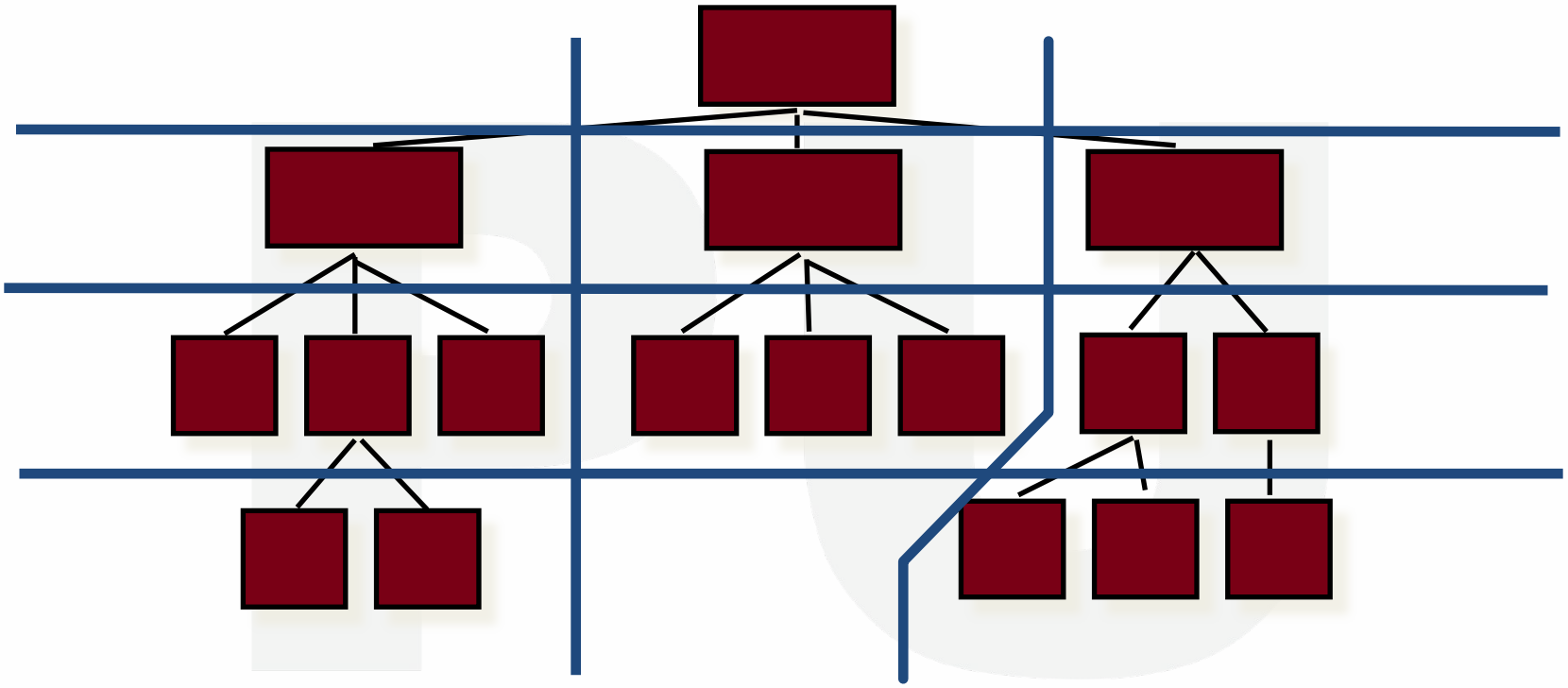
customer requirements



Deriving Program Architecture



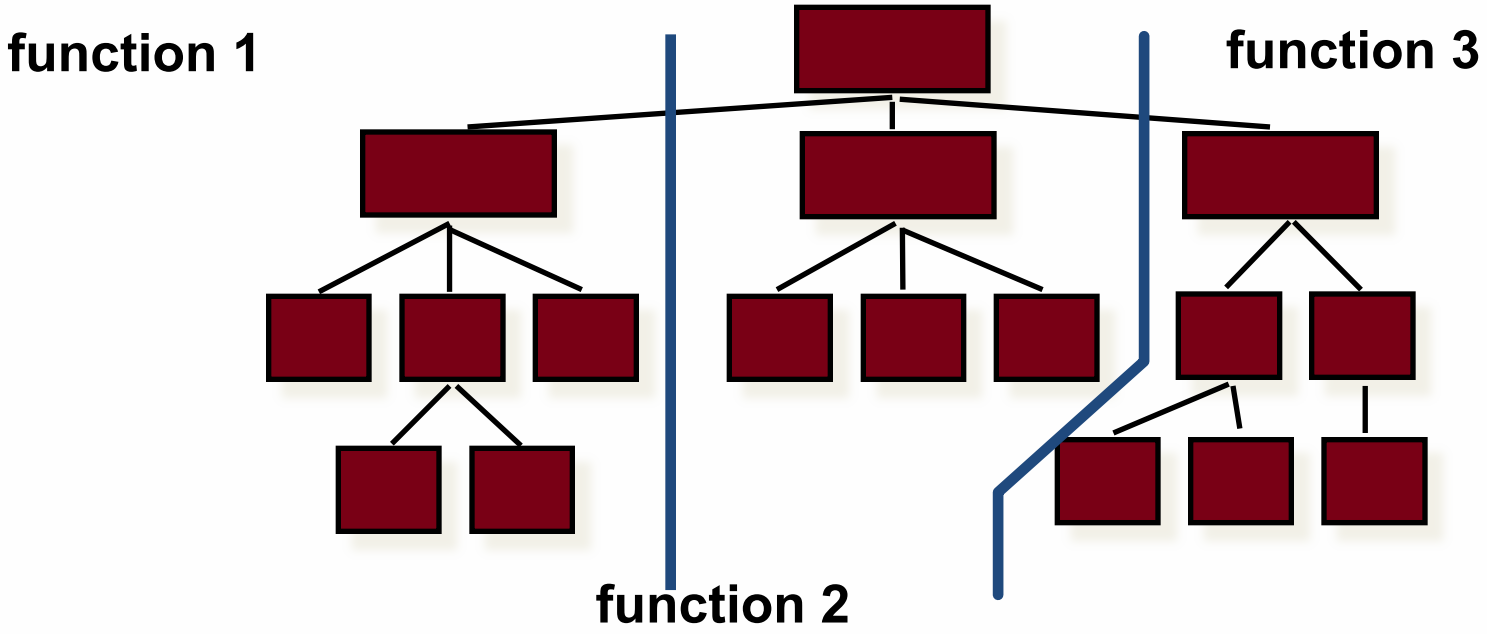
Partitioning the Architecture



"Horizontal" and "vertical" partitioning are required

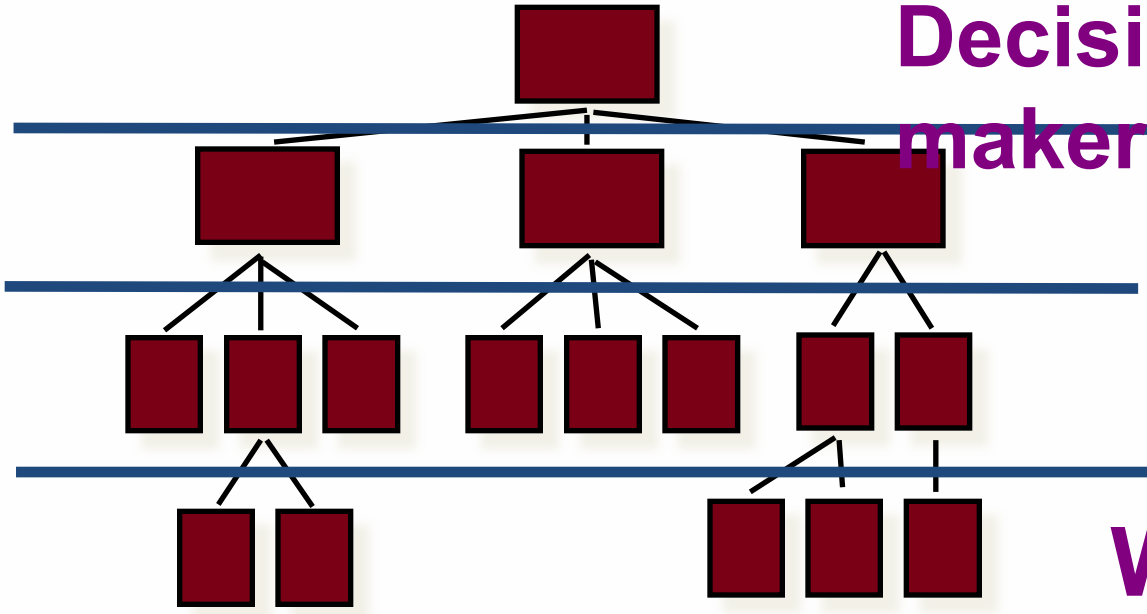
Horizontal Partitioning

- Define separate branches of the module hierarchy for each major function
- Use control modules to coordinate communication between functions



Vertical Partitioning: Factoring

- Design so that decision making and work are stratified
- Decision making modules should reside at the top of the architecture



Why Partitioned Architecture?

- Results in software that is easier to test
- Leads to software that is easier to maintain
- Results in propagation of fewer side effects
- Results in software that is easier to extend

Parul® University

DIGITAL LEARNING CONTENT

Design concept

Design Concepts, Design Model, Software Architecture, Data Design, Architectural Styles and Patterns, Architectural Design, Alternative architectural designs, Modeling Component level design and its modeling, Procedural Design, Object Oriented Design.

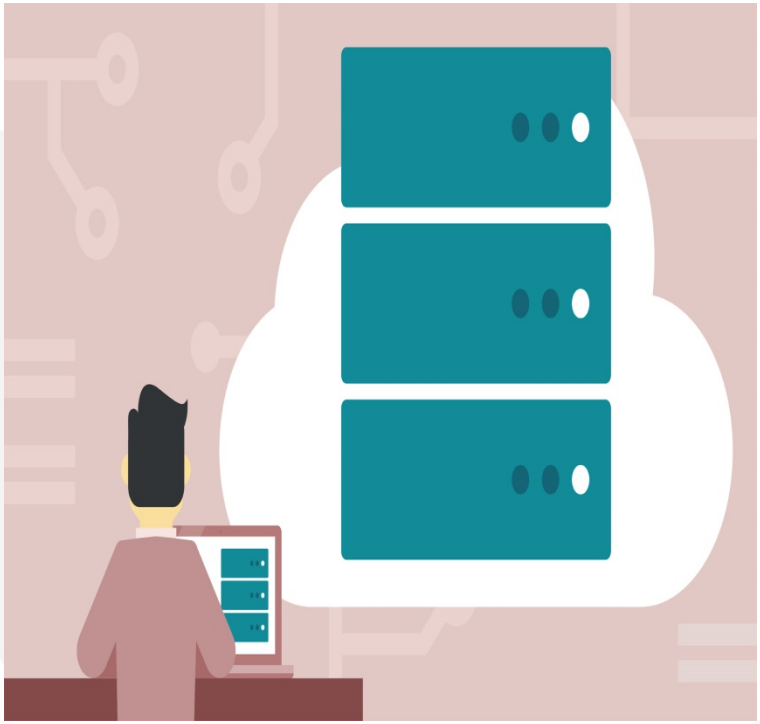


Image source : Google

Software Design

Mitch Kapor, the creator of Lotus 1-2-3, presented a "software design manifesto" in *Dr. Dobbs Journal.* He said:

Good software design should exhibit:

Firmness: A program should not have any bugs that inhibit its function.

Commodity: A program should be suitable for the purposes for which it was intended.

Delight: The experience of using the program should be pleasurable one.

Analysis Model -> Design Model

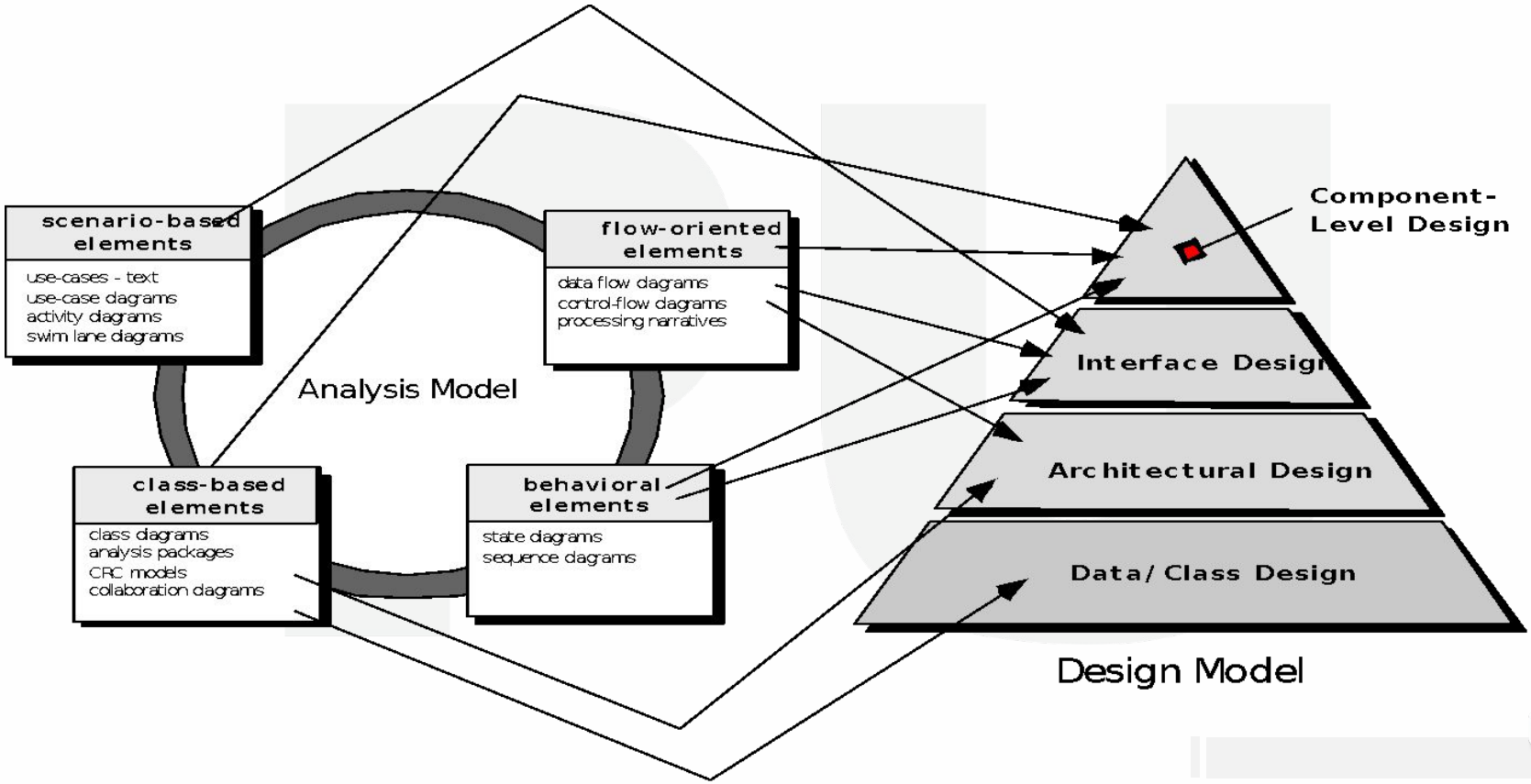


Image source :

Design and Quality

- The design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer.

The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.

The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

Fundamental Concepts

Abstraction—data, procedure, control

Architecture—the overall structure of the software

Patterns—”conveys the essence” of a proven design solution

Separation of concerns—any complex problem can be more easily handled if it is subdivided into pieces

Modularity—compartmentalization of data and function

Hiding—controlled interfaces

Functional independence—single-minded function and low coupling

Fundamental Concepts (Cont.)

