

Requirement Analysis

- Sanghamitra De

Objectives of requirements

- To achieve agreement on the requirements
- To provide a basis for software design
- To provide a reference point for software validation

Steps in requirements definition

- Requirements identification
- Identification of software development constraints
- Requirements analysis
- Requirements representation
- Requirements communication
- Preparation for validation of software

Software Requirements Process

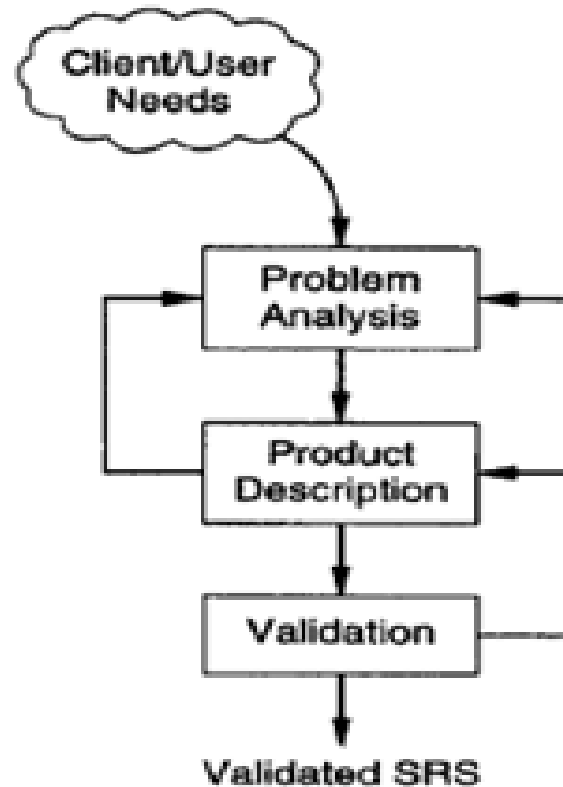


Fig The requirement process.

Problem Analysis

- Informal Approach
- Structured Analysis

Problem Analysis

➤ Informal Approach

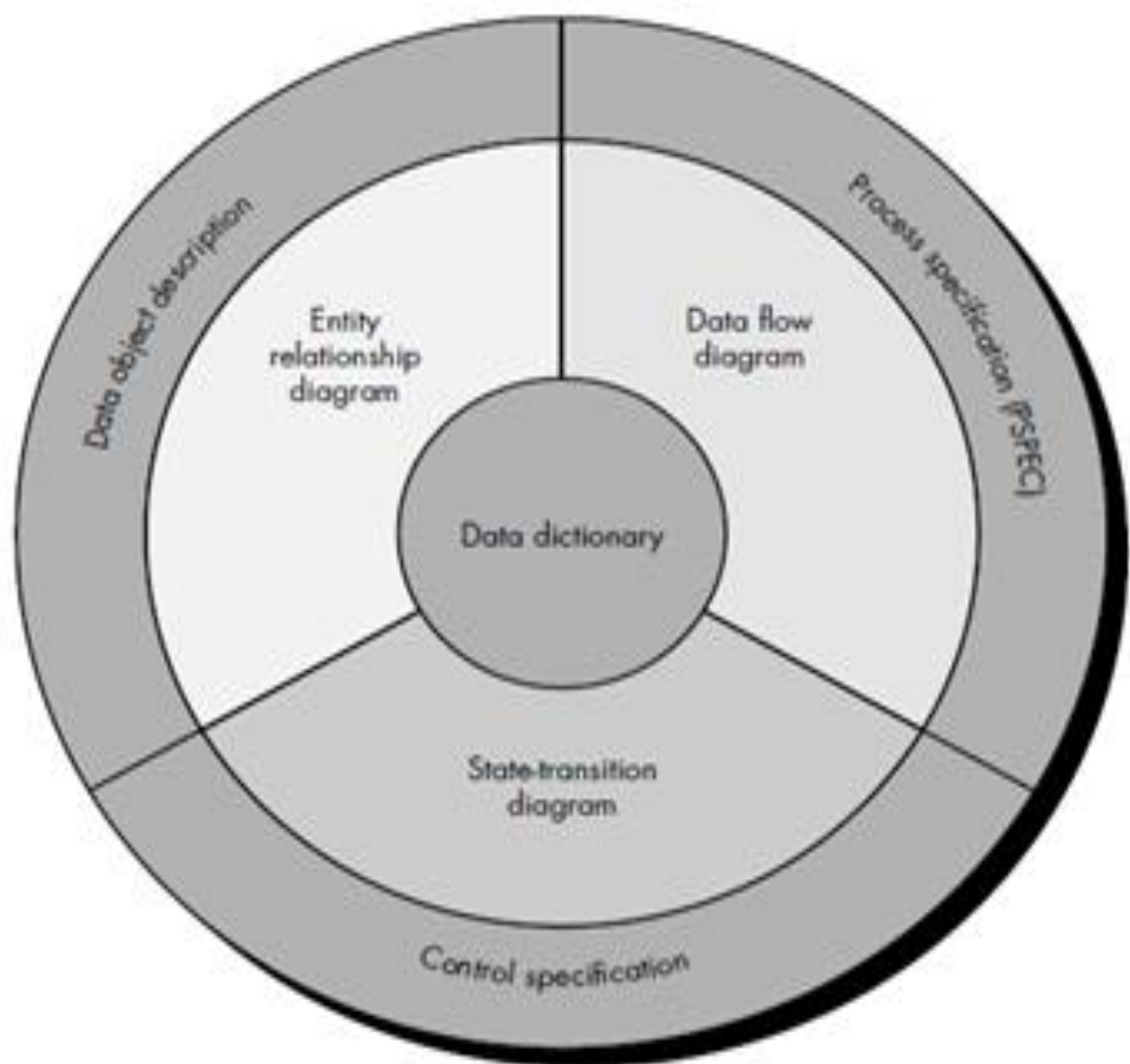
- ✓ Analyst will have a series of meetings with the clients and end users
- ✓ In the early meetings, the clients and end users will explain to the analyst about their work, their environment, and their needs as they perceive them.
- ✓ Any documents describing the work or the organization may be given, along with outputs of the existing methods of performing the tasks.
- ✓ Next few meetings are used to seek clarifications of the parts analyst does not understand
- ✓ An initial draft of the SRS may be used in the final meetings.

