CS335FZ

Systems Analysis and Design

Dr. Lanlan Gao

Pay attention:

1. Except that the lab computer can not work, you can not use your laptop for the Quiz (let me know)

2. Internet signals may not be good at that moment, so just wait a second because you can only collect the Quiz once.

3. Provide 90 minutes to finish the 25 questions. When you complete, can leave (starting 3:50PM)

4. Please have a seat close to your groupmates for the TA's monitor (unopened textbook exam)

5. Lab 2 still needs to be finished within one week (deadline 16 May)

Object-Oriented Programming (OOP)

"Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships."

--Booth, G., Maksimchuk, R.A., Engle, M.W., Young, B.J., Connallen, J. and Houston, K.A., 2008. Object-oriented analysis and design with applications. ACM SIGSOFT software engineering notes, 33(5), pp.29-29.

- Object-oriented programming uses objects, not algorithms, as its fundamental logical building blocks.
- An object is an instance of some class, i.e., objects have an associated type (class).
- Classes may inherit attributes from parent classes (superclass).

Object-Oriented Design (OOD)

"Object-oriented design is a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design."

--Booch, G., Maksimchuk, R.A., Engle, M.W., Young, B.J., Connallen, J. and Houston, K.A., 2008. Object-oriented analysis and design with applications. ACM SIGSOFT software engineering notes, 33(5), pp.29-29.

- Object-oriented design uses *class* and object abstraction to logically structure systems (object-oriented decomposition), in comparison, a conventional structured design uses algorithmic abstractions.
- The term *object-oriented* ***design*** is referred to as **any method** that uses object-oriented approach to derive logical structures of a system.
- The **outcome** of object-oriented **design** is **a set of blueprint** for implementing a system.

Object-Oriented Analysis (OOA)

"Object-oriented analysis is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain."

--Booch, G., Maksimchuk, R.A., Engle, M.W., Young, B.J., Connallen, J. and Houston, K.A., 2008. Object-oriented analysis and design with applications. ACM SIGSOFT software engineering notes, 33(5), pp.29-29.

- Conventional structured analysis techniques, such as Data Flow Diagrams (DFDs), focuses on the flow of data within a system, in comparison, object-oriented analysis emphasizes the building of real-world models.
- The **outcome** of object-oriented **analysis** is **a set of object models**.
- An object-oriented design may start from the products of object-oriented analysis.

Object Identification

- During analysis and the early stage of design, two tasks are of importance:
  - **Identify the classes** that form the vocabular of the problem domain
  - **Create object models** that reflect the requirements of the problem
- Object-oriented analysis common approaches
  - Textual/Use Case Analysis
  - Classical Analysis
  - Behavior analysis
  - CRC (Class-Responsibility-Collaboration) Cards

  - Structured analysis
    - Process Modelling (DFDs - Data Flow Diagrams)
    - Entity-Relationship Modelling (ERDs - Entity-Relationship Diagrams)

Textual/UseCase Analysis (1)

- Perform textual analysis by reviewing the use case diagrams and analyze use case descriptions to identify potential objects, attributes, operations, and relationships.

A common or improper noun implies a class of object.
e.g., "*a **user** needs to login to access the User Story Index Cards of a selected project.*"
class User { }

A proper noun or direct reference implies an instance of a class, i.e., an object.
e.g., "*a **product manager** can create, modify, and remove project related information.*" A product manager is a kind of user of the system.
User productMgr = new ProductManager( );

An adjective implies an attribute of an object.
e.g., "an ***updated*** User Story Index Card needs to be persisted in the project file."
class UserStory {private Date lastModified; }

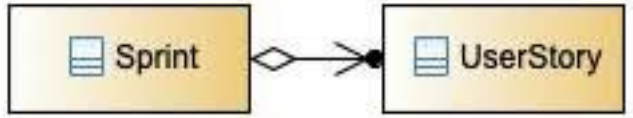
A transitive verb implies an operation.
e.g., "a project manager can ***create*** a project."
class Project {public createProject(Path projectName) { … } }

A doing verb implies an operation.
e.g., "when ***updating*** a filed of a User Story Index Card, the input must also be verified."
class UserStory { public update(…) { … } }

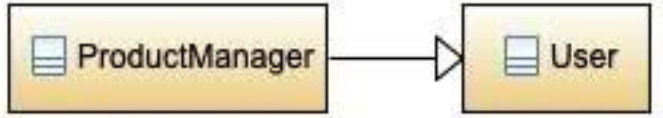
Textual/UseCase Analysis (2)

An intransitive verb implies an exception.
e.g., *"when a filed of a User Story Index Card is modified, the inputs must be **verified**."*
Check the type of input, e.g., Integer, Float, String, Date, etc.