Refinement—elaboration of detail for all abstractions

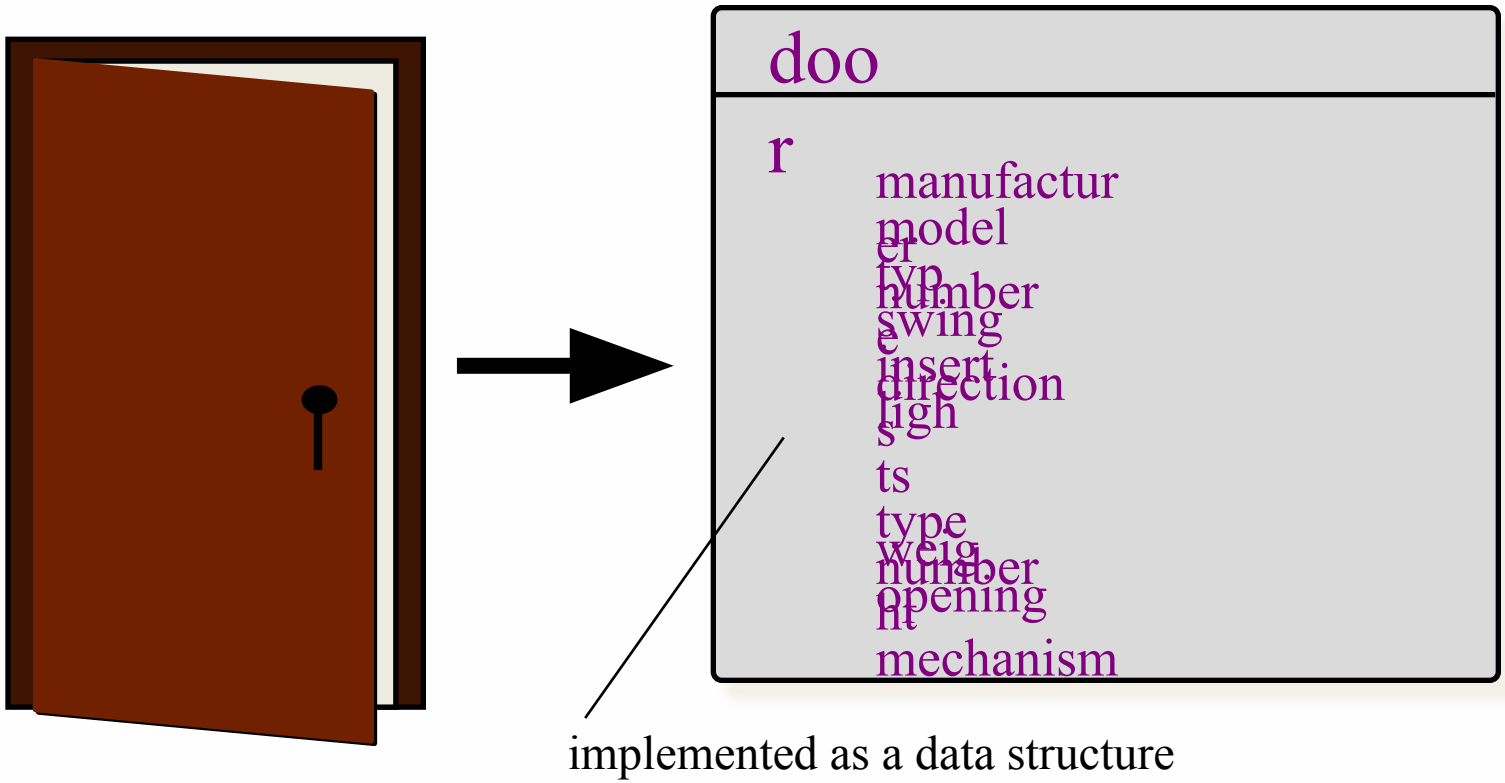
Aspects—a mechanism for understanding how global requirements affect design

Refactoring—a reorganization technique that simplifies the design

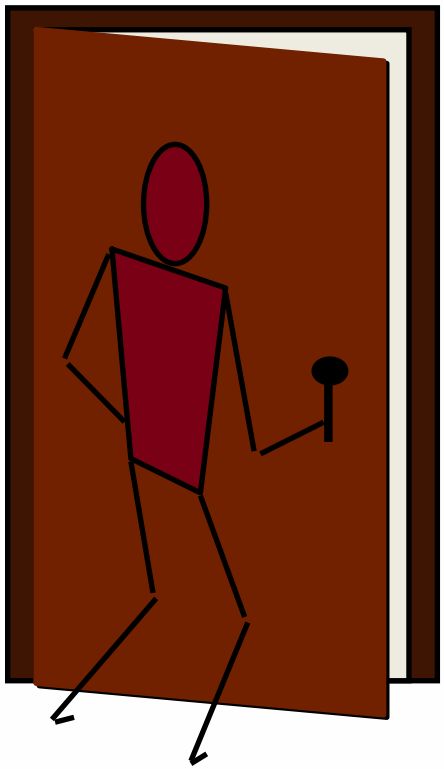
OO design concepts

Design Classes—provide design detail that will enable analysis classes to be implemented

Data Abstraction



Procedural Abstraction



ope
n
details of algorithm
enter
m

implemented with a "knowledge" of the object that is associated with enter

Architecture

“The overall structure of the software and the ways in which that structure provides conceptual integrity for a system.” [SHA95a]

- **Structural properties.** This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods

Architecture (Cont.)

- **Extra-functional properties.** The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.
- **Families of related systems.** The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.

Patterns

Design Pattern Template

- ***Pattern name*—describes the essence of the pattern in a short but expressive name**
- ***Intent*—describes the pattern and what it does**
- ***Also-known-as*—lists any synonyms for the pattern**
- ***Motivation*—provides an example of the problem**
- ***Applicability*—notes specific design situations in which the pattern is applicable**
- ***Structure*—describes the classes that are required to implement the pattern**

Modularity

"Modularity is the single attribute of software that allows a program to be intellectually manageable" [mye78].

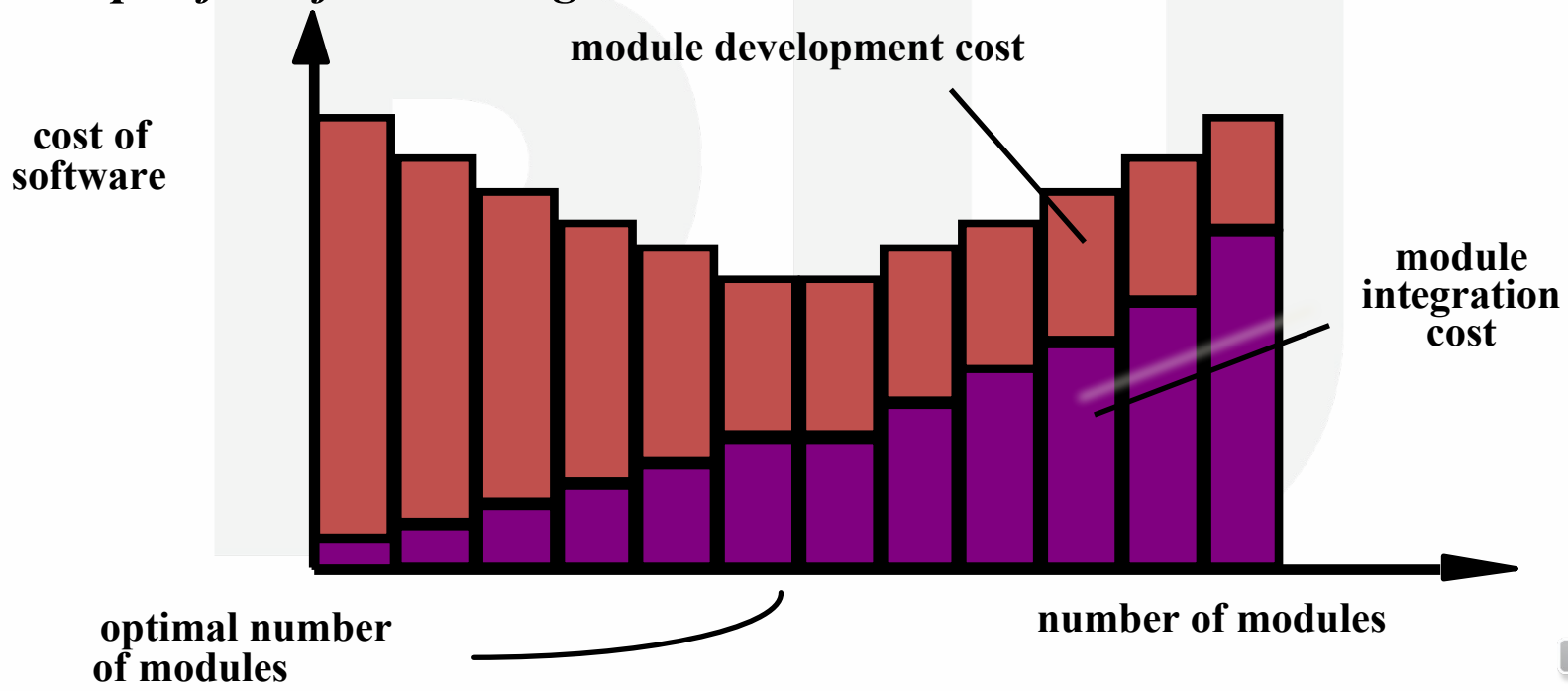
Monolithic software (i.e., A large program composed of a single module) cannot be easily grasped by a software engineer.

Modularity (Cont.)

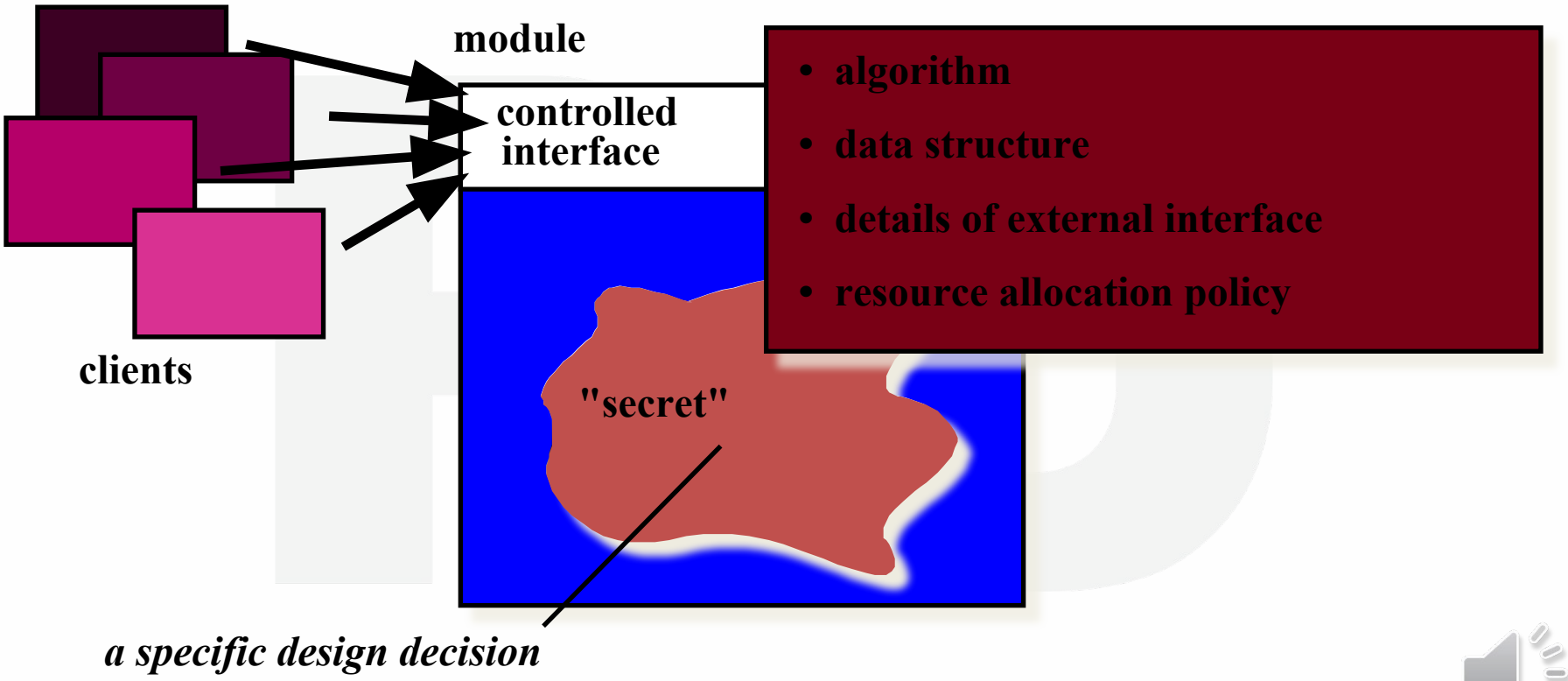
- The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible.
- In almost all instances, you should break the design into many modules, hoping to make understanding easier and as a consequence, reduce the cost required to build the software.

Modularity: Trade-offs

What is the "right" number of modules for a specific software design?



Information Hiding



Why Information Hiding?

- Reduces the likelihood of “side effects”
- Limits the global impact of local design decisions
- Emphasizes communication through controlled interfaces
- Discourages the use of global data
- Leads to encapsulation—an attribute of high quality design
- Results in higher quality software

Stepwise Refinement

open

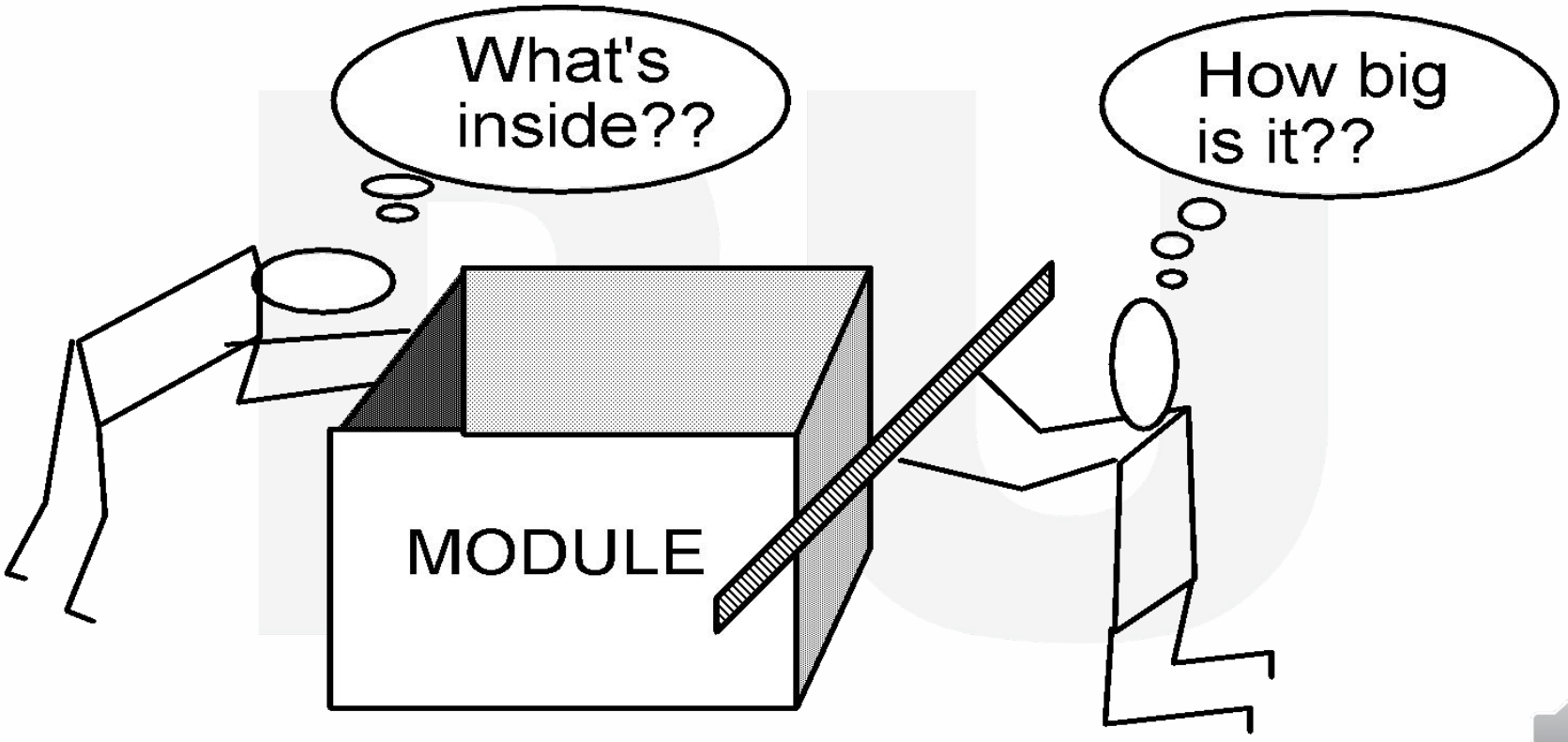
walk to door;
reach for knob;

open door;

walk through;
close door.

repeat until door opens
turn knob clockwise;
if knob doesn't turn, then
    take key out;
    find correct key;
    insert in lock;
endif
pull/push door
move out of way;
end repeat

Sizing Modules: Two Views



Parul®
University

Functional Independence

Functional independence is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules.

Functional Independence (Cont.)

Cohesion is an indication of the relative functional strength of a module.

A cohesive module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing.

Coupling is an indication of the relative interdependence among modules.

Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.

Refactoring

Fowler [FOW99] defines refactoring in the following manner:

"refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure."

OO Design Concepts

Design classes

- Entity classes
- Boundary classes
- Controller classes

Inheritance—all responsibilities of a superclass is immediately inherited by all subclasses

Messages—stimulate some behavior to occur in the receiving object

Polymorphism—a characteristic that greatly reduces the effort required to extend the design

Design Classes

Analysis classes are refined during design to become entity classes

- Boundary classes are developed during design to create the interface (e.g., Interactive screen or printed reports) that the user sees and interacts with as the software is used.

  Boundary classes are designed with the responsibility of managing the way entity objects are represented to users.

Design Classes (Cont.)