Problem Analysis

➤ Structured Analysis

The analysis model must achieve three primary objectives:

- a) to describe what the customer requires
- b) to establish a basis for the creation of a software design
- c) to define a set of requirements that can be validated once the software is built



FIGURE

The structure of
the analysis
model

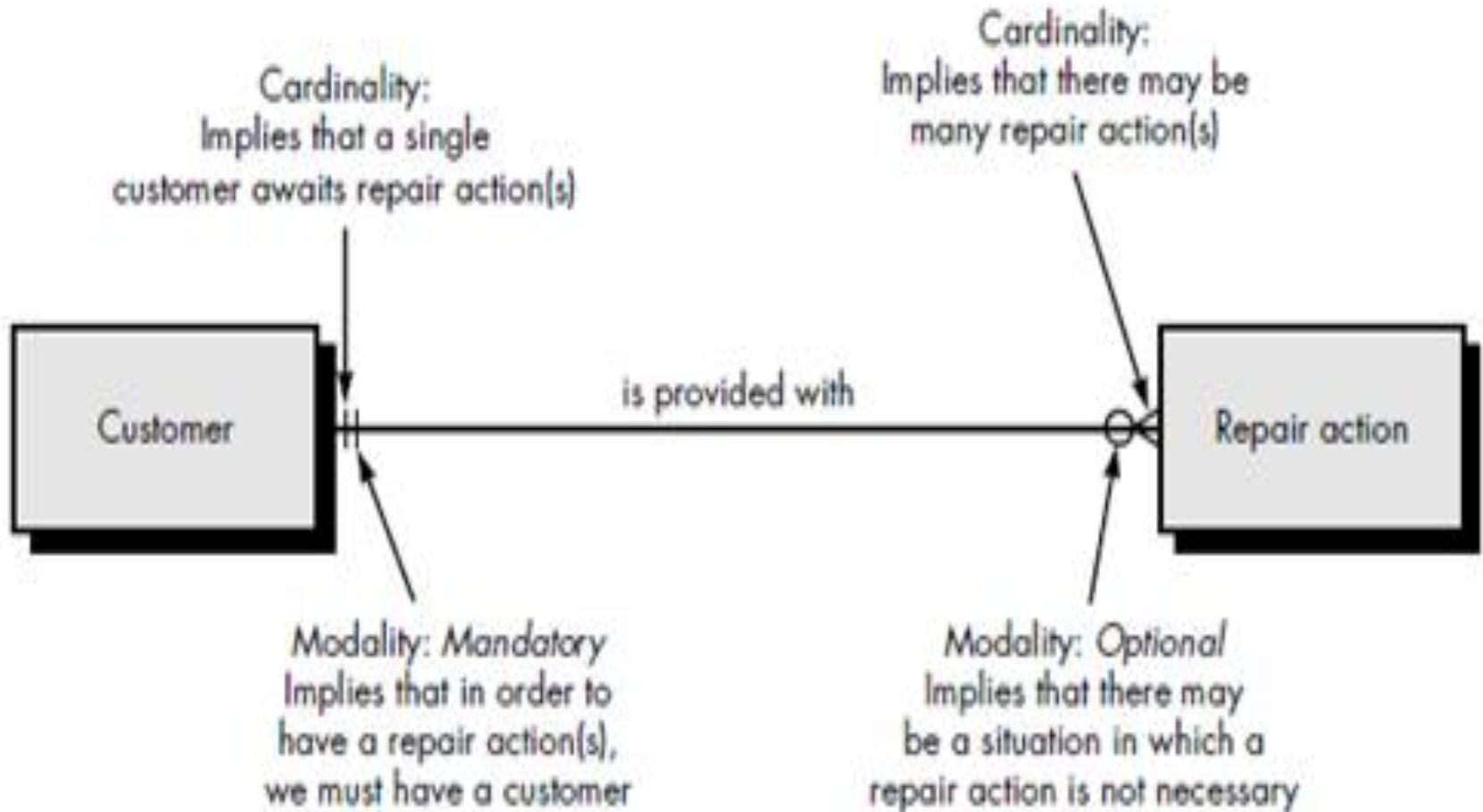
ER Diagrams

- enables a software engineer to identify data objects and their relationships using a graphical notation
- consists of three interrelated pieces of information: the data *object*, the *attributes* that describe the data object, and the *relationships* that connect data objects to one another

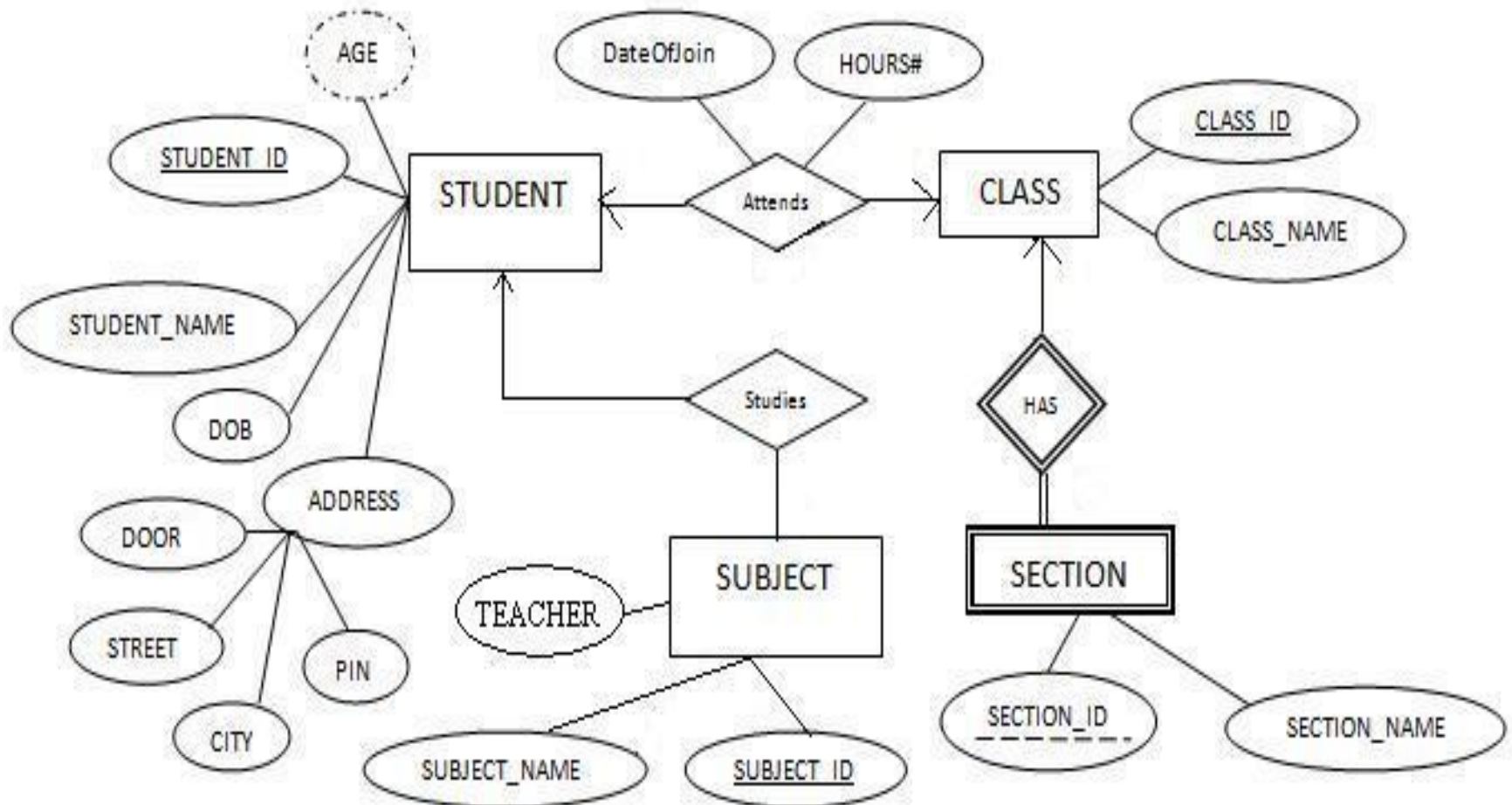
Cardinality & Modality

- **Cardinality** is the specification of the number of occurrences of one [object] that can be related to the number of occurrences of another [object].
 - ✓ *One-to-one* ($1:1$)
 - ✓ *One-to-many* ($1:N$)
 - ✓ *Many-to-many* ($M:N$)
- The **modality** of a relationship is 0 if there is no explicit need for the relationship to occur or the relationship is optional. The modality is 1 if an occurrence of the relationship is mandatory.