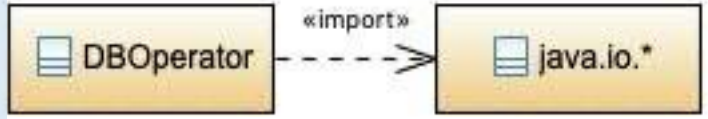
A having verb implies an aggregation or association relationship. e.g., *"a Sprint **has** one or more UserStories."*



A being verb implies a classification relationship between an object and its class.
e.g., *"a product manager **is** a type of user."*



A set of commonly used verbs for dependency: *use, import, depend, refine, extend, include, access, instanceOf, bind, instantiate*, etc.
e.g., *"the system needs the permission to **access** local disk in order to load project files."*



- **The primary purpose of textual/UseCase analysis is to create an initial set of classes**.
- It is often used with other techniques.

Classical Analysis

Tangible things	House, Sensor
Roles	Project Manager, Tester, Staff
Events	Interrupt, Request, Response
Interactions	Meeting, Loan

--Stephen, M., 1988. Object-Oriented Systems Analysis: Modeling the World in Data.

People
Places
Things
Organizations
Concepts
Events

--Ross, R. 1987. Entity Modeling: Techniques and Application. Boston, MA: Database Research Group.

Structure
Other Systems
Events
Devices
Locations
Organizational Units

--Coad, P., and Yourdon, E. 1990. Object-Oriented Analysis. Englewood Cliffs, NJ: Prentice-Hall.

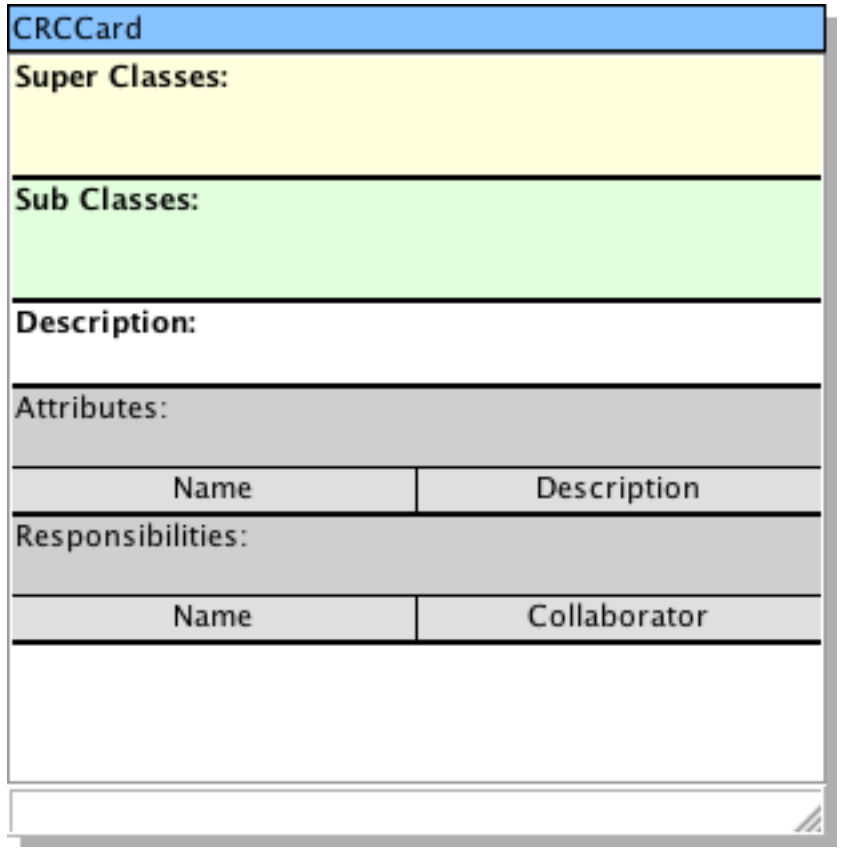
- Derives classes and objects primarily from the principle of classical categorization.
- Focus on tangible things in the problem domain.