- Controller classes are designed to manage

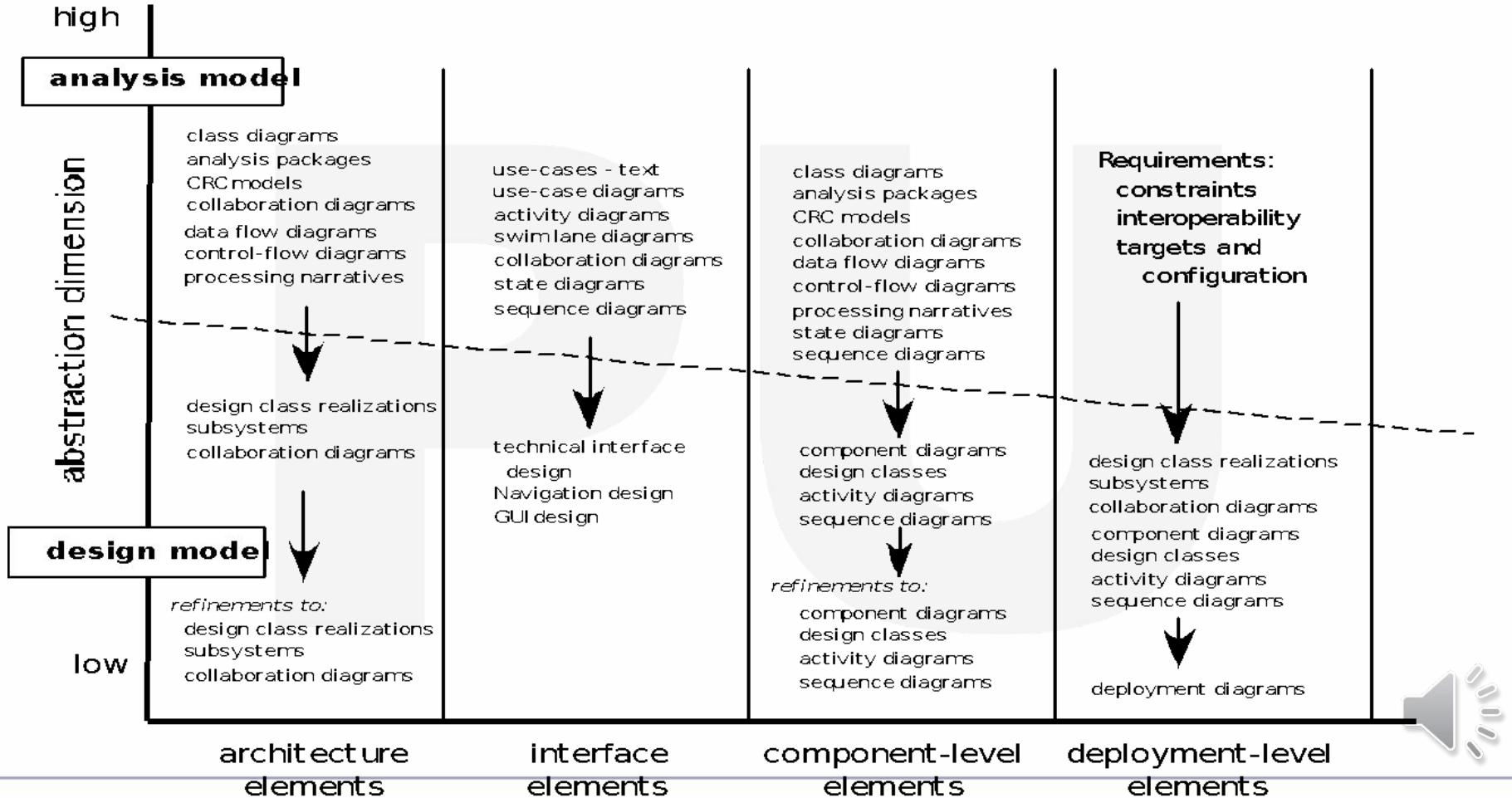
The creation or update of entity objects;

the instantiation of boundary objects as they obtain information from entity objects;

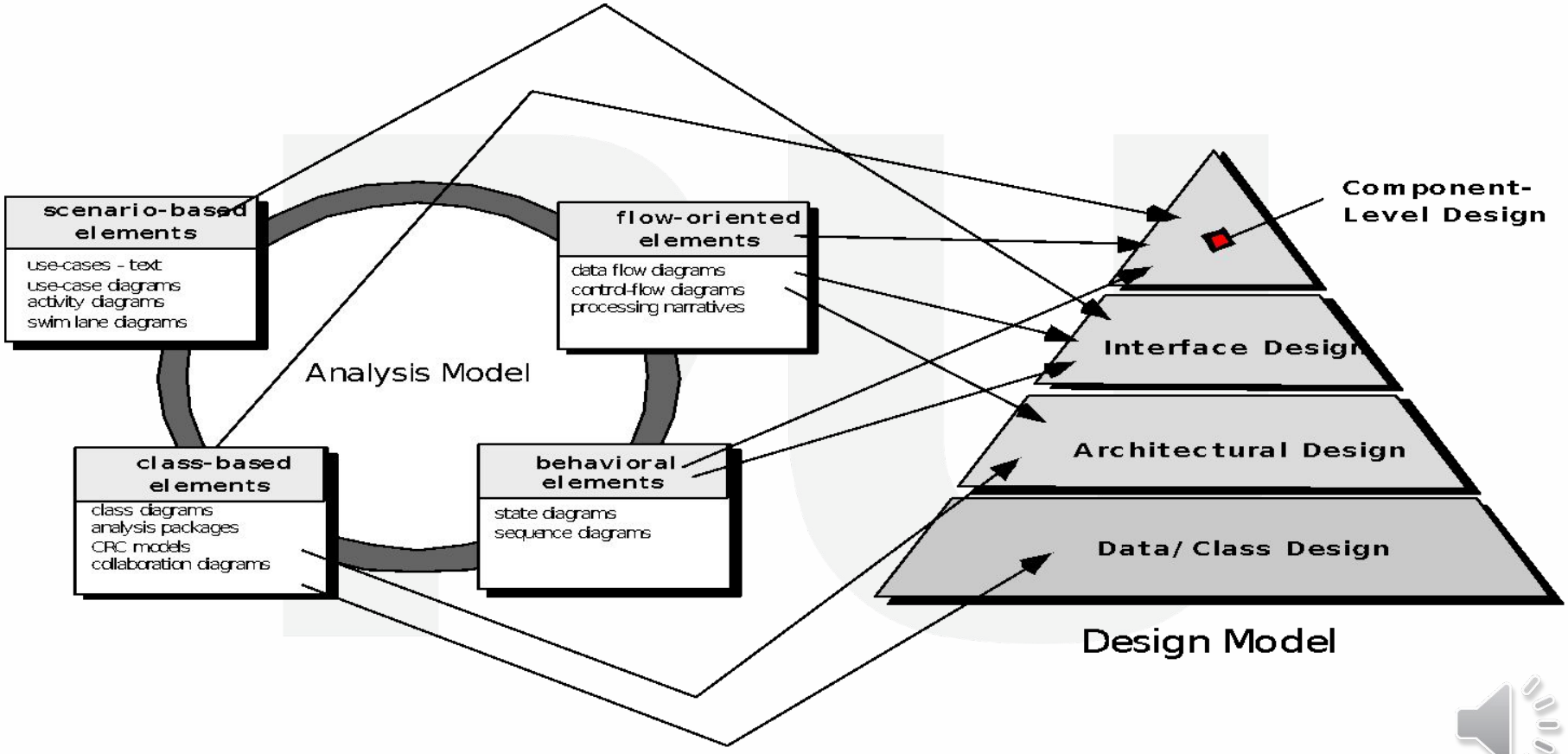
complex communication between sets of objects;

validation of data communicated between objects or between the user and the application.

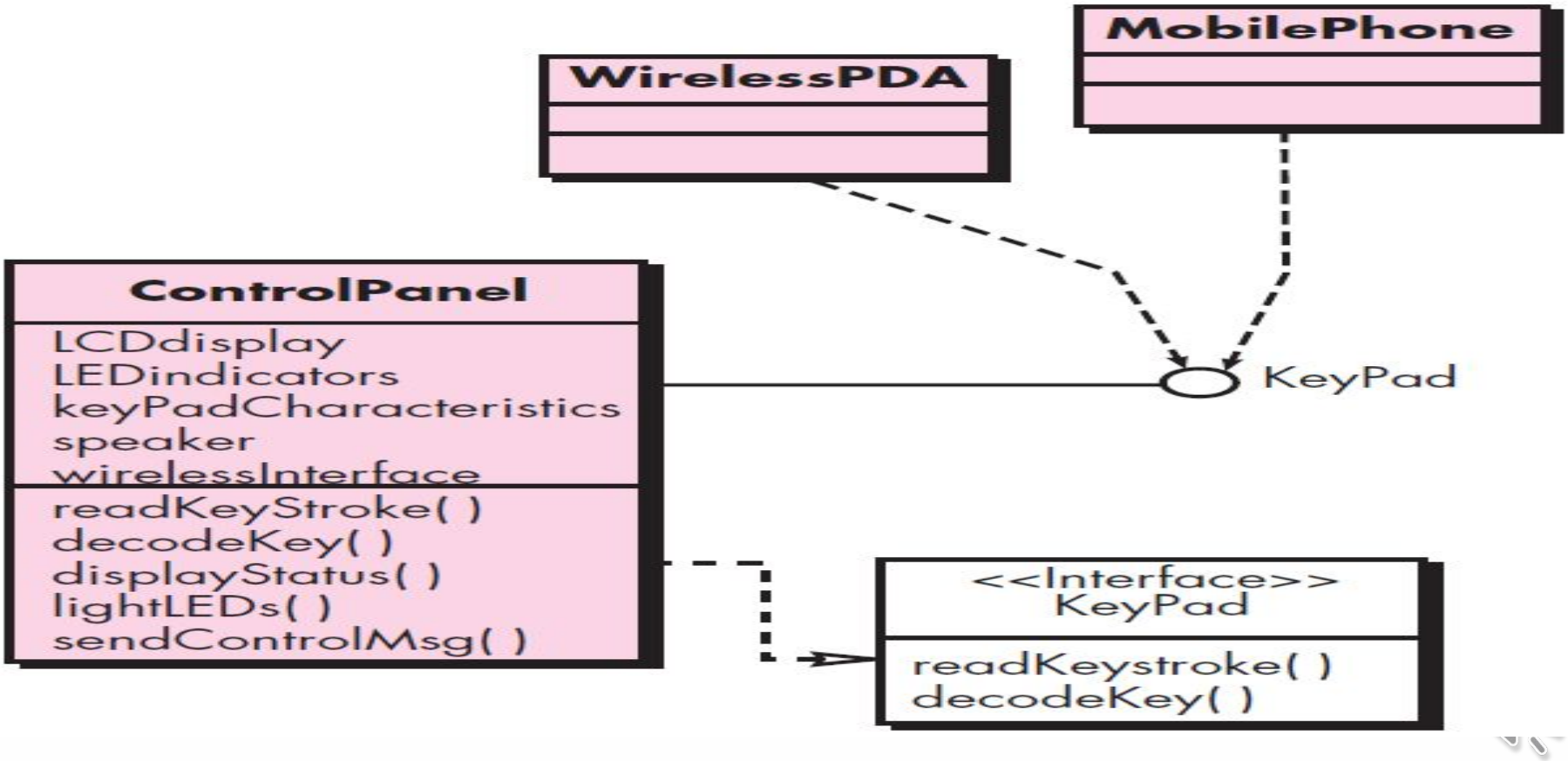
The Design Model



Design Model Elements



Interface Elements



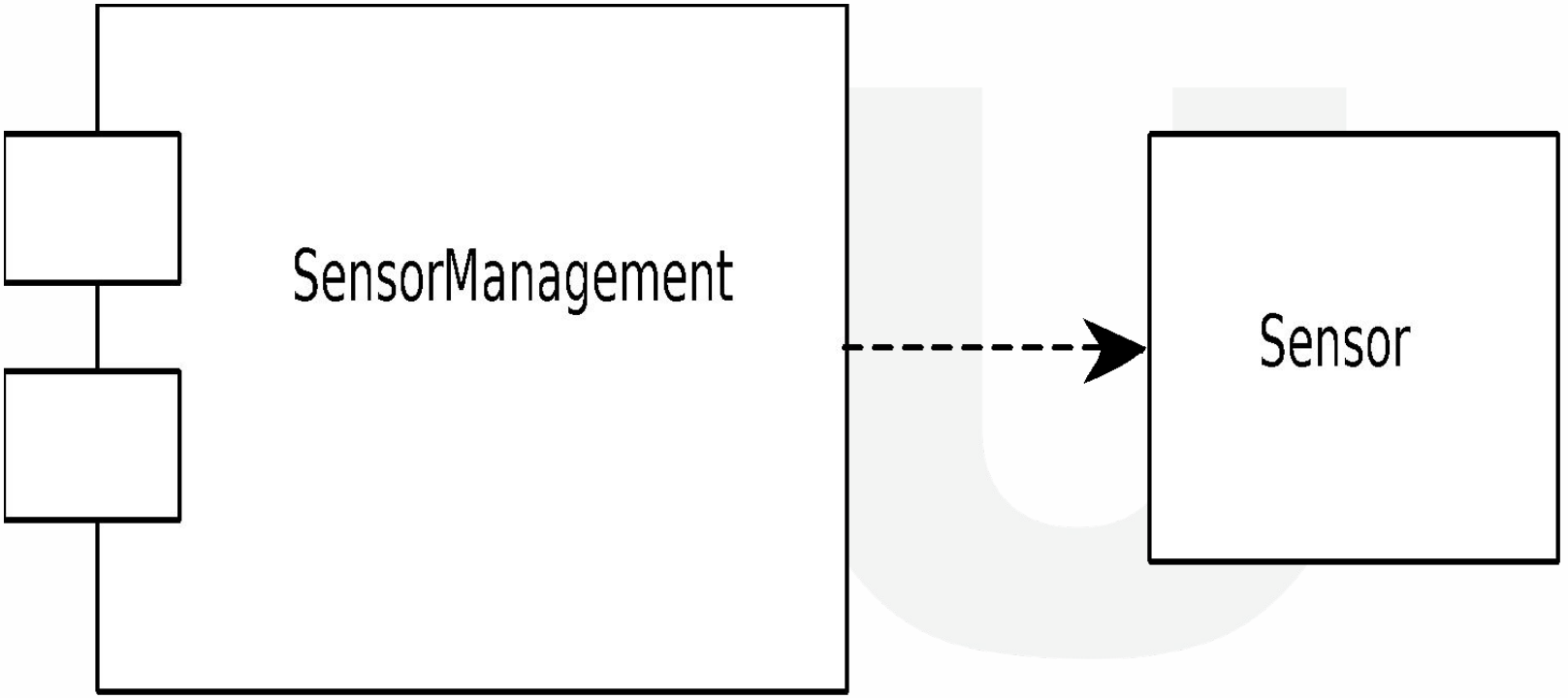
Component-Level Design Elements

The component-level design for software is the equivalent to a set of detailed drawings (and specifications) for each room in a house.

The component-level design for software fully describes the internal detail of each software component.

To accomplish this, the component-level design defines data structures for all local data objects and algorithmic detail for all processing.

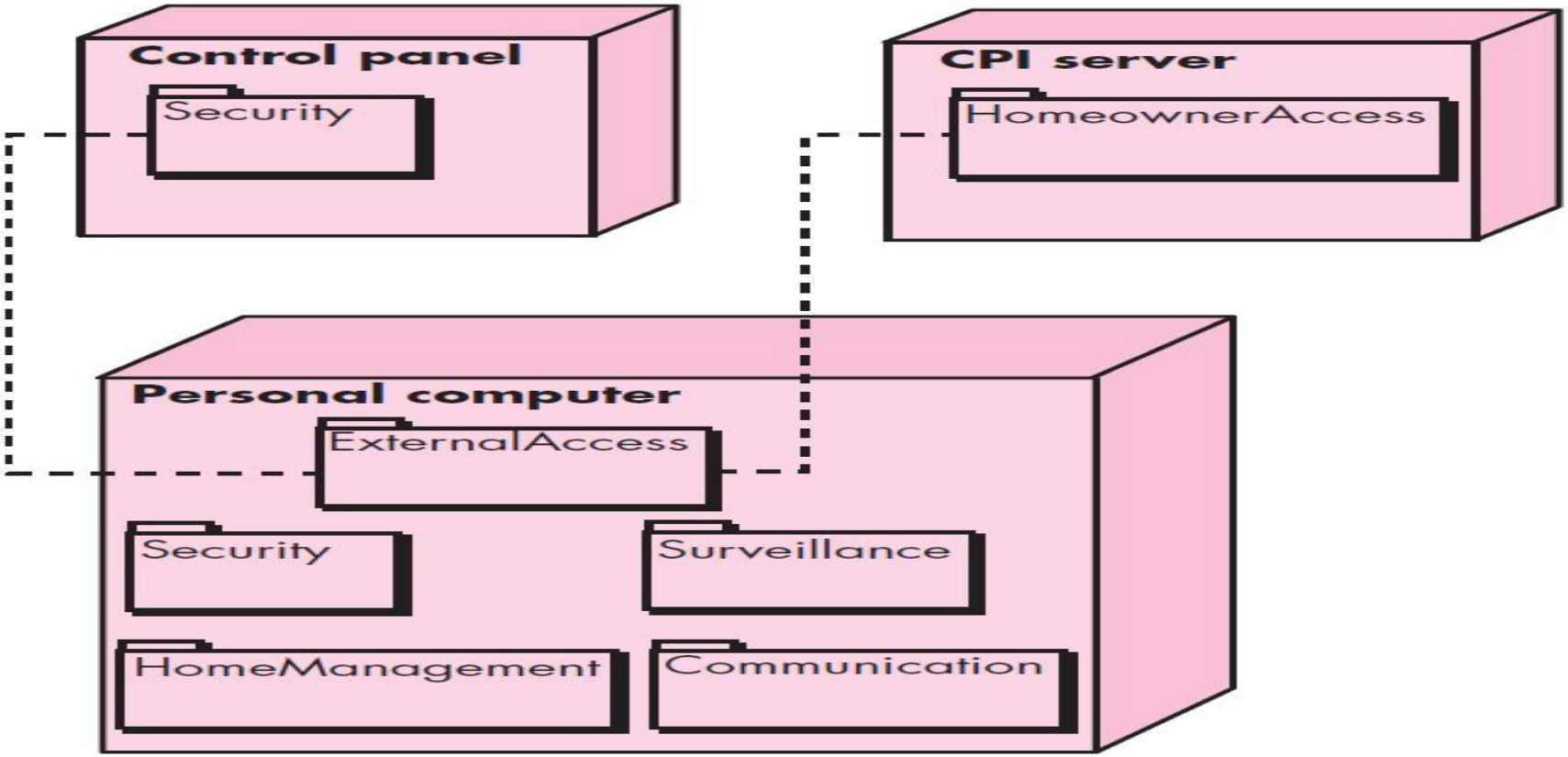
Component Elements



Deployment Elements

Deployment-level design elements indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software.