ER Diagrams



More ER Diagrams



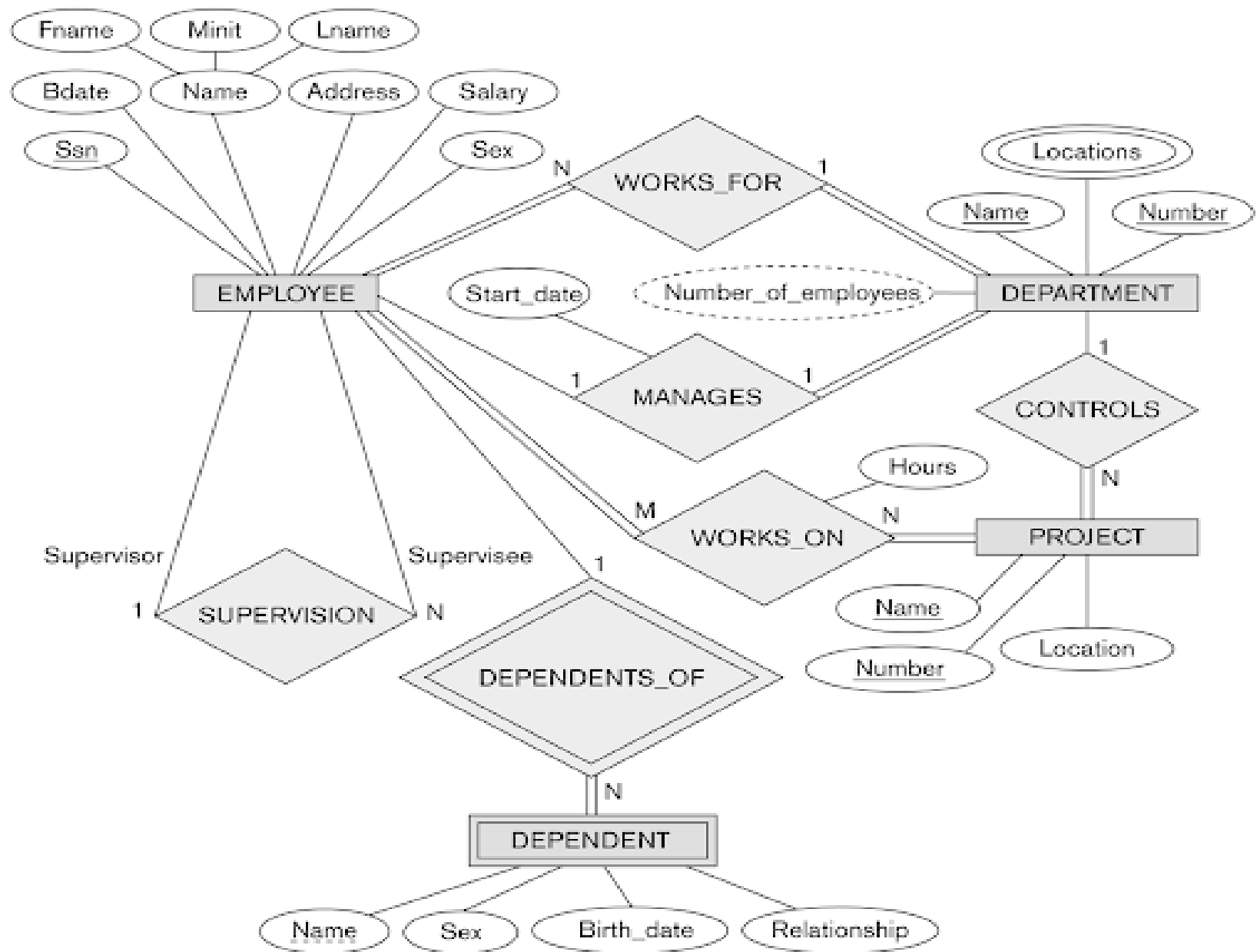


Figure 3.2

An ER schema diagram for the COMPANY database. The diagrammatic notation is introduced gradually throughout this chapter.

Data Flow Modeling

- Uses function-based decomposition while modeling the problem
- Uses Data Flow Diagram & Data Dictionary

Data flow diagrams (DFDs)

- Categorized as either logical or physical.
- Logical DFD focuses on the business and how the business operates. It describes the business events that take place and the data required and produced by each event.
- Physical DFD shows how the system will be implemented as mentioned before.