Behavior Analysis

- Focuses on dynamic behavior as the primary sources of classes and objects.
- Forms classes based on groups of objects that have similar behavior.
- Group things that have common responsibilities.
  - Hierarchies of classes are created from general responsibilities to specialized behavior.
- Derived from system functions
  1. Assign system behaviors to parts of the system
  2. Understand who initiates the behaviors and who participates in the behaviors
  3. Initiators and participants that paly significant roles are considered as objects.

CRC Cards

- Class-Responsibility-Collaboration (CRC) cards are used to document the responsibilities and collaborations of a class.
- The set of CRC cards contains all the information for building logical structural model of the problem under investigation.
- The analyst can use the CRC cards and role-playing with Use Cases to discover hidden classes, objects, properties, operations and relationships.



Responsibilities and Collaborations

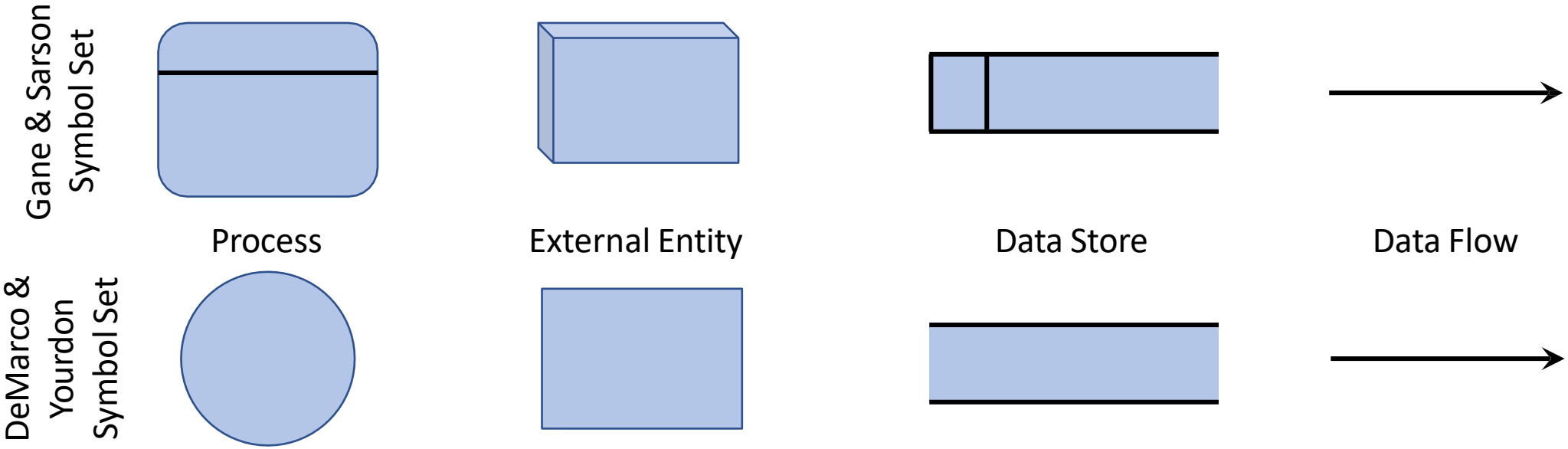
- Responsibilities are related to the behaviors of an object in terms of its role.
- Responsibilities are assigned to classes of objects during analysis and object design.
- A responsibility is different from a method of a class, but methods fulfill responsibilities.
  - A *little* responsibility might be fulfilled by a single method of an object, whereas a *big* responsibility could take multiple classes and methods.
- Two types of responsibilities:
  - Doing responsibilities of an object
    - E.g., doing a calculation, creating an object, initiating/controlling action/activities in other objects.
  - Knowing responsibilities of an object
    - E.g., knowing about encapsulated data, related objects.
- A set of classes that support a business process or a Use Case forms *collaborations*.

CRC Cards Analysis

1. Review User Case descriptions.
   - Pick a specific Use Case to role-play.
   - Start with the most important, the most complex, or the least understood Use Cases.
   - Identify object and classes from the Use Case.
     - Preferred to have Use Case description written in the format of <Subject> -- <Verb> -- <Direct Object> -- <Preposition> -- <Indirect Object> (SVDPI).
     - Perform textual/classical/behavior analysis
   - Create a CRC Card for each identified class.
2. Identify relevant actors and objects
   - Identify the relevant roles to play.
   - Each role is associated with either an actor or an object.
3. Role-play scenarios of Use Case
   - Each team member acts as an instance of the role (identified in Step 2).
   - Each role player askes some simple questions
     4. What do you know? (the knowing responsibilities)
     5. Wan can you do? (the doing responsibilities)
   - Fill in the CRC Card for your role
4. Repeat the above steps until all Use Cases are exercised.