Deployment Elements (Cont.)



Design Guidelines

Components

Naming conventions should be established for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model

Interfaces

Interfaces provide important information about communication and collaboration (as well as helping us to achieve the OPC)

Dependencies and inheritance

It is a good idea to model dependencies from left to right and inheritance from bottom (derived classes) to top (base classes).

Cohesion & Coupling

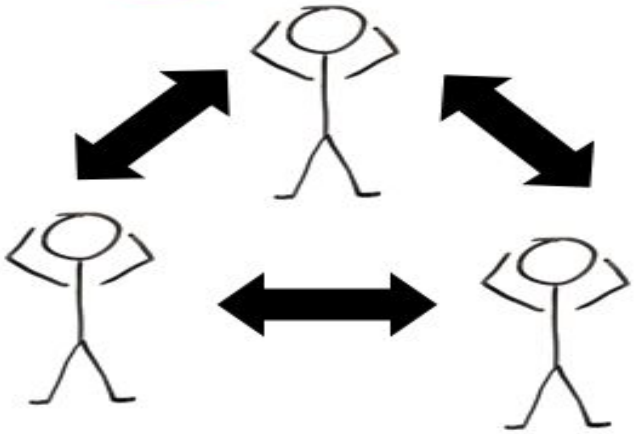
A **good software design** implies **clean decomposition** of the **problem into modules**, and the **neat arrangement** of these **modules** in a **hierarchy**.

The primary **characteristics** of **neat module decomposition** are **high cohesion** and **low coupling**.

Cohesion & Coupling

A **cohesive module** performs a single task, requiring little interaction with other components.

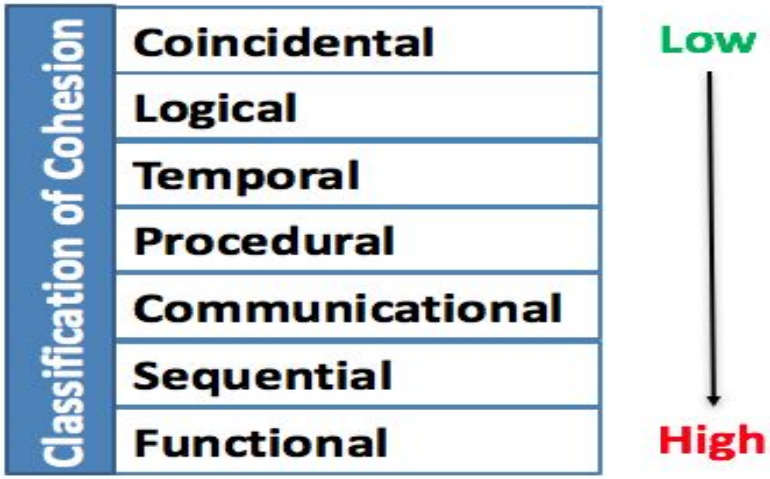
A **Coupling** is an indication of the relative interdependence among modules.



Cohesion

- Cohesion is an **indication** of the **relative functional strength** of a module.
- A **cohesive module** performs a **single task**, **requiring little interaction** with other components.
- Stated simply, a **cohesive module** should (ideally) **do just one thing**.
- A module having **high cohesion** and **low coupling** is said to be **functionally independent** of other modules.
- By the term functional independence, we mean that a **cohesive module performs a single task or function**.

Classification of Cohesion



Classification of Cohesion cont.

Coincidental cohesion

- A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely.
- In this case, the module contains a random collection of functions.
- It is likely that the functions have been put in the module out of pure coincidence without any thought or design.
- For Ex., in a transaction processing system (TPS), the get-input, print-error, and summarize-members functions are grouped into one module.

Classification of Cohesion cont.

Logical cohesion

- A module is said to be logically cohesive, if all elements of the module perform similar operations.
- For Ex., error handling, data input, data output, etc.
- An example of logical cohesion is the case where a set of print functions generating different output reports are arranged into a single module.

Temporal cohesion

- When a module contains functions that are related by the fact that all the functions must be executed in the same time span.
- For Ex., the set of functions responsible for initialization, start-up, shutdown of some process, etc.

Classification of Cohesion cont.

Procedural cohesion

- If the set of functions of the module are all part of a procedure (algorithm) in which certain sequence of steps have to be carried out for achieving an objective
- For Ex., the algorithm for decoding a message.

Communicational cohesion

- If **all functions** of the module **refer** to the **same data structure**
- For Ex., the set of functions defined on an array or a stack.

Classification of Cohesion cont.

Sequential cohesion

- If the elements of a module form the parts of sequence, where the output from one element of the sequence is input to the next.
- For Ex., In a Transaction Processing System, the get-input, validate-input, sort-input functions are grouped into one module.

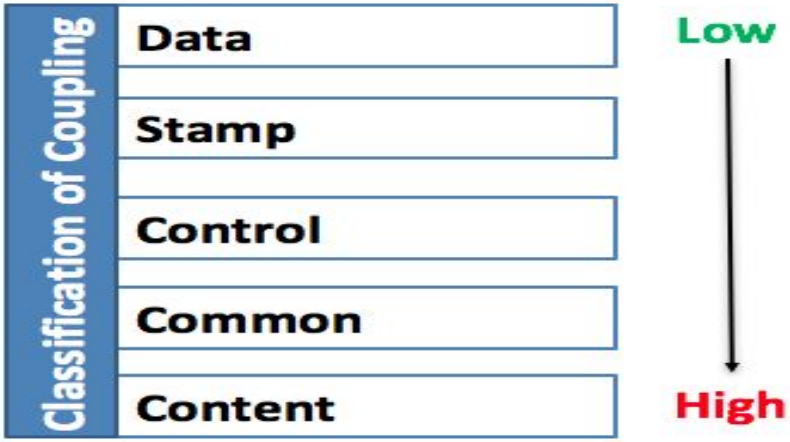
Functional cohesion

- If different elements of a module cooperate to achieve a single function.
- For Ex., A module containing all the functions required to manage employees' pay-roll exhibits functional cohesion.

Coupling

- **Coupling** between two modules is a **measure of the degree** of **interdependence** or interaction **between** the **two modules**.
- A module having high cohesion and low coupling is said to be functionally independent of other modules.
- If **two modules interchange large amounts of data**, then they are **highly interdependent**.
- The degree of coupling between two modules depends on their interface complexity.
- The interface complexity is basically determined by the number of types of parameters that are interchanged while invoking the functions of the module.

Classification of Coupling



Classification of Coupling Cont.

Data coupling

- Two modules are data coupled, if **they communicate through a parameter**.
- An example is an elementary (primal) data item passed as a parameter between two modules, e.g. an integer, a float, a character, etc.

Stamp coupling

- This is a special case (or extension) of data coupling
- Two modules (``A'' and ``B'') exhibit stamp coupling if **one passes** directly to the other a **composite data item** - such as a record (or structure), array, or (pointer to) a list or tree.
- This occurs when **ClassB** is **declared as a type** for an argument of an **operation of ClassA**

Classification of Coupling Cont.

Control coupling

- If **data** from **one** module **is used to direct the order of instructions** execution **in another**.
- An example of control coupling is a flag set in one module and tested in another module.

Common coupling

- Two modules are common coupled, if they **share data** through some **global data items**.
- Common coupling can **leads to uncontrolled error propagation** and **unforeseen side effects** when changes are made.

Classification of Coupling Cont.

Content coupling

- Content coupling occurs when **one component secretly modifies data** that is **internal to another component.**
- This violets information hiding – a basic design concept
- Content coupling exists between two modules, if they share code.

Parul®
University

DIGITAL LEARNING CONTENT

Algorithm Design

The closest design activity to coding

The approach:

- Review the design description for the component
- Use stepwise refinement to develop algorithm
- Use structured programming to implement procedural logic
- Use ‘formal methods’ to prove logic

Stepwise Refinement

open

walk to door;
reach for knob;

open door;

walk through;
close door.

repeat until door opens
turn knob clockwise;
if knob doesn't turn, then
    take key out;
    find correct key;
    insert in lock;
endif
pull/push door
move out of way;
end repeat

Parul®
University

CBSE Activities

- Component qualification
- Component adaptation
- Component composition
- Component update

Classifying and Retrieving Components

Enumerated classification—components are described by defining a hierarchical structure in which classes and varying levels of subclasses of software components are defined

Faceted classification—a domain area is analyzed and a set of basic descriptive features are identified

Attribute-value classification—a set of attributes are defined for all components in a domain area

UI Design

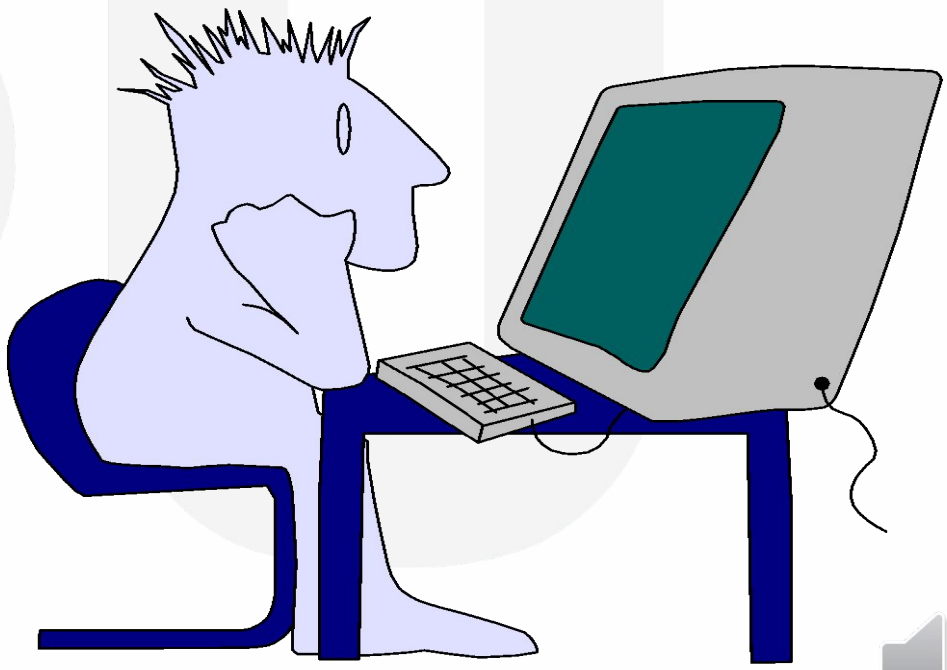
User Interface Design

Interface Design

Easy to learn?

Easy to use?

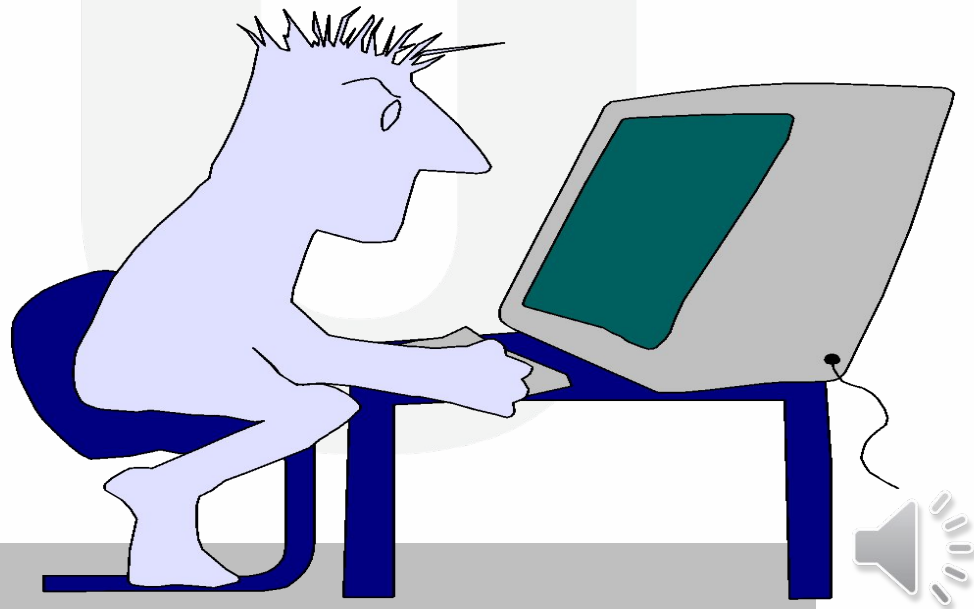
Easy to understand?



Interface Design

Typical Design Errors

- **lack of consistency**
- **too much memorization**
- **no guidance / help**
- **no context**



Golden Rules of User Interface Design

Place the User in Control	Reduce the User's Memory Load
	

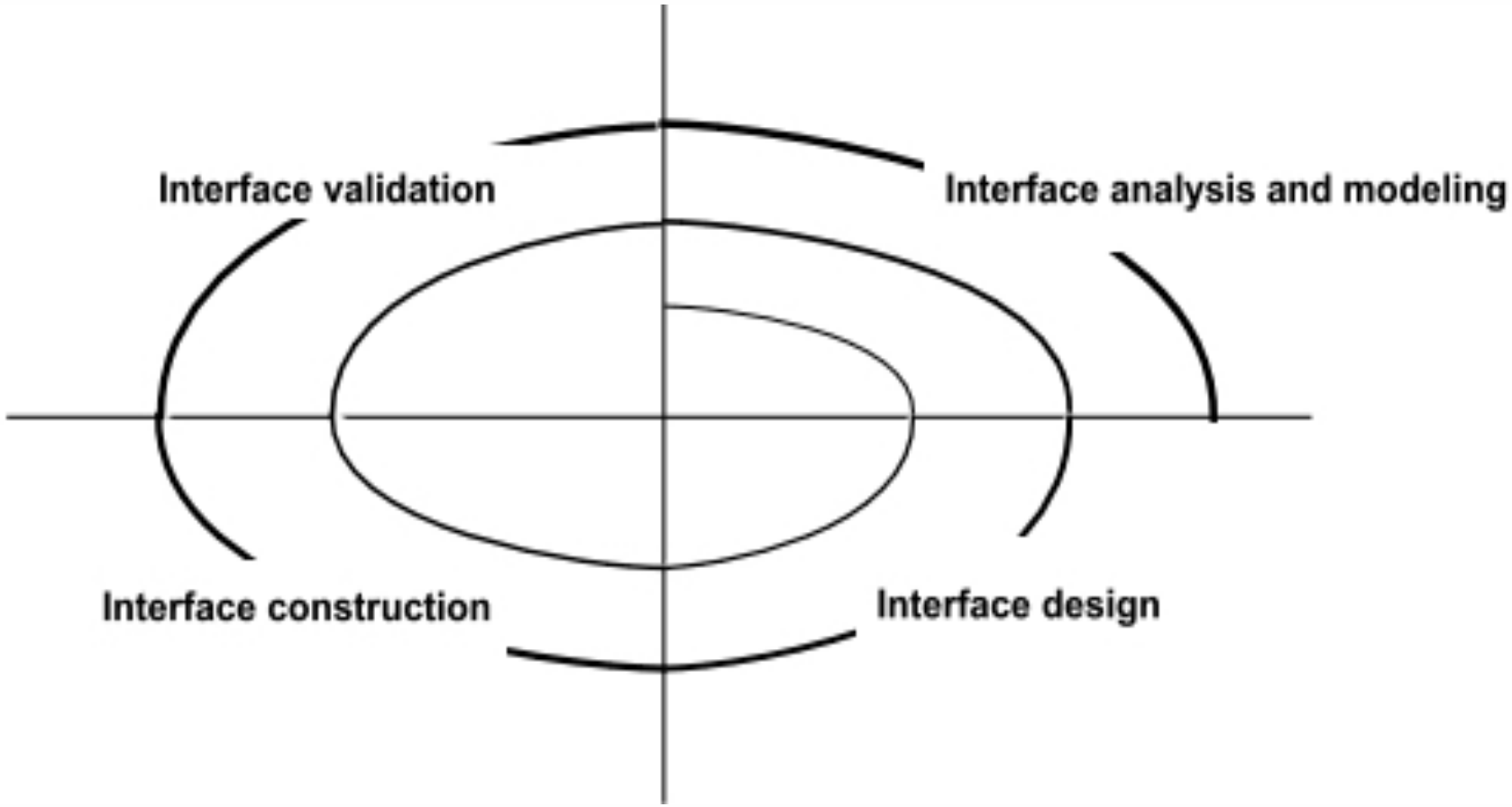
Make the Interface Consistent

Microsoft Word

Microsoft Excel

Microsoft Powerpoint

User Interface Design Process



Interface Analysis

Interface analysis means understanding

(1) the people (end-users) who will interact with the system through the interface;

(2) the tasks that end-users must perform to do their work,

(3) the content that is presented as part of the interface

(4) the environment in which these tasks will be conducted.