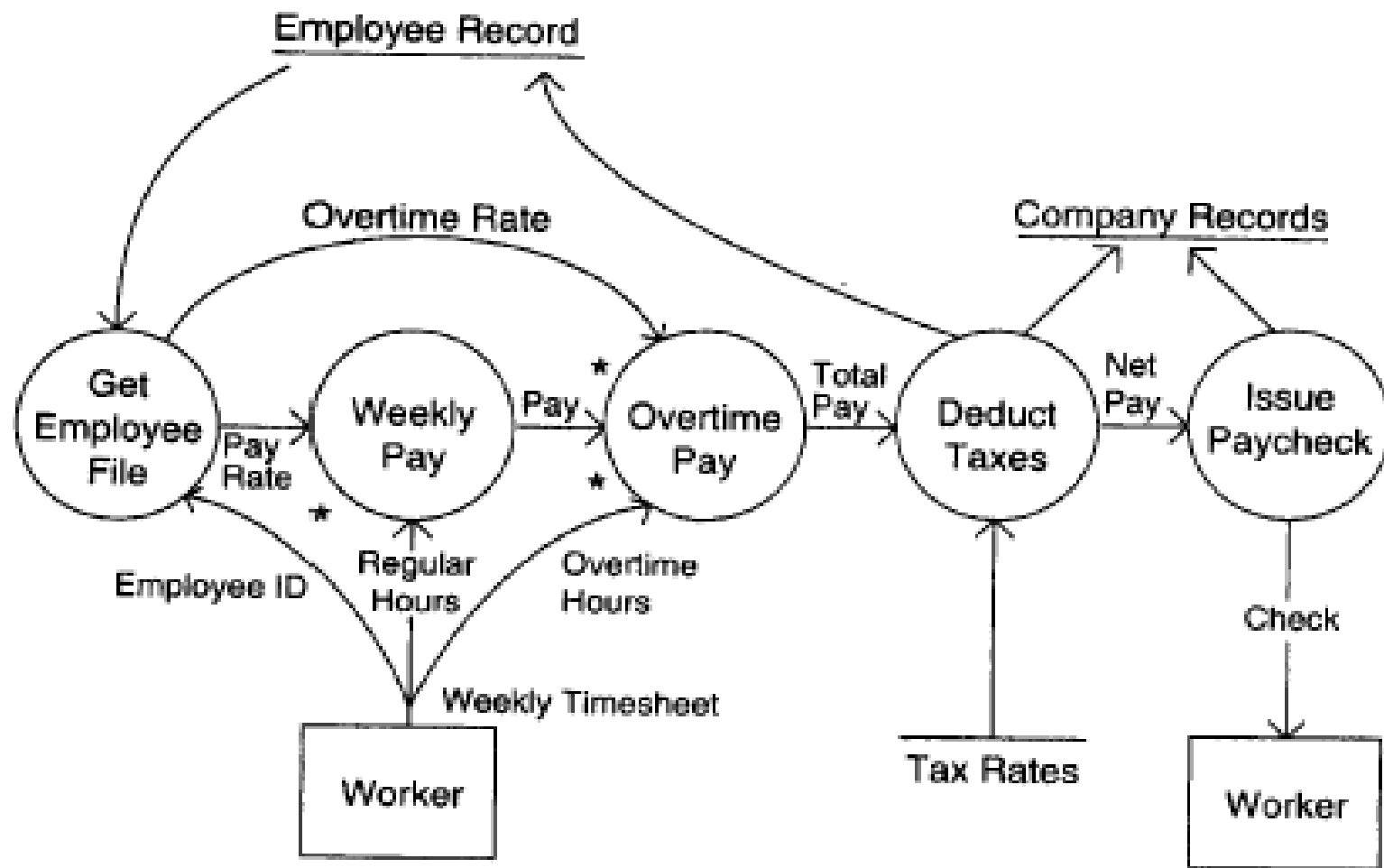


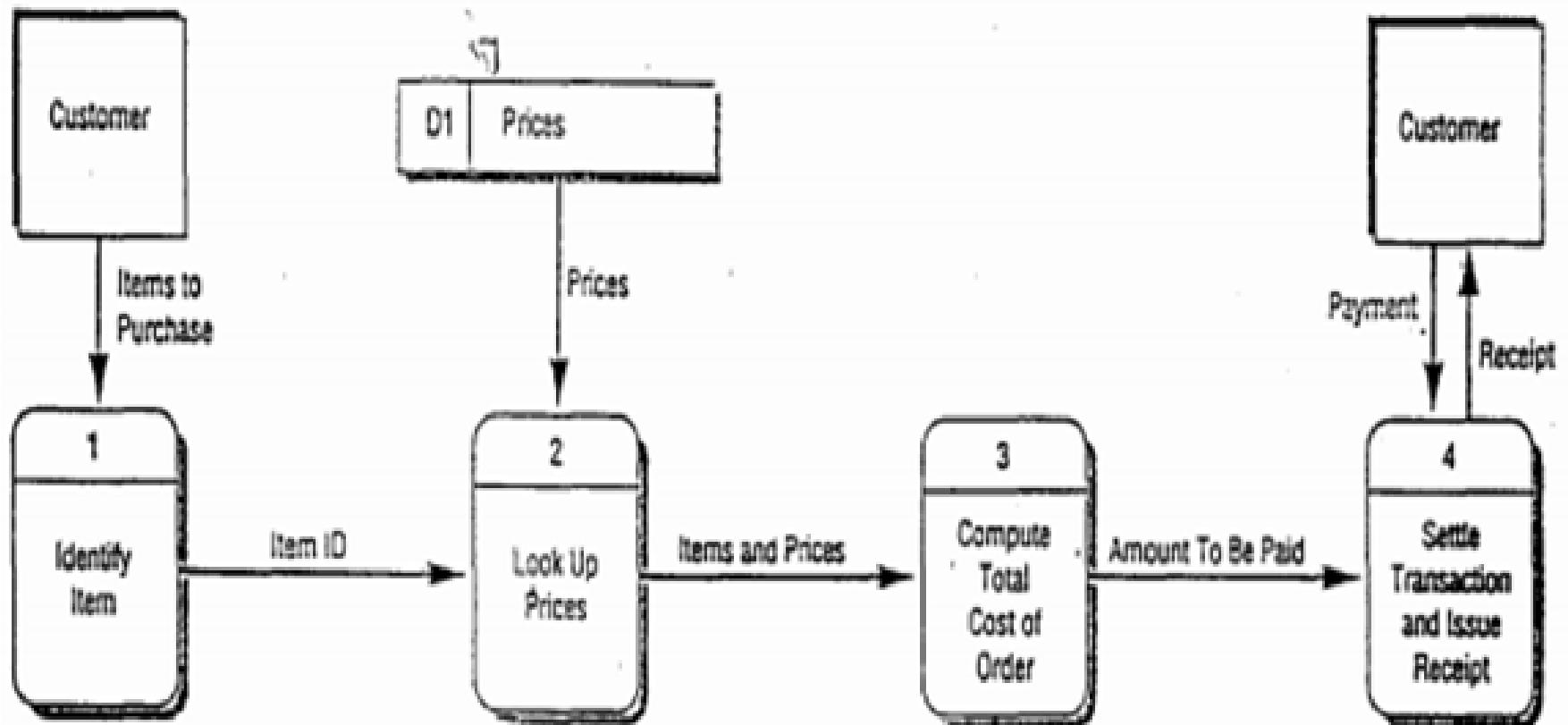
Figure . DFD of a system that pays workers.

Design Feature	Logical	Physical
What the model depicts	How the business operates	How the system will be implemented (or how the current system operates)
What the processes represent	Business activities	Programs, program modules and manual procedures
What the data stores represent	Collections of data, regardless of how the data is stored	Physical files and databases, manual files
Type of data stores	Show data stores representing permanent data collections	Master files, transaction files. Any processes that operate at two different times must be connected by a data store
System controls	Show business controls	Show controls for validating input data, for obtaining a record (record found status), for ensuring successful completion of a process and for system security (example: journal records)

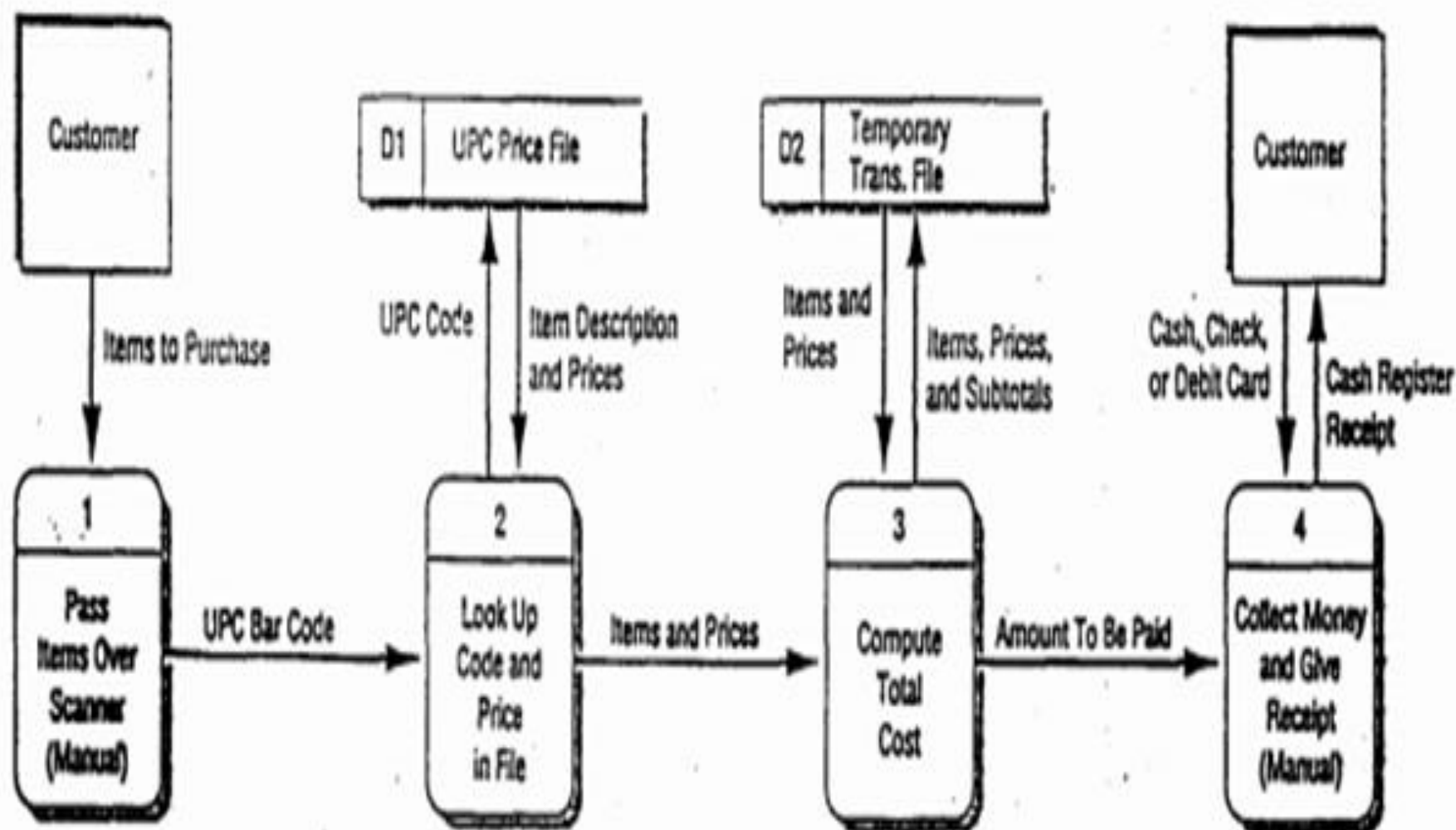
Figure : Features common of logical and physical data flow diagrams.