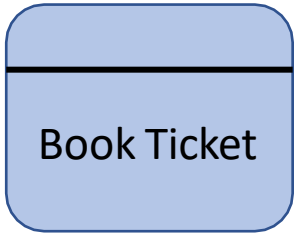
Structured Analysis – Process Modeling

- System analysts use Data Flow Diagram (DFD) to show how data moves through an information system.
- Data Flow Diagrams show the processes that change or transform data.
- A set of data flow diagrams provides a logical model that shows what system does, but not how it does it, i.e., data and process modelling.



Basic Rules in DFDs -- *Process*

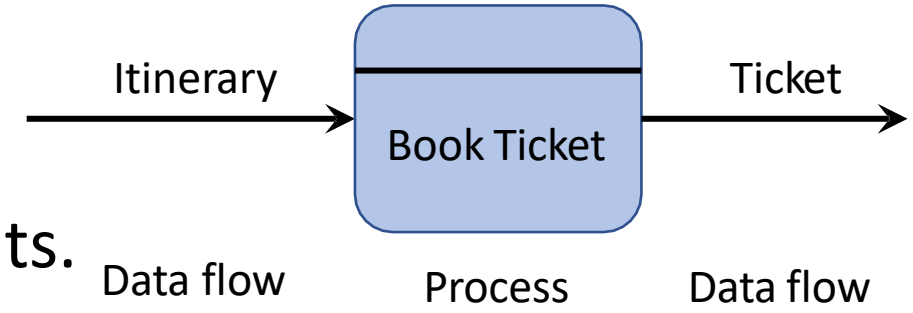
- A *process* receives input data and produces output that has a different contents, form, or both.
- A *process* can be simple or complex.
- A *process* contain the business logic that transforms the data and produces the desired output.
- A process name identifies a specific function
  - A verb followed by a singular noun.



- Processing details are not shown in a DFD; thus, a process is often called a black box.
  - Details are often documented in an associated *process description*.

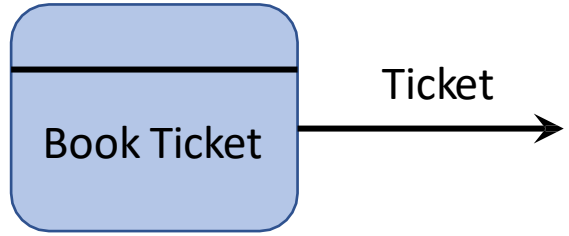
Basic Rules in DFDs – *Data Flow*

- A *data flow* is a path for data to move from one part of the system to another.
- A data flow represents one or more data items.
  - E.g., a Sprint ID, the Reporter of an *issue*, a User Story (consists of many fields).
- A data flow has a name.
  - A singular noun and optionally an adjective.
- A process can have multiple inputs and outputs.
- A data flow must have a process on at least one end.

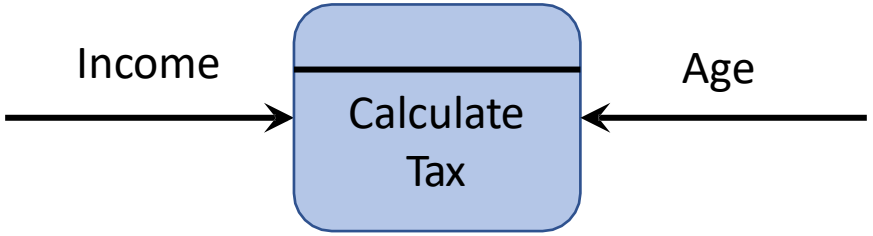


Basic Rules in DFDs – To Avoid

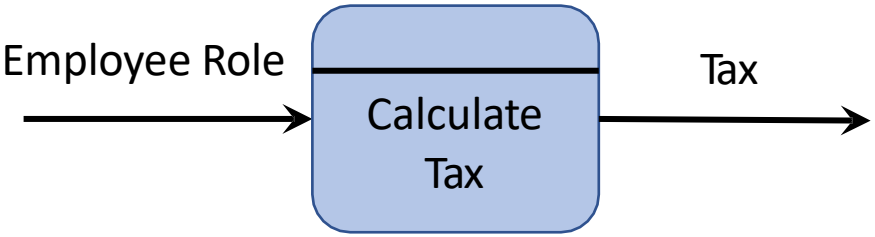
- Spontaneous generation
  - A process has output, but no input data flow