Task Analysis and Modeling

- The goal of task analysis is to answer the following questions:
  - What work will the user perform in specific circumstances?
  - What tasks and subtasks will be performed as the user does the work?
  - What specific problem domain objects will the user manipulate as work is performed?
  - What is the sequence of work tasks—the workflow?
  - What is the hierarchy of tasks?

Task Analysis and Modeling (Cont.)

To answer these questions, many techniques are discussed, but in this instance, these techniques are applied to the user interface.

Use-cases define basic interaction

Task elaboration refines interactive tasks

Object elaboration identifies interface objects (classes)

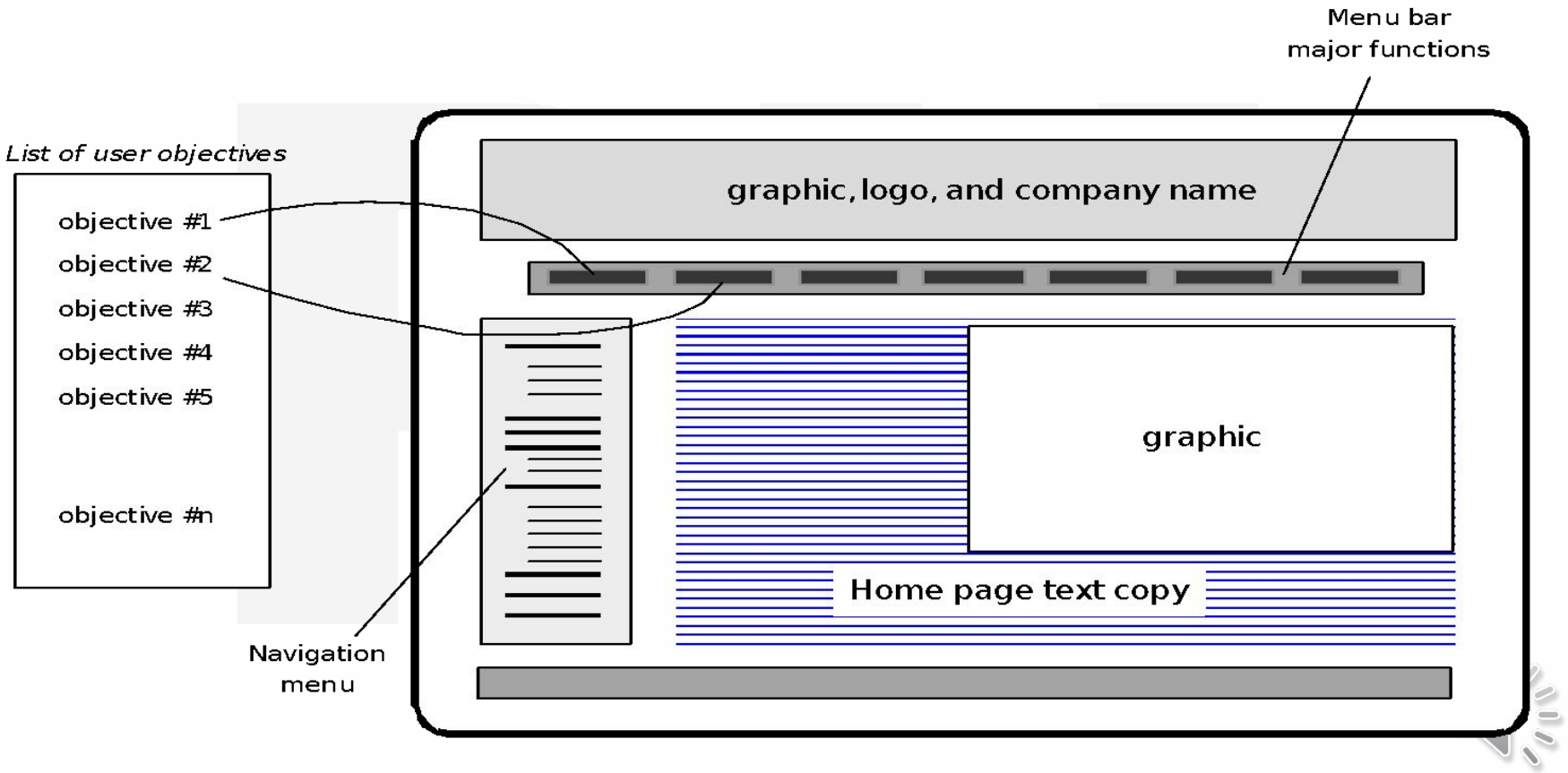
Workflow analysis defines how a work process is completed when several people (and roles) are involved

Parul®
University

User Interface Design Patterns: Design Issues

- Response time
- Help facilities
- Error handling
- Menu and command labeling
- Application accessibility

Mapping User Objectives



Parul®
University

DIGITAL LEARNING CONTENT

Aesthetic Design

- Don’t be afraid of white space.
- Emphasize content.
- Organize layout elements from top-left to bottom right.
- Group navigation, content, and function geographically within the page.
- Don’t extend your real estate with the scrolling bar.
- Consider resolution and browser window size when designing layout.

Parul®
University

DIGITAL LEARNING CONTENT

Data Oriented Analysis & Design

Difference between Data and Information, E-R Diagram, Dataflow Model, Control Flow Model, Control and Process Specification, Data Dictionary

Data

So what is Data?

Data is raw, unorganized facts that need to be processed. Data can be something simple and apparently random and useless until it is organized.

Parul®
University

DIGITAL LEARNING CONTENT

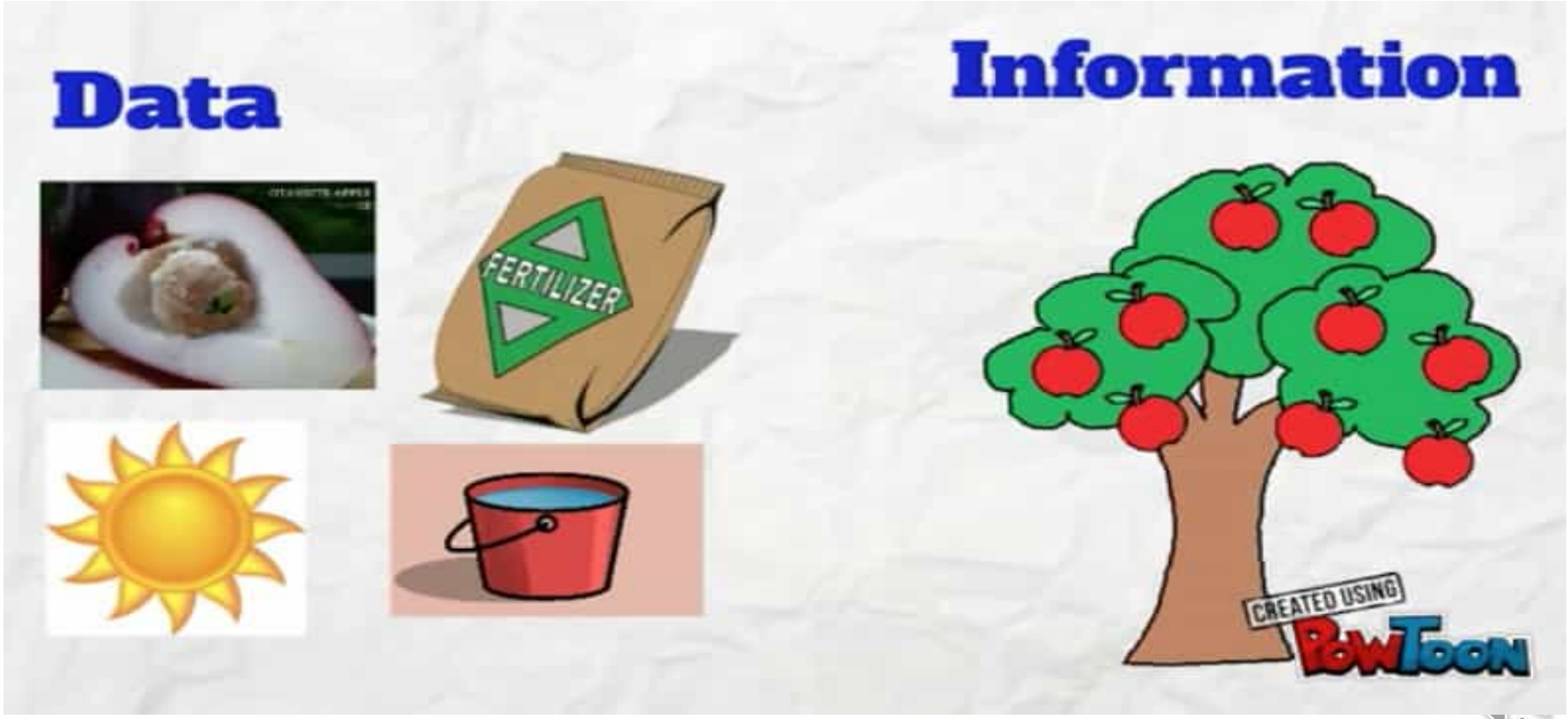
Information

What is information?

When data is processed, organized, structured or presented in a given context so as to make it useful, it is called information.

CREATED USING PowToon

Data & Information



Parul® University

E-R Model

- It is the most conceptual model used for designing a database.
- Proposed by Dr. Peter Chen in 1960
- E-R model views the real world as a set of basic objects (entities) , their characteristics (attributes) and associations among objects (relationship).
- Entities , Attributes & relationship are the basic concept of an E-R model.

Entity

An **entity** is an object that exists and is distinguishable from other objects.

Example: specific person, company, event, plant

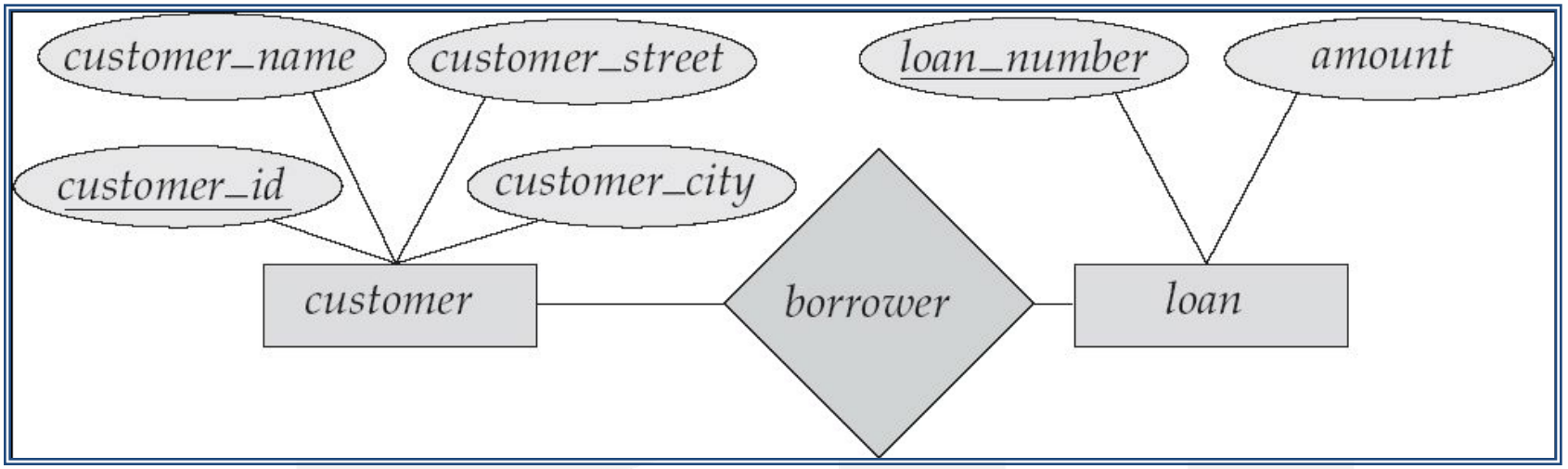
Entities have *attributes*

Example: people have *names* and *addresses*

An **entity set** is a set of entities of the same type that share the same properties.

Example: set of all persons, companies, trees, holidays

E-R Diagrams



Rectangles represent entity sets.

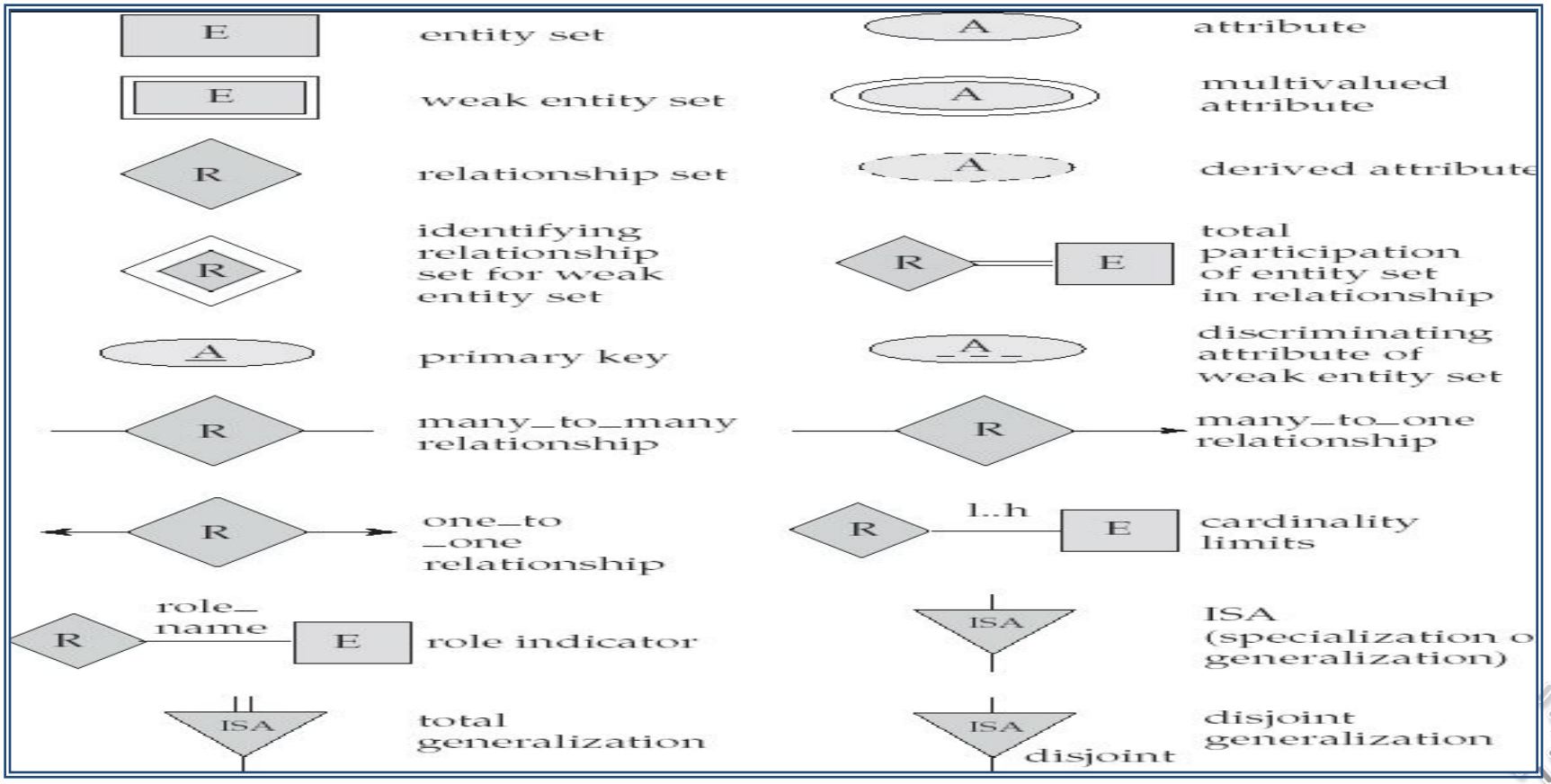
Diamonds represent relationship sets.

Lines link attributes to entity sets and entity sets to relationship sets.