Example: grocery store cashier

Logical Data Flow Diagram



Physical Data Flow Diagram



Data Dictionary

- Data dictionary associated with a DFD states precisely the structure of each data flow in the DFD.
- Components in the structure of a data flow may also be specified in the data dictionary, as well as the structure of files shown in the DFD.
- To define the data structure, different notations are used:
 - ✓ Sequence or composition (represented by '+')
 - ✓ Selection (represented by vertical bar '|') means one OR the other
 - Repetition (represented by '*') means one or more occurrences.

```

weekly_timesheet =
    Employee_name +
    Employee_Id +
    [Regular_hours + Overtime_hours] *

pay_rate =
    [Hourly | daily | weekly] +
    Dollar_amount

Employee_name =
    Last + First + Middle_initial

Employee_Id =
    digit + digit + digit + digit

```

Figure 4.1: Data dictionary.

Structured Analysis Method

- Each system can be viewed as a transformation function operating within an environment that takes some inputs from the environment and produces some outputs for the environment.
- Overall transformation function of the entire system should be partitioned into sub functions that together form the overall function.
- Subfunctions can be further partitioned and the process repeated until we reach a stage where each function can be comprehended easily.
- In any complex system the data transformation from input to output will occur in a series of transformations starting from the input and culminating in the desired output.
- By tracking as the data flows through the system, the various functions being performed by a system can be identified.
- This approach can be modeled easily by data flow diagrams.
- For drawing a DFD, a top-down approach is suggested in the structured analysis method.
- This results in a leveled set of DFDs

Object-Oriented Modeling

- Objects interact with each other through the services they provide.
- Some objects also interact with the users through their services such that the users get the desired services.
- Goal of modeling:
 - ✓ identify the object classes that exist in the problem domain
 - ✓ define the classes by specifying what state information they encapsulate and what services they provide
 - ✓ identify relationships that exist between objects of different classes
- *Generalization-specialization* structure can be used by a class to inherit all or some attributes and services of a general class and add more attributes and services. This structure is modeled through inheritance.

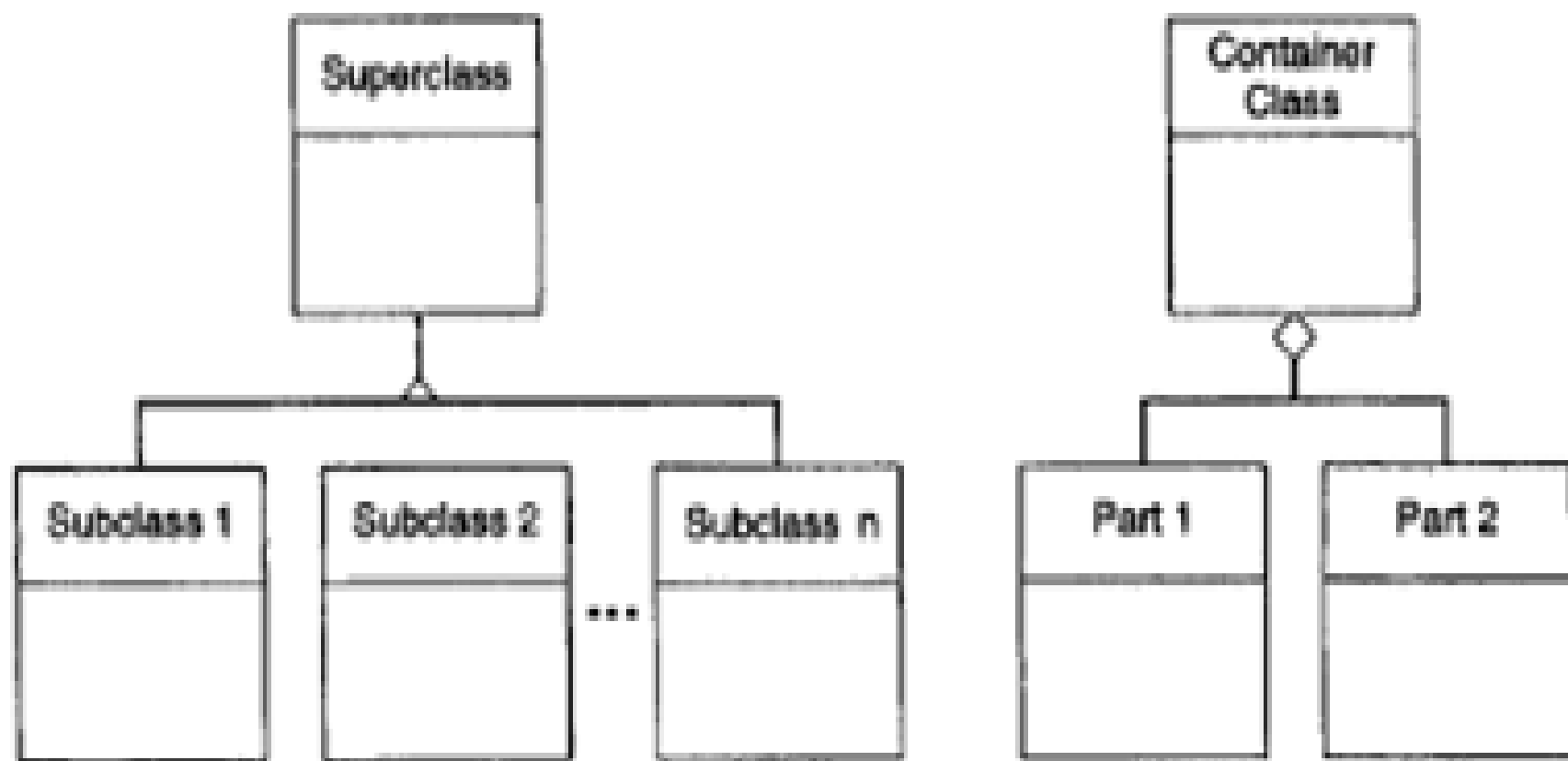


Figure : Class structures.