- Black hole
  - A process has input, but no outputs

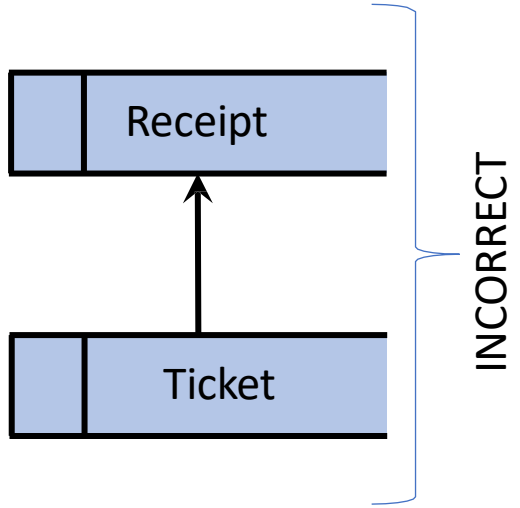


- Gray hole
  - A process has at least one input and one output, but the input is not sufficient to produce the output.



Basic Rules in DFDs – Data Store

- A data store represent data that the system stores or that will be used by one or more processes.
- A data store does NOT show its detailed contents.
  - Structures, format and elements are defined in the *data dictionary*.
- The physical characteristics of a data store are NOT important.
- A data store name is a plural name consisting of a noun and optionally adjectives.
- A data store can NOT connect to another data store directly.