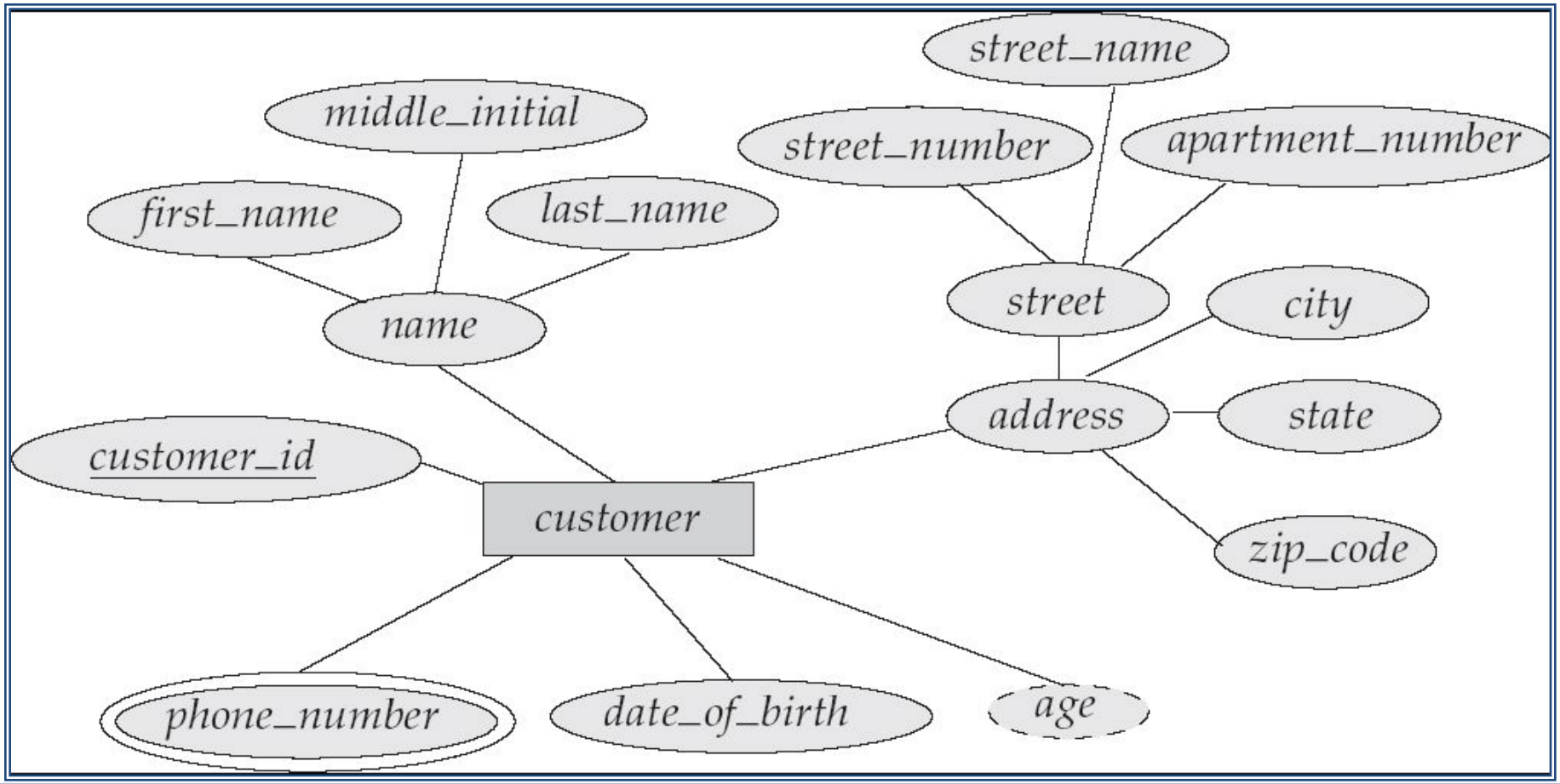
Ellipses represent attributes

Double ellipses represent multivalued attributes.

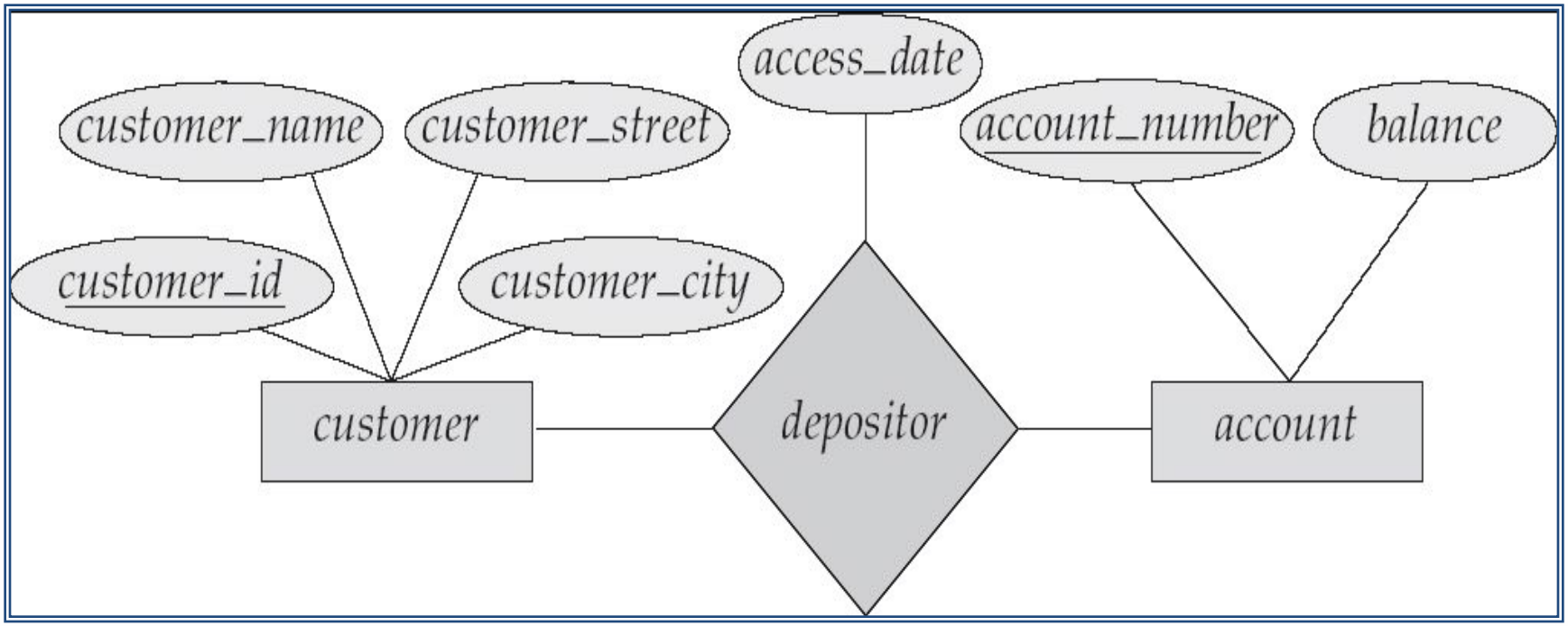
E-R Diagram Notations



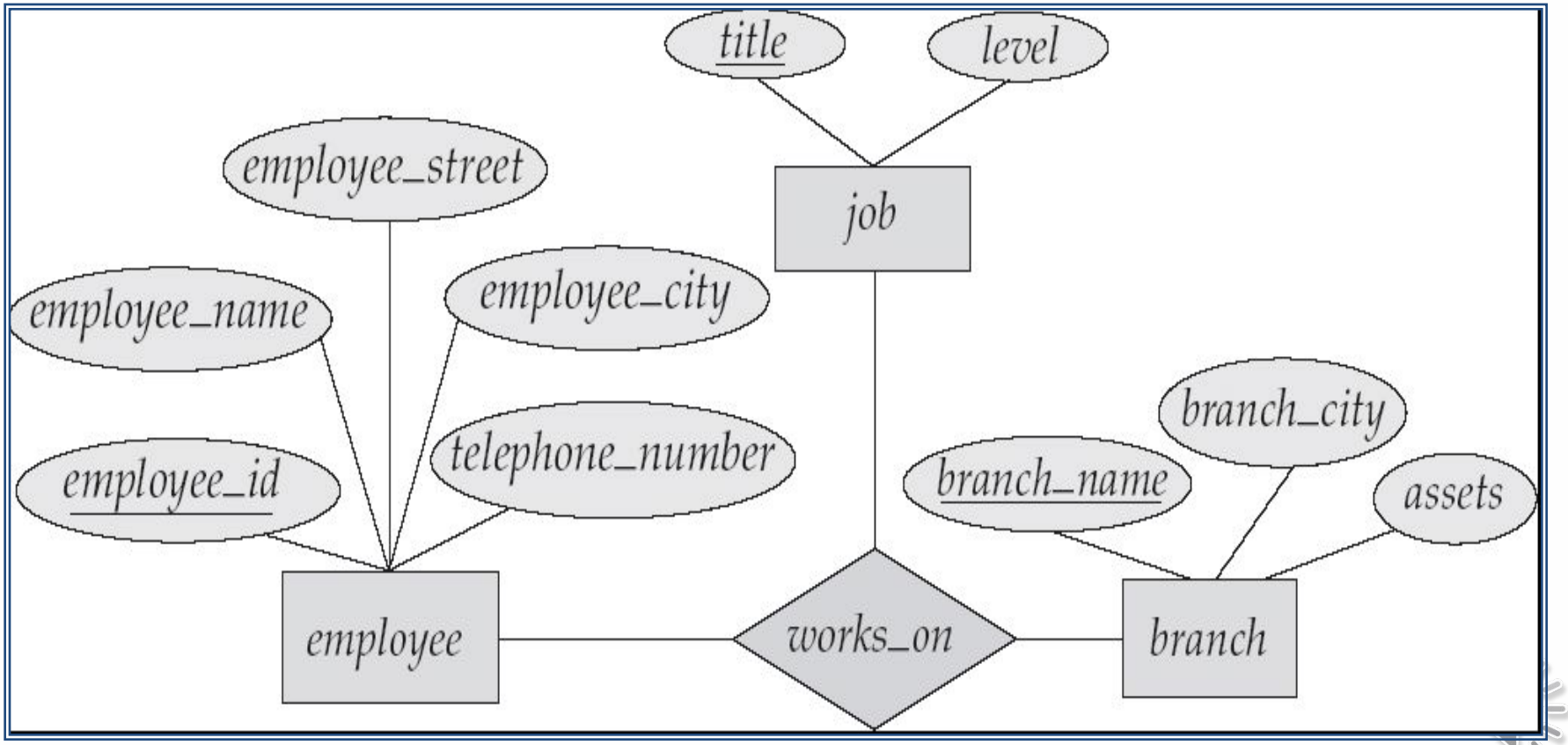
E-R Diagram With Composite, Multivalued, and Derived Attributes



Relationship Sets with Attributes



E-R Diagram with a Ternary Relationship



Cardinality Constraints

We express cardinality constraints by drawing either a directed line (→), signifying “one,” or an undirected line (—), signifying “many,” between the relationship set and the entity set.

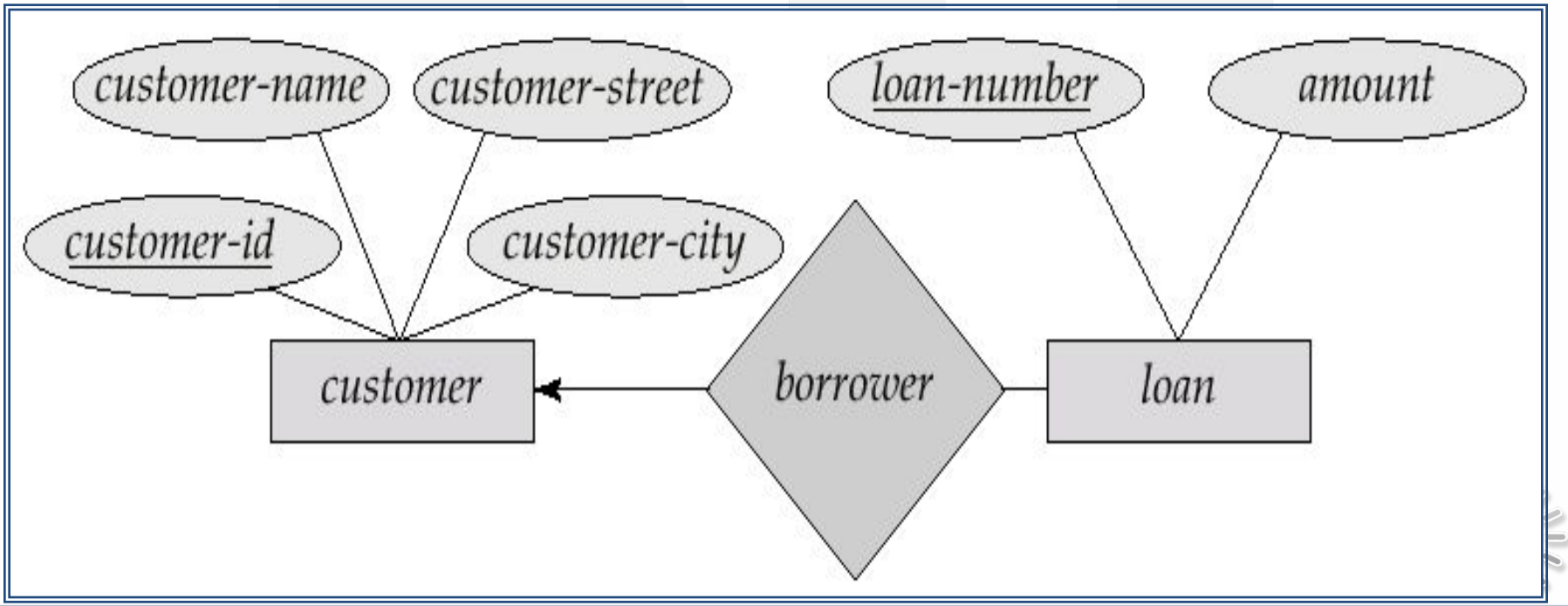
One-to-one relationship:

A customer is associated with at most one loan via the relationship *borrower*

A loan is associated with at most one customer via *borrower*

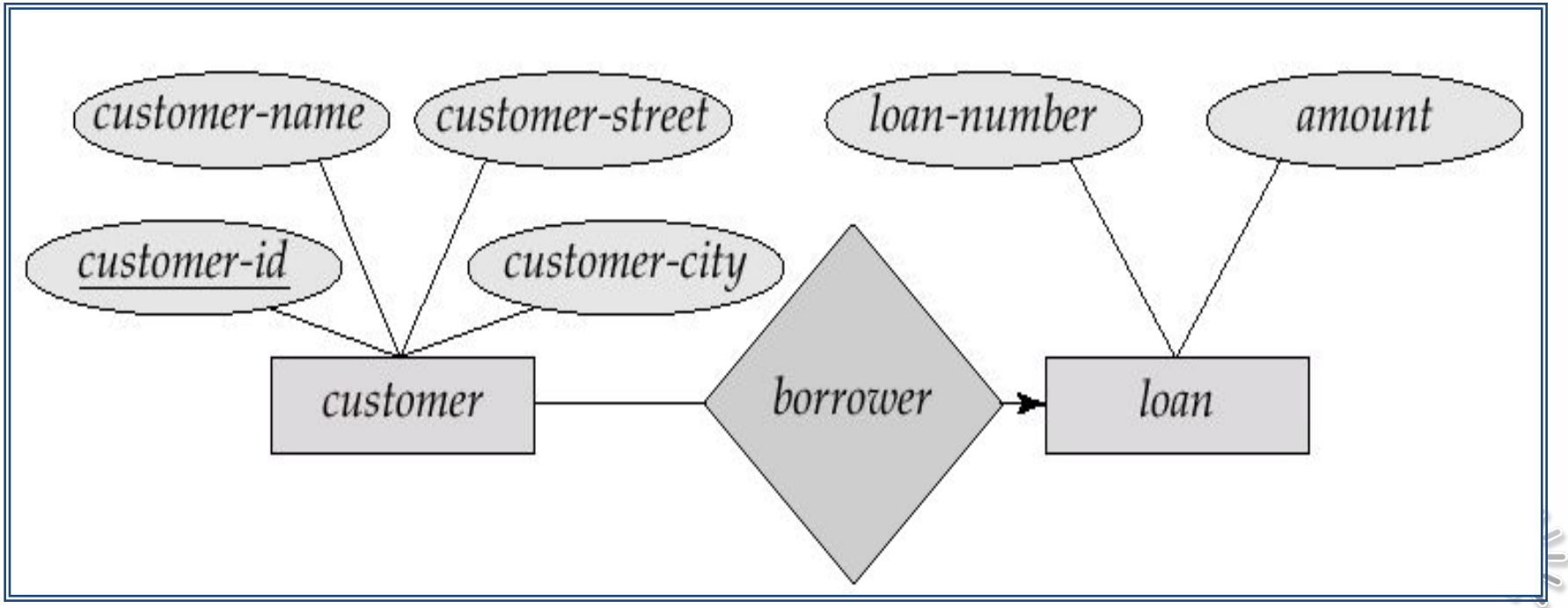
One-To-Many Relationship

In the one-to-many relationship a loan is associated with at most one customer via *borrower*, a customer is associated with several (including 0) loans via *borrower*



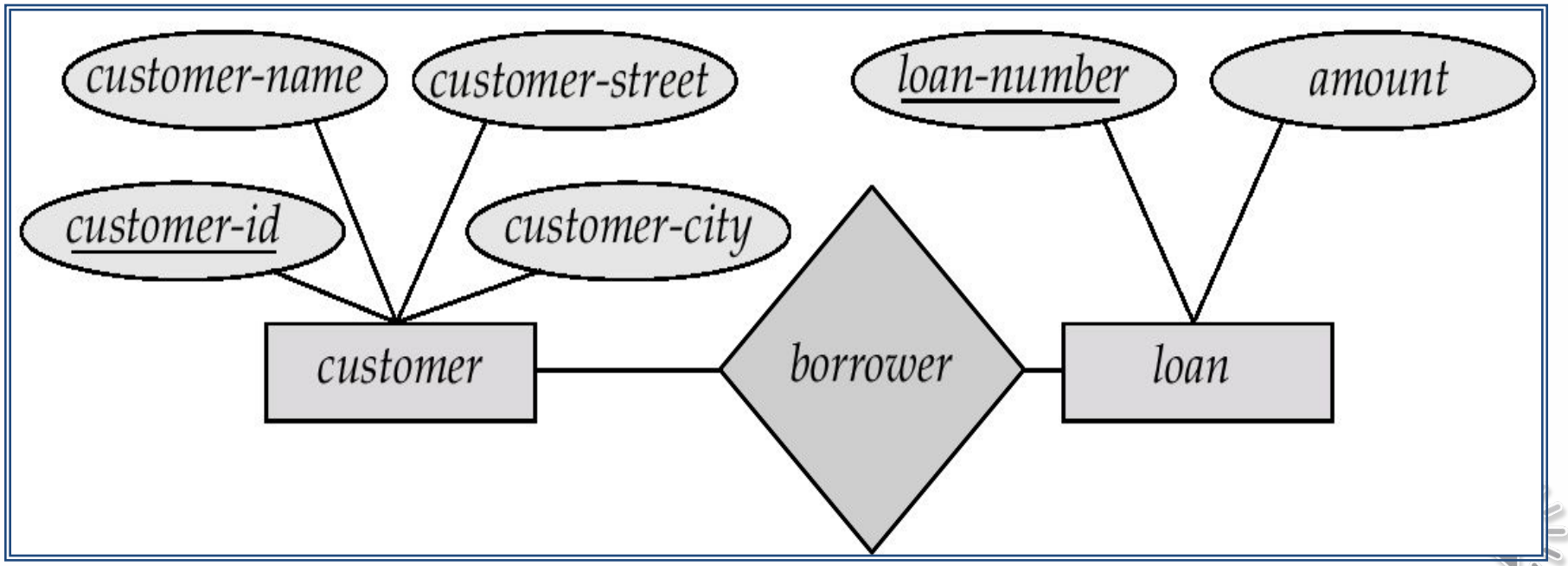
Many-To-One Relationships

In a many-to-one relationship a loan is associated with several (including 0) customers via *borrower*, a customer is associated with at most one loan via *borrower*