Object-Oriented Modeling

- *Aggregation* structure models the whole-part relationship. An object may be composed of many objects; this is modeled through the aggregation structure.
- Relationship between objects also has to be captured if a system is to be modeled properly. This is captured through *associations*.
- An association is shown in the class diagram by having a line between the two classes.
- The multiplicity of an association specifies how many instances of one class may relate to instances of the other class through this association.
- Multiplicity is specified by having a star (*) on the line adjacent to the class representing zero or more instances of the class may be related to an instance of the other class.

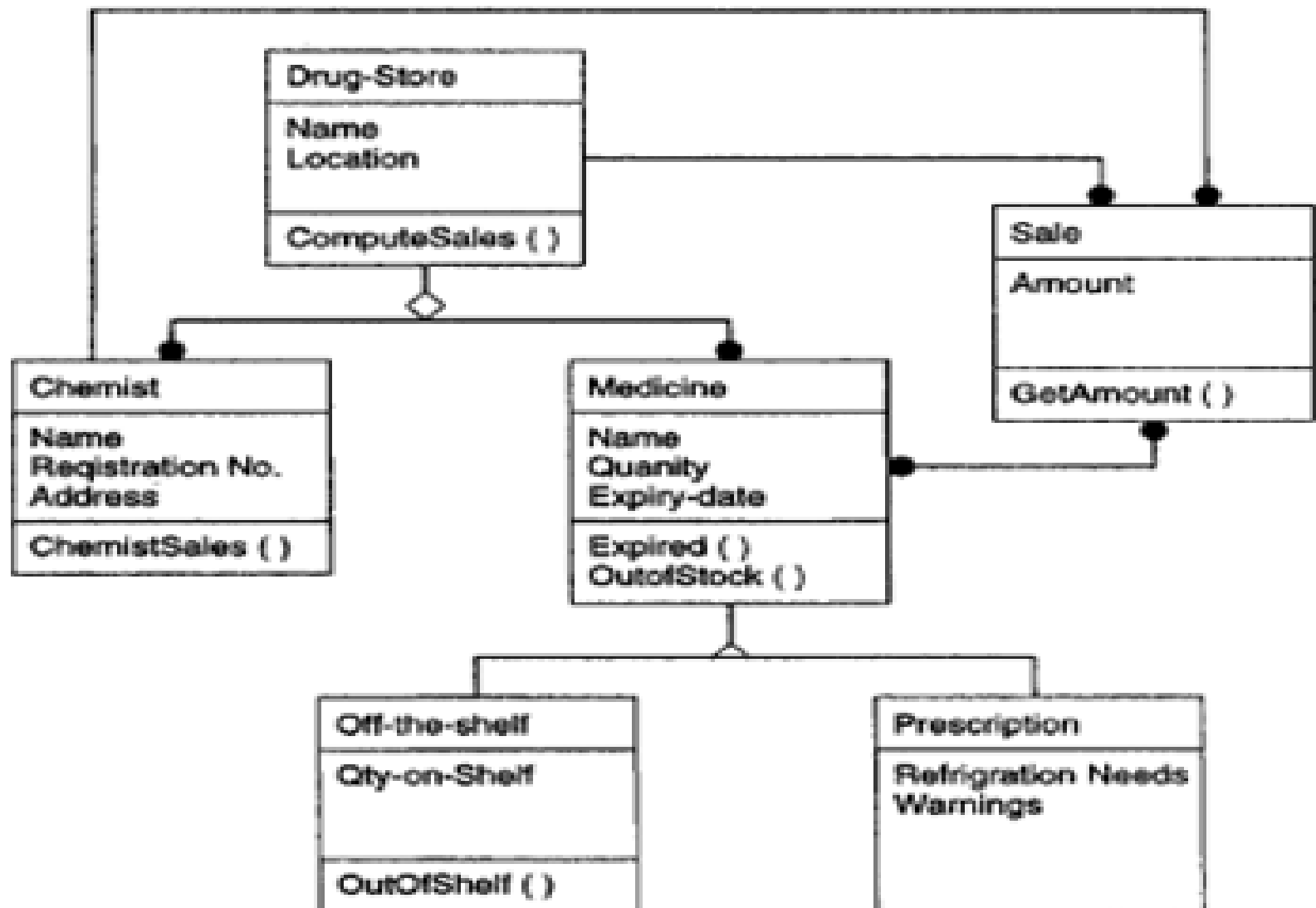


Figure : Model of a drugstore.

Performing Analysis

- Steps to performing analysis:
 - ✓ Identifying objects and classes
 - ✓ Identifying structures
 - ✓ Identifying attributes
 - ✓ Identifying associations
 - ✓ Defining services

Performing Analysis

➤ *Identifying Objects and Classes*

- ✓ To identify analysis objects, start by looking at the problem space and its description.
- ✓ In the summary and descriptions of the problem space, consider the nouns.
- ✓ Frequently, nouns represent entities in the problem space which will be modeled as objects.

Performing Analysis

➤ *Identifying Structure*

- ✓ Structures represent the hierarchies that exist between object classes.
- ✓ In object oriented modeling, the hierarchies are defined between classes that capture generalization-specialization and whole-part relationships.
- ✓ To identify the classification structure, consider the classes that have been identified as a generalization and see if there are other classes that can be considered as specializations of this.
- ✓ Similarly, consider classes as specializations and see if there are other classes that have similar attributes. If so, see if a generalized class can be identified of which these are specializations.
- ✓ To identify assembly structure, consider each object of a class as an assembly and identify its parts or components.

Performing Analysis

➤ *Identifying Attributes*

- ✓ Attributes add detail about the class and are the repositories of data for an object.
- ✓ Which attributes should be used to define the class of an object depends on the problem and what needs to be done
- ✓ To identify attributes, consider each class and see which attributes are needed by the problem domain.
- ✓ Then position each attribute properly using the structures; if the attribute is a common attribute, it should be placed in the superclass, while if it is specific to a specialized object it should be placed with the subclass.

Performing Analysis

➤ *Identifying Associations*

- ✓ Associations capture the relationship between instances of various classes.
- ✓ The associations between objects are derived from the problem domain directly once the objects have been identified.
- ✓ An association may have attributes of its own; these are typically attributes that do not naturally belong to either object.

Performing Analysis

➤ *Defining Services*

- ✓ An object performs a set of predefined services.
- ✓ A service is performed when the object receives a message for it.
- ✓ Services really provide the active element in object-oriented modeling; they are the agent of state change or "processing."
- ✓ It is through the services that desired functional services can be provided by a system.
- ✓ A method for identifying services is to define the system states and then in each state list the external events and required responses.
- ✓ For each of these, identify what services the different classes should possess.

Prototyping

- In prototyping, a partial system is constructed, which is then used by the client, users, and developers to gain a better understanding of the problem and the needs.
- A software prototype can be defined as a partial implementation of a system whose purpose is to learn something about the problem being solved or the solution approach.
- The rationale behind using prototyping for problem understanding and analysis is that the client and the users often find it difficult to visualize how the eventual software system will work in their environment just by reading a specification document.