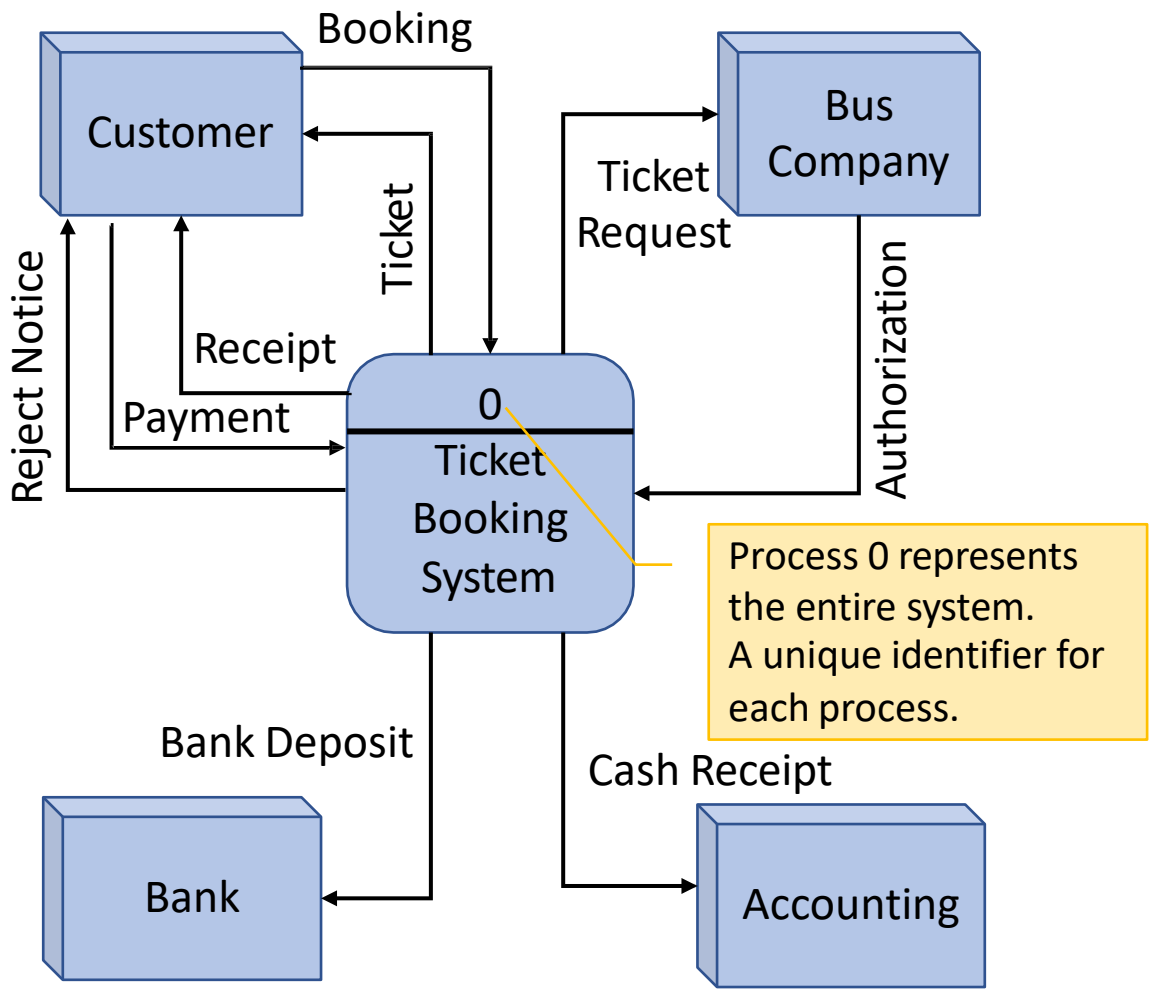
Basic Rules in DFDs – External Entity

- Only external entities that provide data to the system or receive output from the system will be shown in a DFD diagram.
- The boundaries of the system can be determined from where external entities appear.
- An external entity that feed data to the system is called a *source*; an entity that receives data from the system is called a *sink*.
- An external entity name is the singular form of an organization, a department, another information system or a person.

- External entities can NOT send data to or receive data from each other directly.
- An external entity can NOT send data to or receive data from data stores.

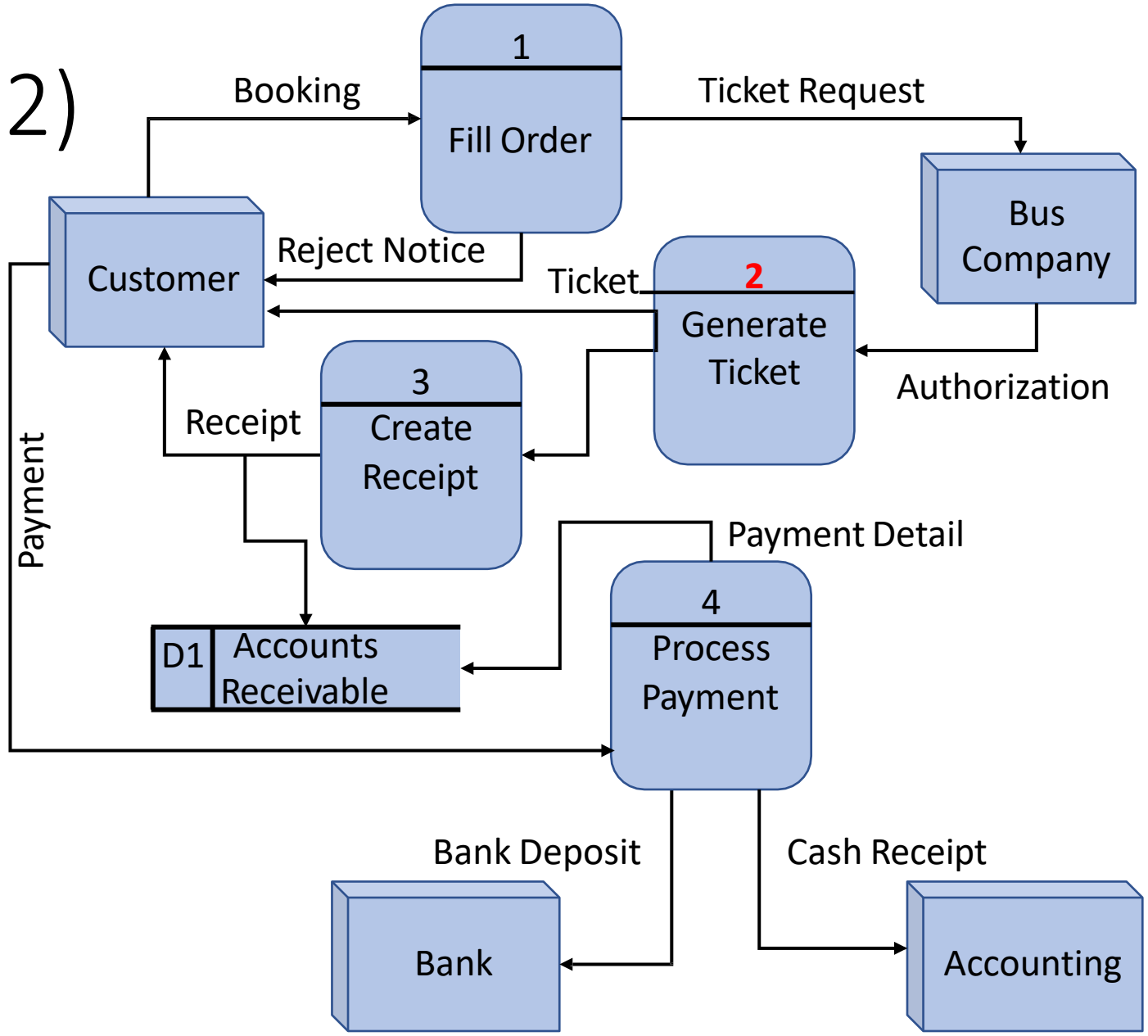
Creating DFDs (1)