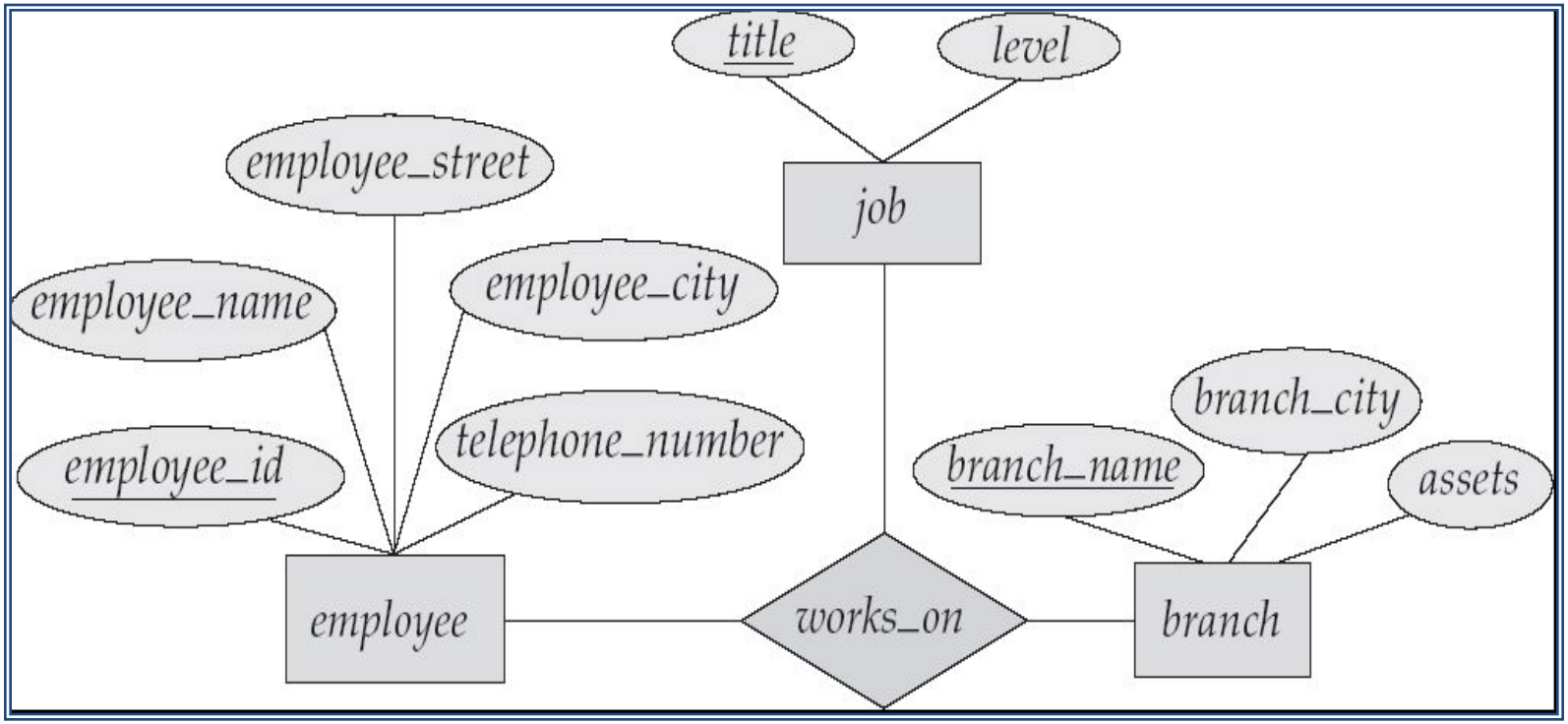


Many-To-Many Relationship

A customer is associated with several (possibly 0) loans via borrower
A loan is associated with several (possibly 0) customers via borrower



E-R Diagram with a Ternary Relationship



Parul®
University

DIGITAL LEARNING CONTENT

Cont...

- Extended E-R features
  - Specialization
  - Generalization
  - Aggregation

Specialization

Top-down design process; we designate subgroupings within an entity set that are distinctive from other entities in the set.

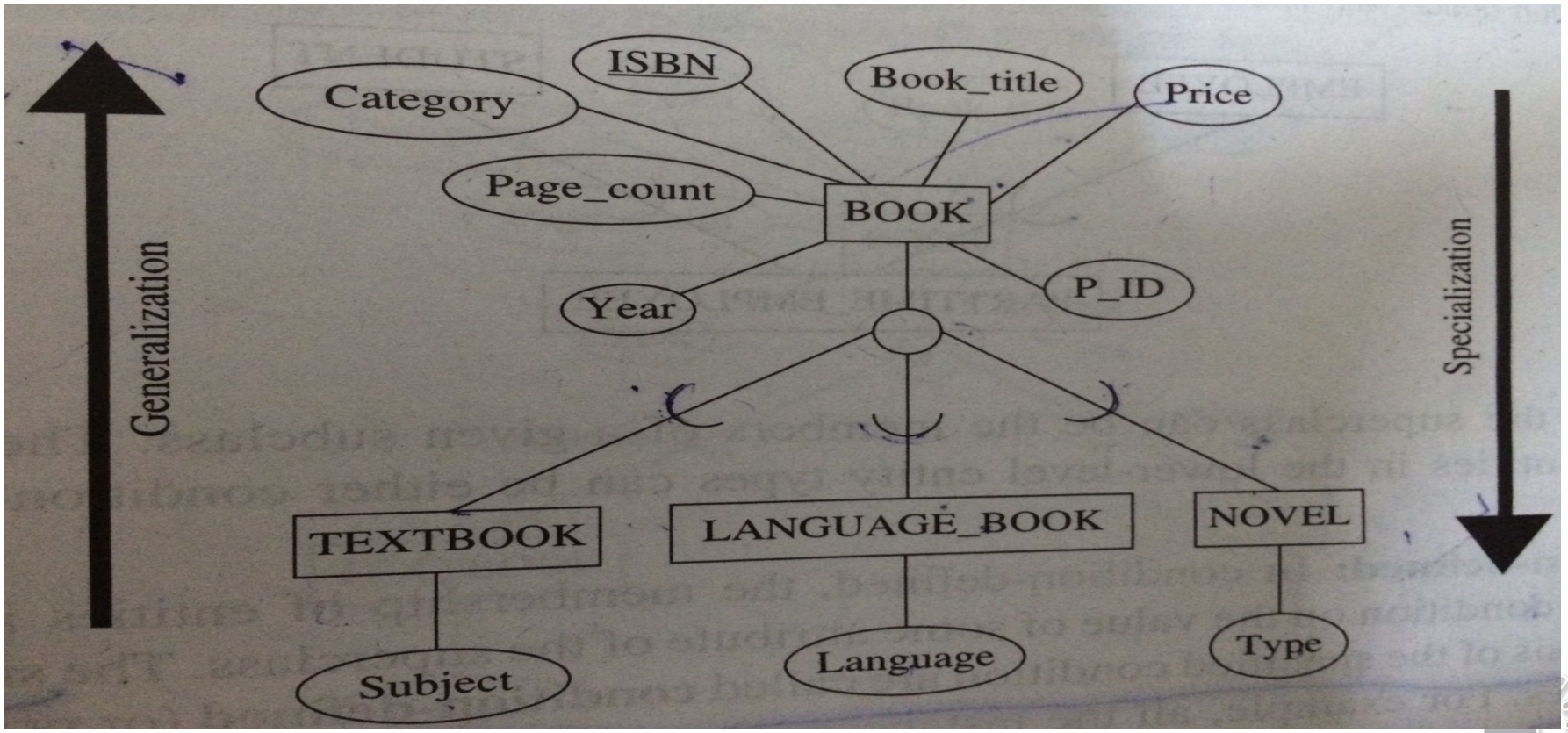
These subgroupings become lower-level entity sets that have attributes or participate in relationships that do not apply to the higher-level entity set.

Ex. Entity type BOOK can be further classified into three types. TEXTBOOK, LANGUAGE_BOOK and NOVEL.

Local or Specific Attribute- some additional set of that differentiate them from each other.

TEXTBOOK may have additional attribute Subject(computer , maths , science ctc

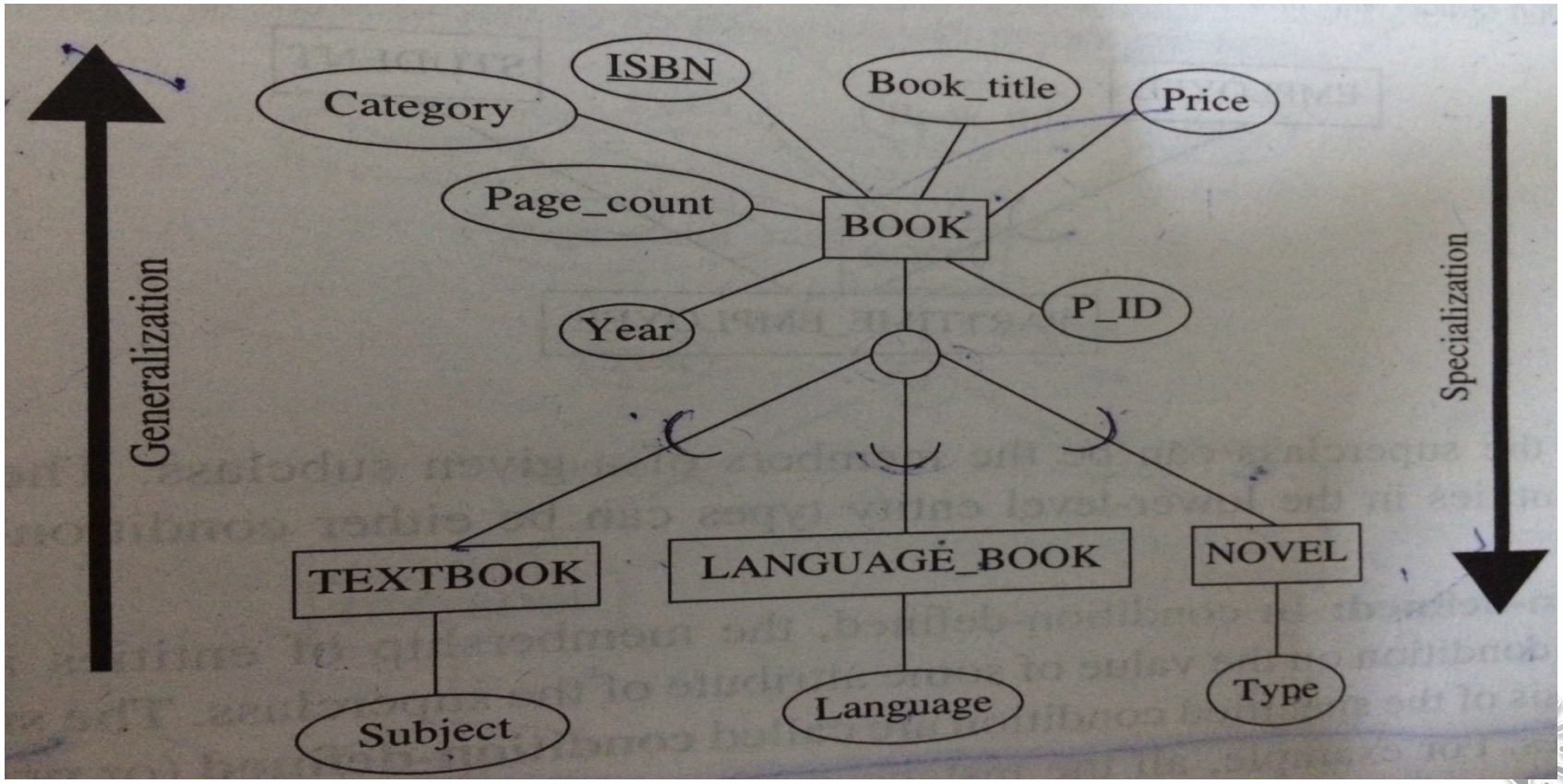
Cont....


ISBN
Category
Book_title
Price
Page_count
BOOK
Year
P_ID
Generalization
Specialization
TEXTBOOK
LANGUAGE_BOOK
NOVEL
Subject
Language
Type

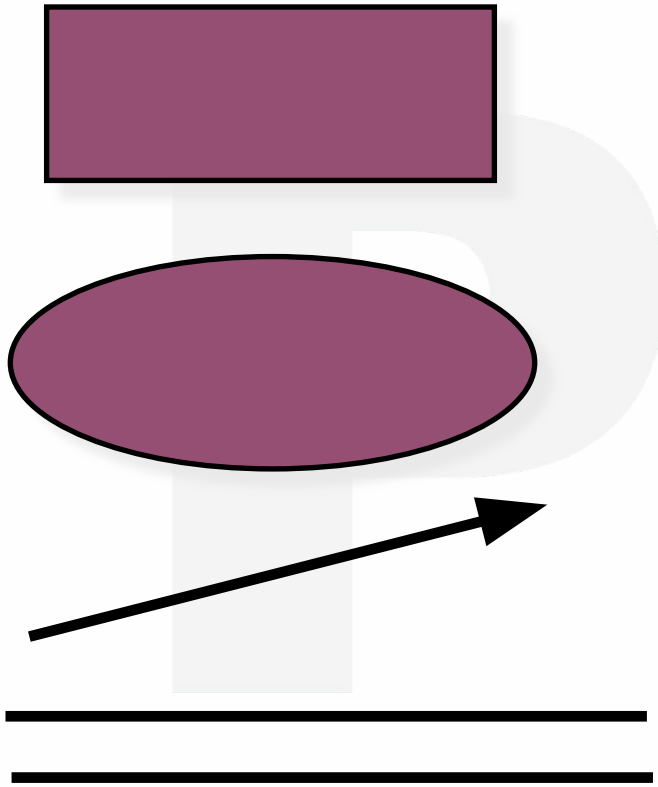
Generalization

- **A bottom-up design process** – combine a number of entity sets that share the same features into a higher-level entity set.
  - DBD may first identify entity type TEXTBOOK, LANG_BOOK, NOVEL then combine the common attribute to form BOOK.
- Specialization and generalization are simple inversions of each other; they are represented in an E-R diagram in the same way.
- The terms specialization and generalization are used interchangeably

Cont....