Prototyping

- Prototypes are of two types:
 - ✓ In throwaway approach the prototype is constructed with the idea that it will be discarded after the analysis is complete and the final system will be built from scratch.
 - ✓ In the evolutionary approach, the prototype is built with the idea that it will eventually be converted into the final system.
- From the point of view of problem analysis and understanding, the throwaway prototypes are more suited.

Prototyping

➤ In a throwaway prototype

- ✓ Whether or not to prototype.
- ✓ Requirements of a system can be divided into three sets - those that are **well understood**, those that are **poorly understood**, and those that are **not known**
- ✓ In a throwaway prototype, the poorly understood requirements are the ones that should be incorporated.
- ✓ Based on the experience with the prototype, these requirements then become well understood.
- ✓ Divide the set of poorly understood requirements further into two sets—those **critical** to design, and those **not critical** to design.
- ✓ Requirements that can be easily incorporated in the system later are considered noncritical to design. If which of the poorly understood requirements are critical and which are noncritical can be determined, then the throwaway prototype should focus mostly on the critical requirements.

Prototyping

- If the set of poorly understood requirements is substantial (in particular the subset of critical requirements), then a throwaway prototype should be built.
- The development activity starts with an SRS for the prototype.
- Developing the SRS for the prototype requires identifying the functions that should be included in the prototype.
- Those requirements that tend to be unclear and vague, or where the clients and users are unsure or keep changing their mind, are the ones that should be implemented in the prototype.
- Based on what aspects of the system are included in the prototype, the prototyping can be considered **vertical** or **horizontal**.
- In horizontal prototyping the system is viewed as being organized as a series of layers and some layer is the focus of prototyping. E.g., the user interface layer is frequently a good candidate for such prototyping, where most of the user interface is included in the prototype.

Prototyping

- In vertical prototyping, a chosen part of the system, which is not well understood, is built completely.
- This approach is used to validate some functionality or capability of the system.
- Focus during prototyping is to keep costs low and minimize the prototype production time.
 - ✓ Many of the bookkeeping, documenting, and quality control activities are kept to a minimum during prototyping.
 - ✓ Efficiency concerns are less
 - ✓ Exception handling, recovery, conformance to some standards and formats are typically not included
 - ✓ Design documents, a test plan, and a test case specification are not needed
 - ✓ Reduced testing

Prototyping

- Benefits obtained due to the use of prototype
 - ✓ reduced requirement errors
 - ✓ reduced volume of requirement change requests
 - ✓ reduced cost of development itself.

SRS: characteristics

A good SRS is:

- Correct
- Complete
- Unambiguous
- Verifiable
- Consistent
- Ranked for importance and/or stability
- Modifiable
- Traceable

Components of an SRS

- Functionality or Functional Requirements
- Performance or Performance Requirements: *static and dynamic*.
- Design constraints imposed on an implementation
 - ✓ *Standards Compliance*
 - ✓ *Hardware Limitations*
 - ✓ *Reliability and Fault Tolerance*
 - ✓ *Security*
- External interfaces or External Interface Requirements

Specification Language

- The language should support the desired qualities of the SRS:
 - ✓ Modifiability
 - ✓ Understandability
 - ✓ Unambiguous
- Language should be easy to learn and use
- Formal notations exist for specifying specific properties of the system, natural languages are now most often used for specifying requirements.
- Formal languages used to specify particular properties or for specific parts of the system
- Formal specifications are generally contained in the overall SRS, which is in a natural language.

Specification Language

- Drawbacks of natural languages: imprecise and ambiguous.
- To overcome drawbacks natural language is used in a structured fashion. In structured English:
 - ✓ requirements are broken into sections and paragraphs
 - ✓ each paragraph is then broken into subparagraphs
 - ✓ many organizations also specify strict uses of some words like "shall," "perhaps," and "should" and try to restrict the use of common phrases in order to improve the precision and reduce the verbosity and ambiguity.
- General rule when using a natural language is to be precise, factual, and brief, and organize the requirements hierarchically where possible, giving unique numbers to each separate requirement
- Decision tables & regular expressions are used

Structure of a Software Requirements Specification (SRS)

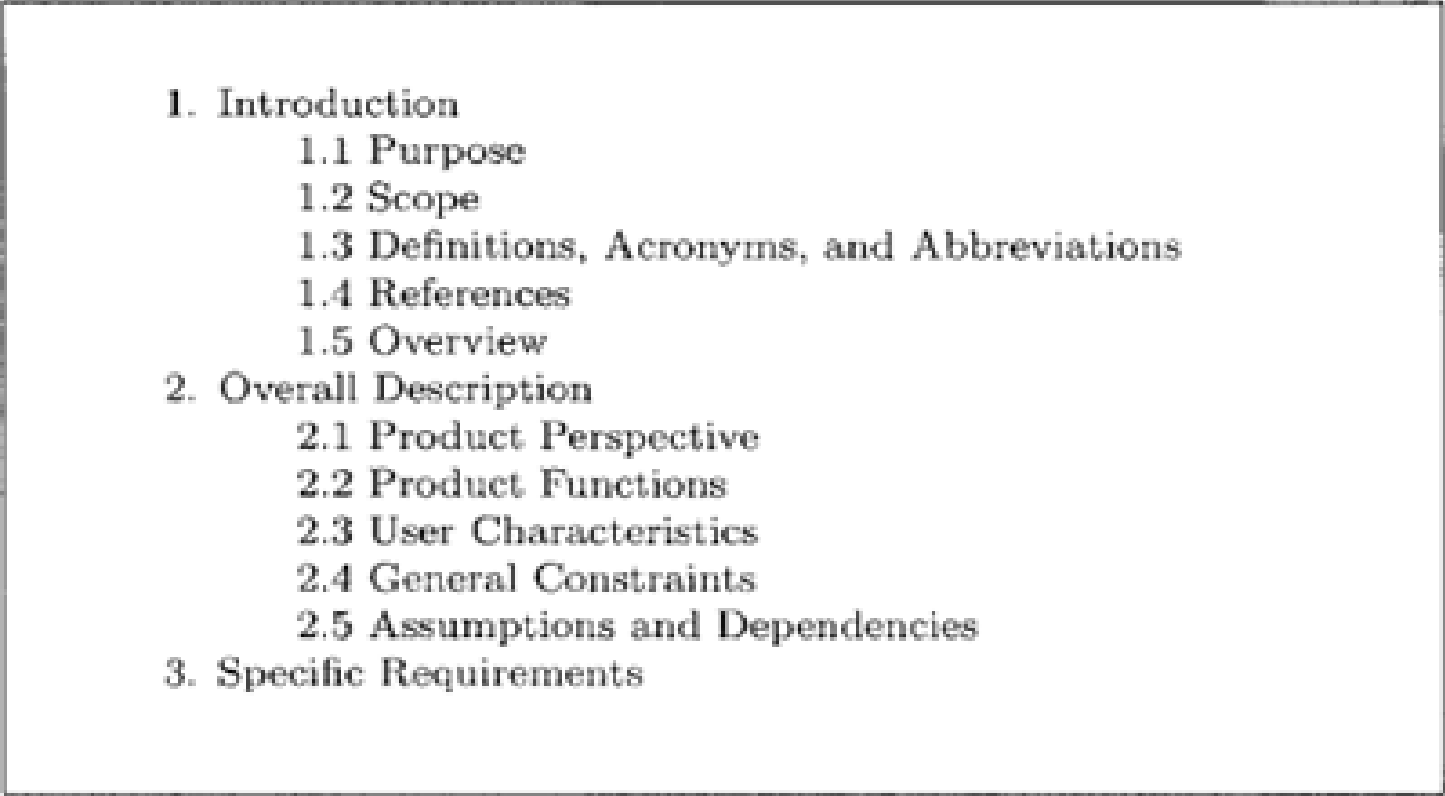
- 
- 1. Introduction
 - 1.1 Purpose
 - 1.2 Scope
 - 1.3 Definitions, Acronyms, and Abbreviations
 - 1.4 References
 - 1.5 Overview
 - 2. Overall Description
 - 2.1 Product Perspective
 - 2.2 Product Functions
 - 2.3 User Characteristics
 - 2.4 General Constraints
 - 2.5 Assumptions and Dependencies
 - 3. Specific Requirements

Figure : General structure of an SRS.