- Step 1: Draw a context diagram
  - In DFD, a context diagram is a top-level view of a software system.
  - a context diagram defines the system's boundary and scope.
  - Identify external entities that will supply data to or receive data from the system under development.
  - Data stores are NOT included in the context diagram.
  - The entire system under development is denoted as process 0 (zero).



Creating DFDs (2)

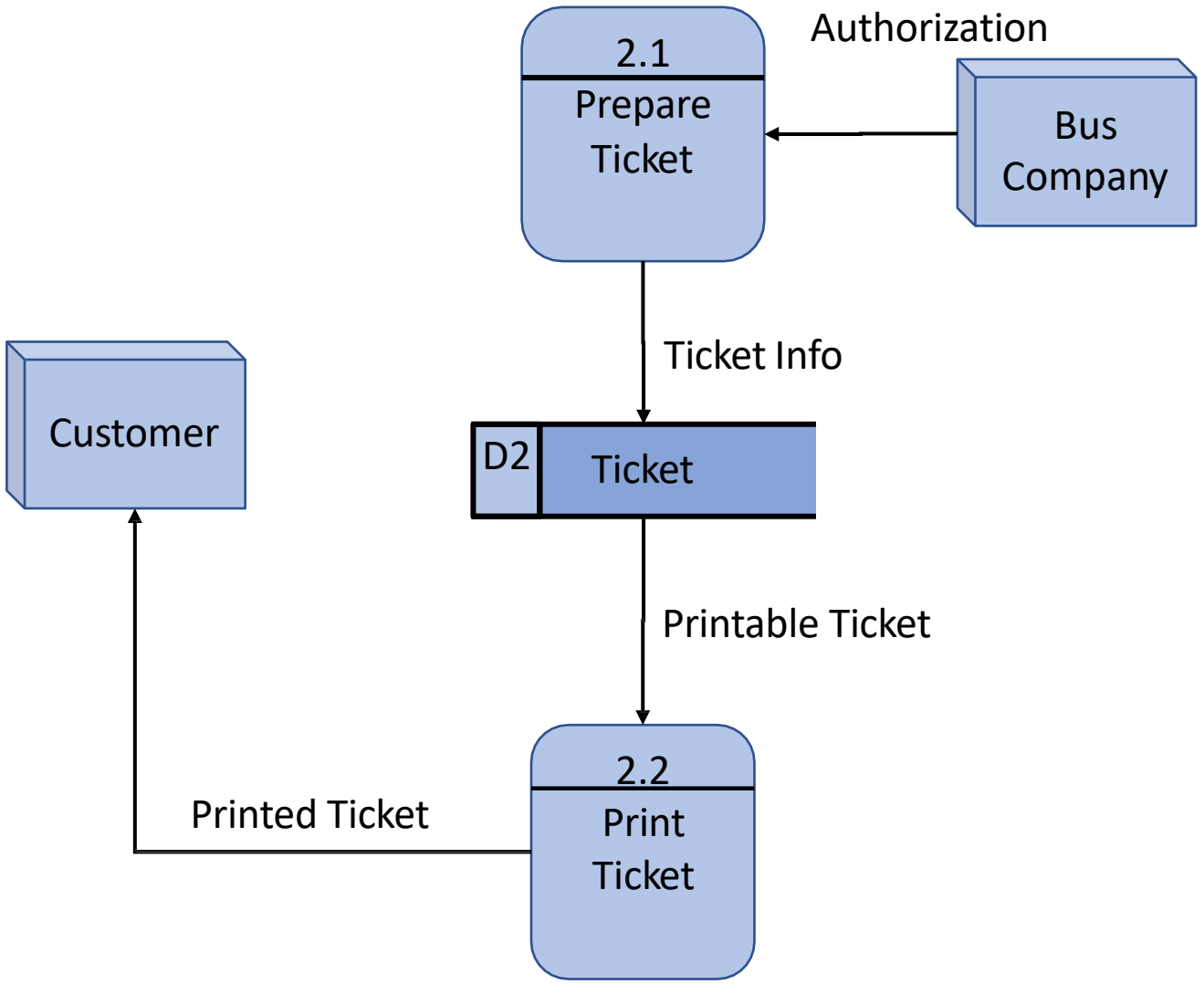
- Step 2: Draw a *Diagram 0* DFD
  - Diagram 0 provides an overview of all the components of the system.
  - It shows the main internal processes, data flows, and data stores of the process 0.
  - At the level of Diagram 0, it also includes the identified external entities in the context diagram.
  - Process numbers do NOT mean the order of process execution.