ISBN
Category
Book_title
Price
Page_count
BOOK
Year
P_ID
Generalization
Specialization
TEXTBOOK
LANGUAGE_BOOK
NOVEL
Subject
Language
Type

Flow Modeling Notation



external entity

process

data flow

data store

External Entity

A producer or consumer of data

Examples: a person, a device, a sensor

Another example: computer-based system

Data must always originate somewhere and must always be sent to something

Process

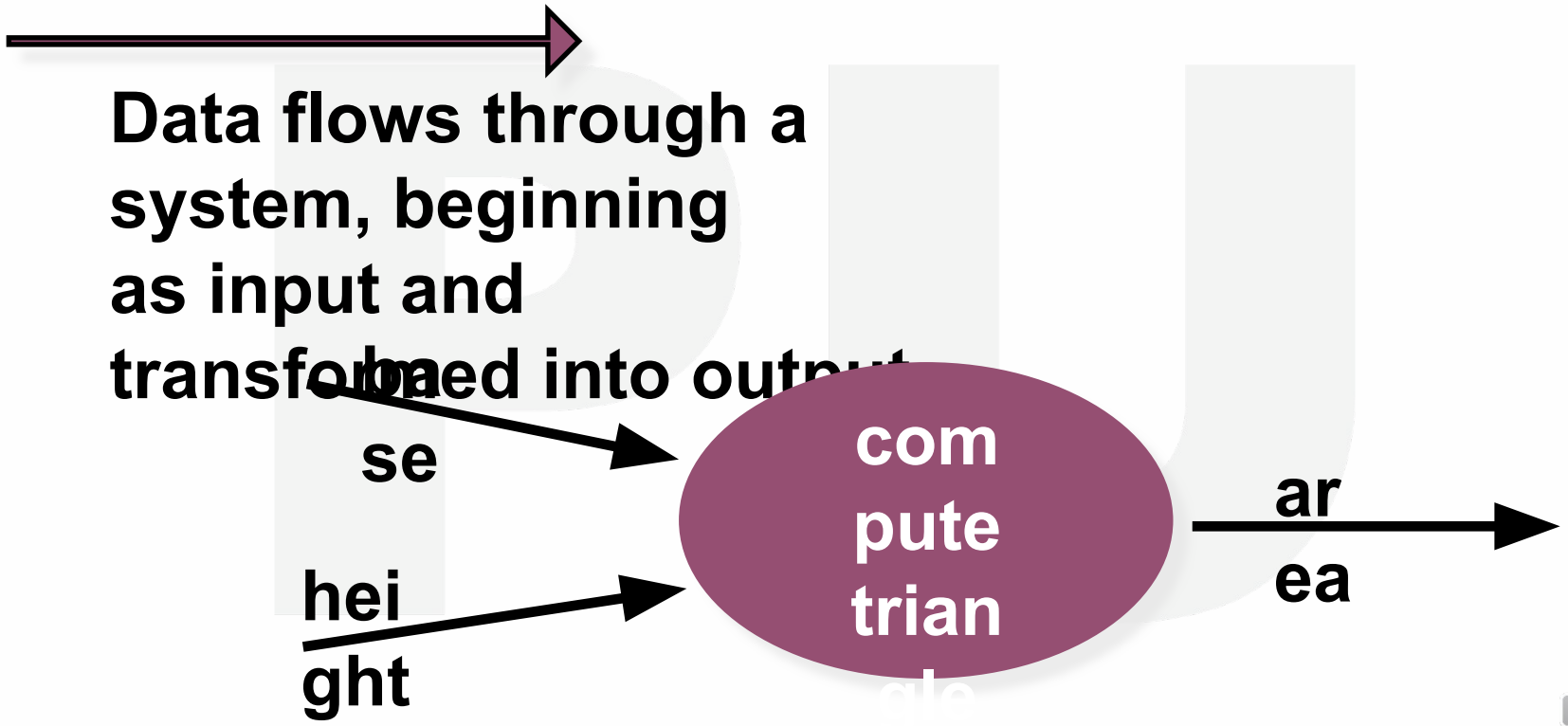
**A data transformer (changes input
to output)**

Examples: *compute taxes, determine area, format report, display graph*

Data must always be processed in some way to achieve system function

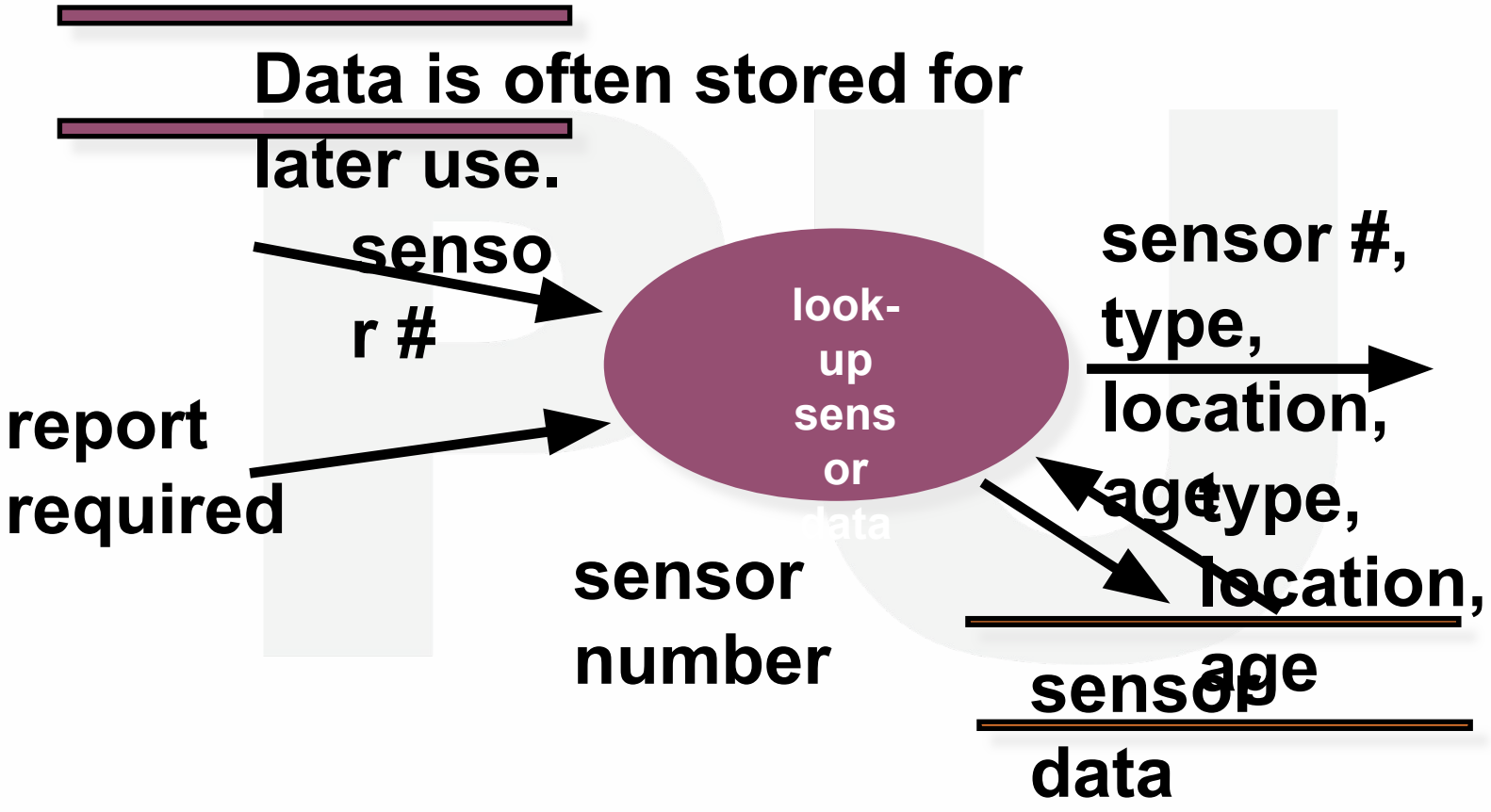
Data Flow

Data flows through a system, beginning as input and transformed into output.



Data Stores

Data is often stored for later use.



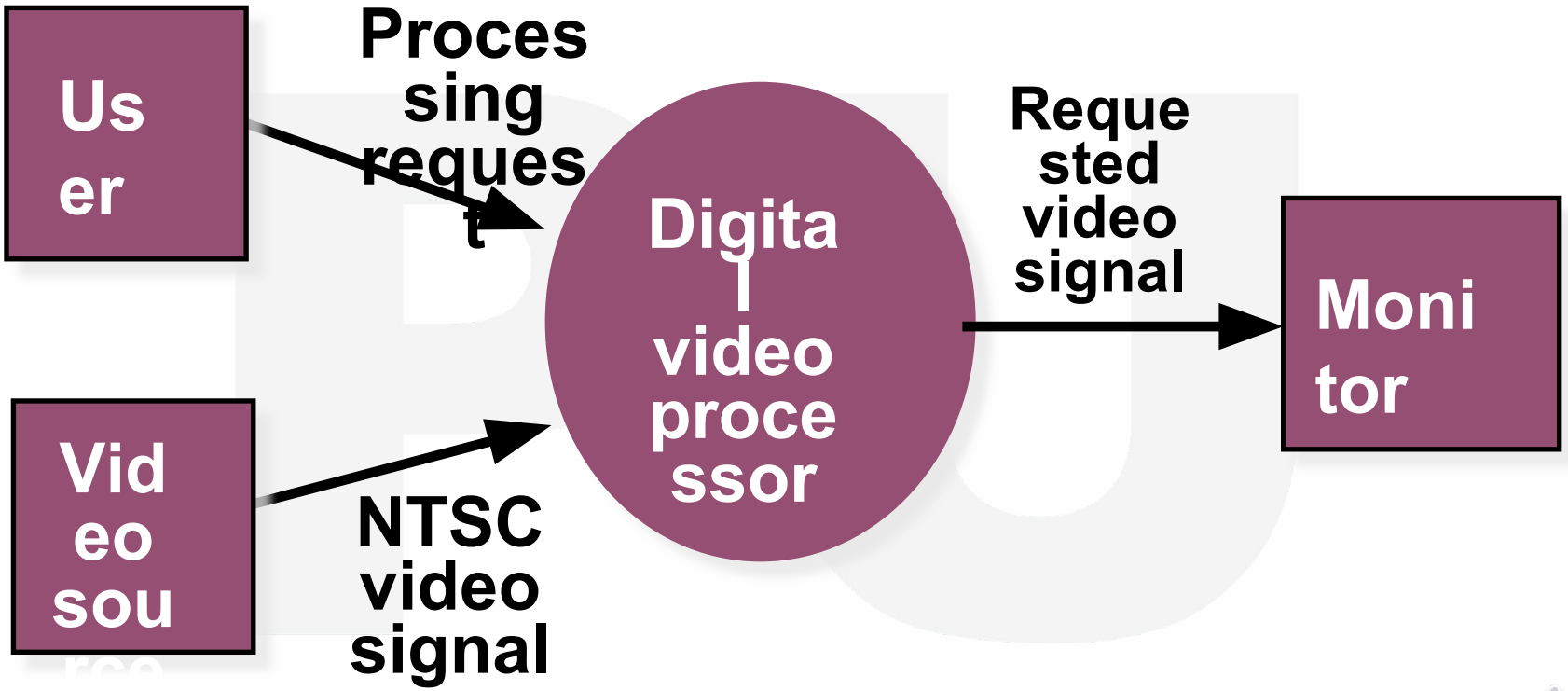
Data Flow Diagramming: Guidelines

- All icons must be labeled with meaningful names
- The DFD evolves through a number of levels of detail
- Always begin with a context level diagram (also called level 0)
- Always show external entities at level 0
- Always label data flow arrows
- Do not represent procedural logic

Constructing a DFD—I

- Review user scenarios and/or the data model to isolate data objects and use a grammatical parse to determine “operations”
- Determine external entities (producers and consumers of data)
- Create a level 0 DFD

Level 0 DFD Example



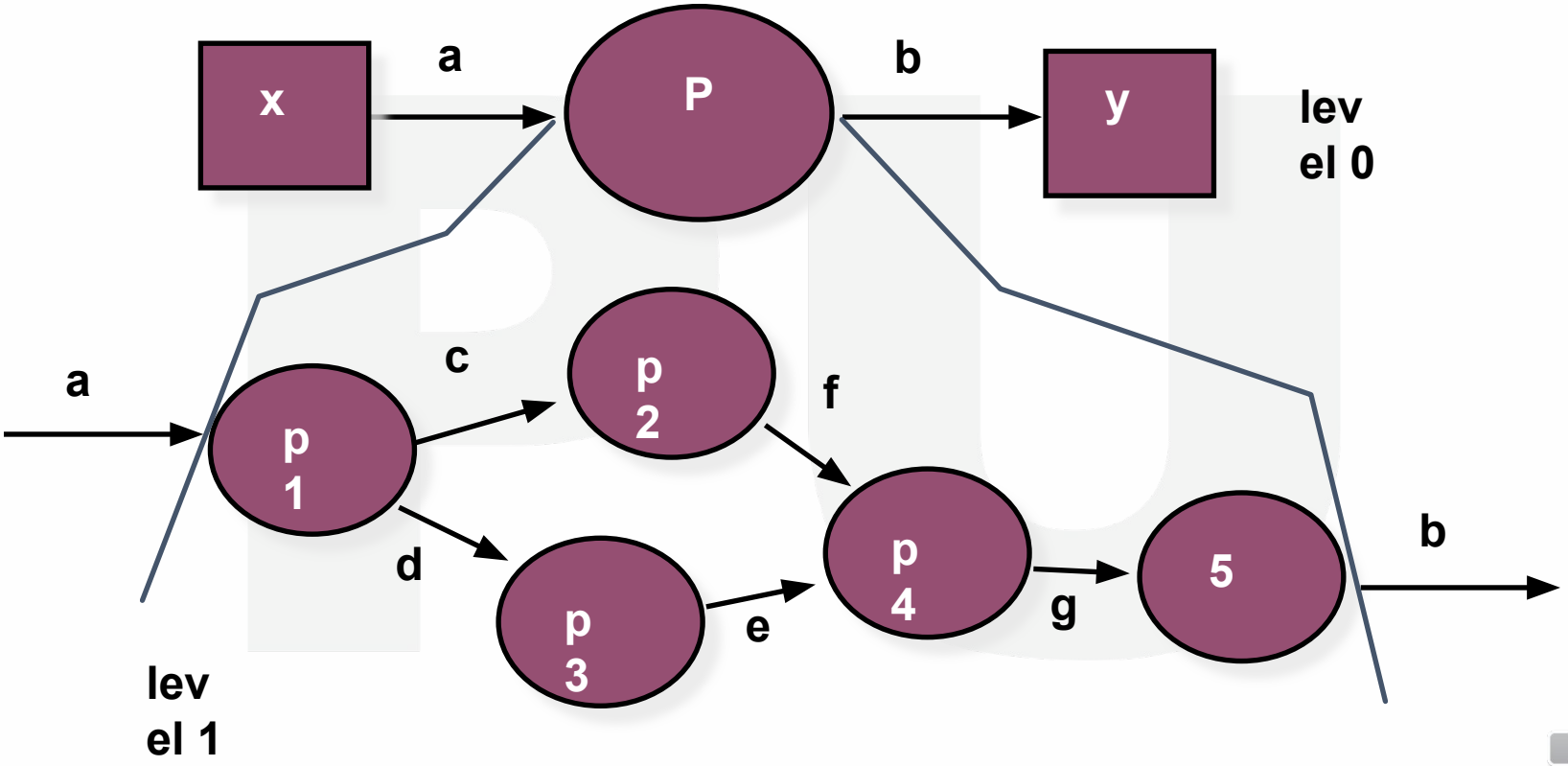
Parul® University

DIGITAL LEARNING CONTENT

Constructing a DFD—II

- Write a narrative describing the transform
- Parse to determine next level transforms
- “Balance” the flow to maintain data flow continuity
- Develop a level 1 DFD
- Use a 1:5 (approx.) Expansion ratio

The Data Flow Hierarchy



Parul®
University

DIGITAL LEARNING CONTENT

Data Dictionary

- The data dictionary is a reference work of data about data (that is, *metadata*), one that is compiled by systems analysts to guide them through analysis and design.
- Data flow diagrams are an excellent starting point for collecting data dictionary entries.

Data Dictionary (Cont.)

Data Dictionary may be used to also:

- Validate the data flow diagram for completeness and accuracy.
- Provide a stating point for developing screens and reports.
- Determine the contents of data stored in files.
- Develop the logic for data-flow diagram processes.

Data Dictionary Example

- Table name: scheme1_master

  Primary key: schemeid

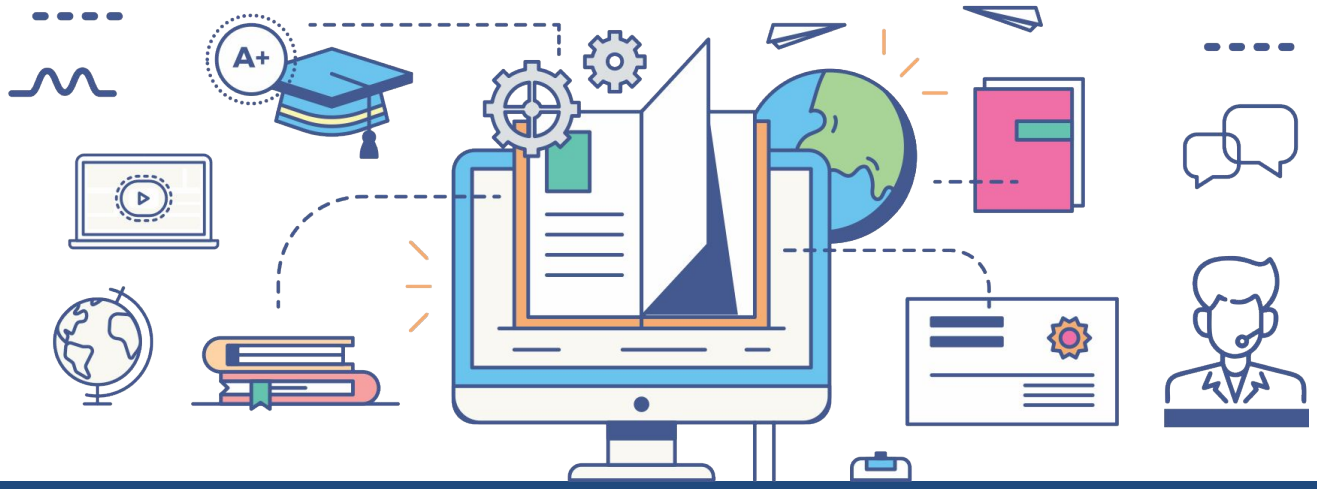
  Foreign key: null

- References:

Data Dictionary Example

FIELDNAME	DATATYPE	SIZE	CONSTRAINT	DESCRIPTION
Schemeid	Varchar2	20	Primary key	Stores the id of scheme
Time	Number	10		Stores the total time provided by scheme (in hours)
Days	Number	10		Stores the total days for expiring the scheme
Rupees	Number	4		Stores the amount for the schemes

DIGITAL LEARNING CONTENT



Parul® University

www.paruluniversity.ac.i