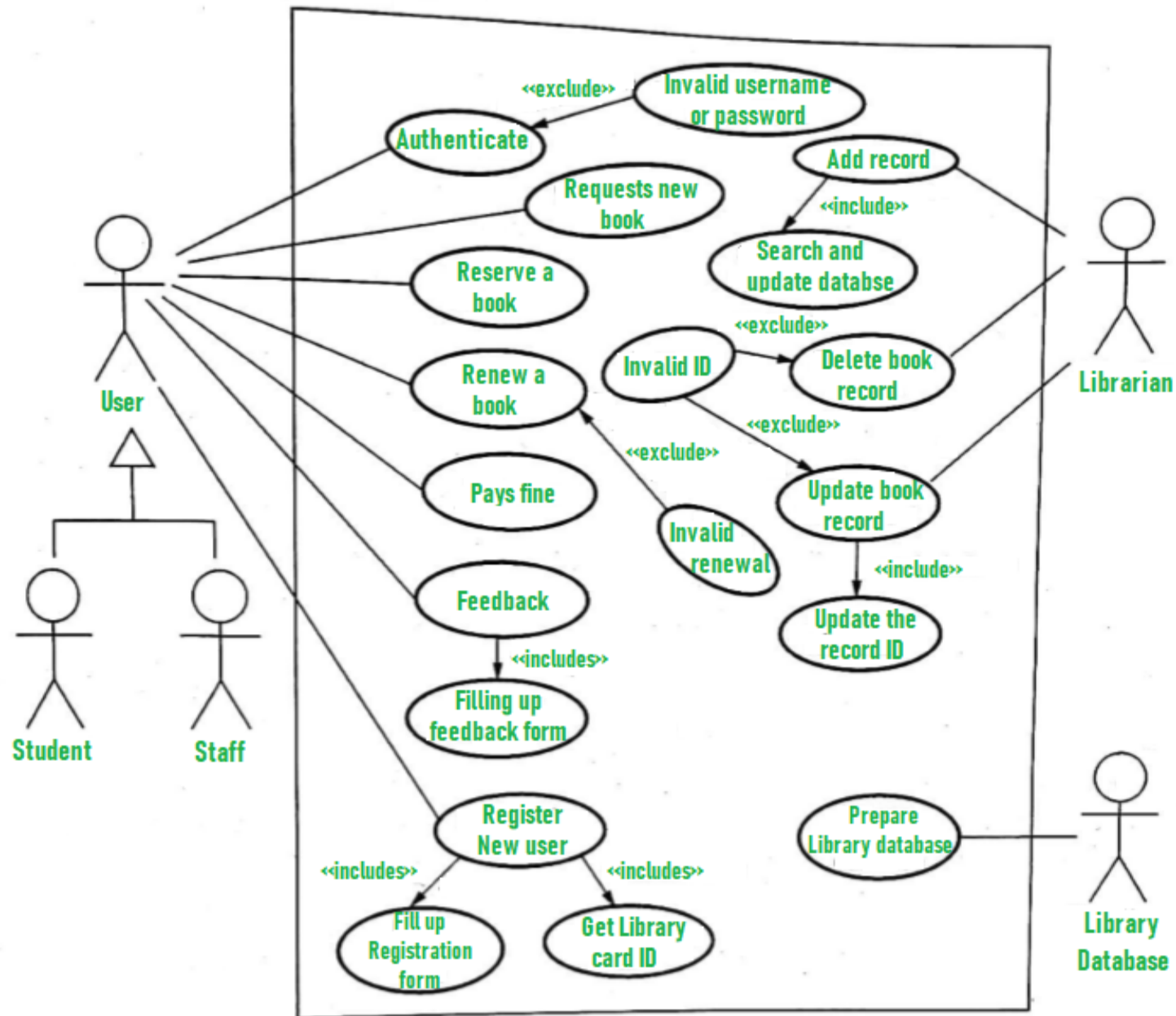
- 3. Specific Requirements
 - 3.1 External Interface Requirements
 - 3.1.1 User Interfaces
 - 3.1.2 Hardware Interfaces
 - 3.1.3 Software Interfaces
 - 3.1.4 Communication Interfaces
 - 3.2. Functional Requirements
 - 3.2.1 Mode 1
 - 3.2.1.1 Functional Requirement 1.1
 - :
 - 3.2.1.*n* Functional Requirement 1.*n*
 - :
 - 3.2.*m* Mode *m*
 - 3.2.*m*.1 Functional Requirement *m*.1
 - :
 - 3.2.*m*.*n* Functional Requirement *m*.*n*
 - 3.3 Performance Requirements
 - 3.4 Design Constraints
 - 3.5 Attributes
 - 3.6 Other Requirements

Figure 3.14: One organization for specific requirements.

Functional Specification with Use Cases

Term	Definition
Actors	A person or a system which uses the system being built for achieving some goal.
Primary actor	The main actor for whom a use case is initiated and whose goal satisfaction is the main objective of the use case.
Scenario	A set of actions that are performed to achieve a goal under some specified conditions.
Main success scenario	Describes the interaction if nothing fails and all steps in the scenario succeed.
Extension scenario	Describe the system behavior if some of the steps in the main scenario do not complete successfully.

Table : Use Case terms.



Validation of Requirements

- Type of errors that occur in an SRS: *omission, inconsistency, incorrect fact, and ambiguity.*
- Requirements validation must involve the clients and the users
- Requirements review team generally consists of client as well as user representatives.
- Requirements review is a review by a group of people to find errors and point out other matters of concern in the requirements specifications of a system
- Review group includes: author of the SRS, someone who understands the needs of the client, a person of the design team, and the person(s) responsible for maintaining the SRS. and someone not directly involved with product development, like a software quality engineer.

Validation of Requirements

- Goal of the review process
 - ✓ detecting requirement errors
 - ✓ consider factors affecting quality, such as testability and readability
 - ✓ uncover the requirements that are too subjective and too difficult to define criteria for testing that requirement.
- Checklists are frequently used in reviews to focus the review effort and ensure that no major source of errors is overlooked by the reviewers.

Validation of Requirements

A checklist for requirements review should include items like:

- ✓ Are all hardware resources defined?
- ✓ Have the response times of functions been specified?
- ✓ Have all the hardware, external software, and data interfaces been defined?
- ✓ Have all the functions required by the client been specified?
- ✓ Is each requirement testable?
- ✓ Is the initial state of the system defined?
- ✓ Are the responses to exceptional conditions specified?
- ✓ Does the requirement contain restrictions that can be controlled by the designer?
- ✓ Are possible future modifications specified?

Questions?