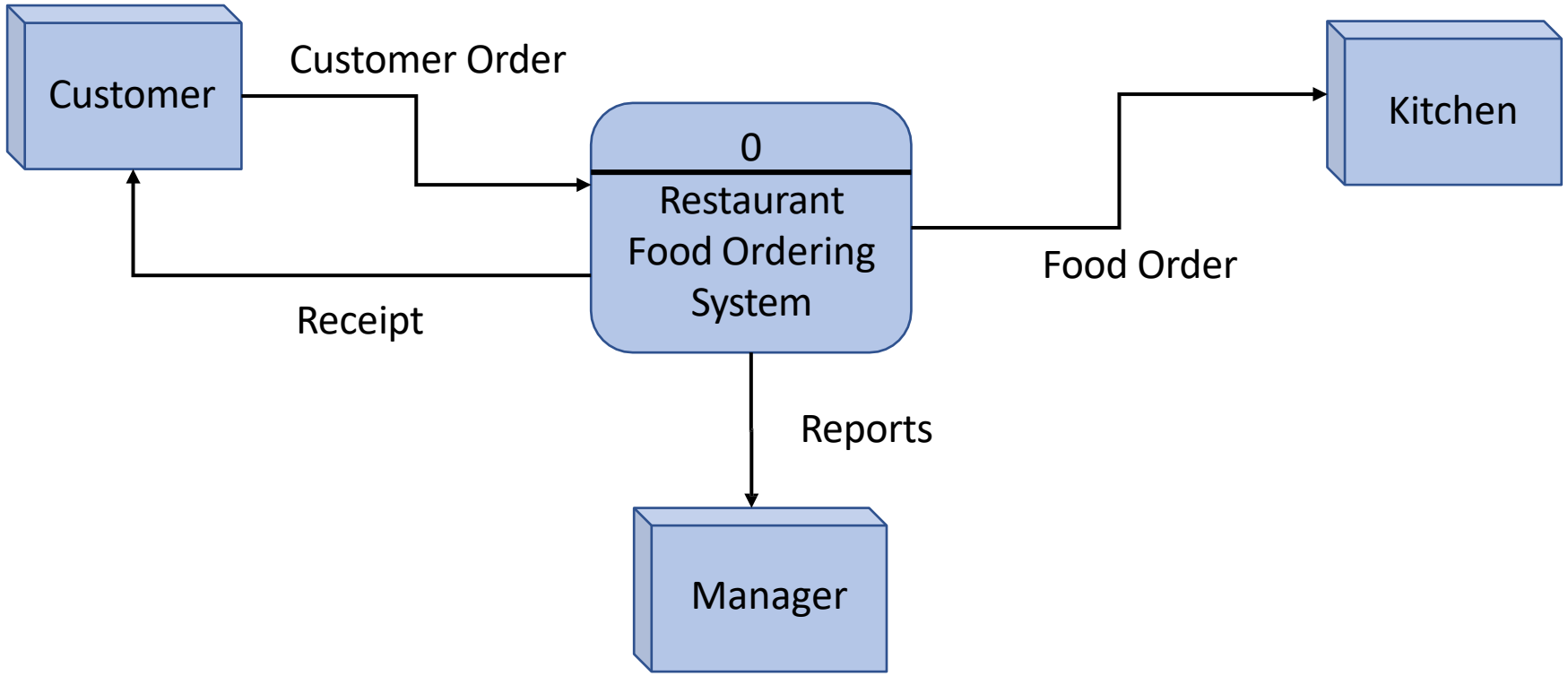
Accounts receivable *(AR) is the balance of money due to a firm for goods or services delivered or used but not yet paid for by customers.*

Creating DFDs (3)

- Step 3: Draw the Low-level DFDs
  - The level of low-level DFDs can only be determined when all functional primitives are identified (leveling).
  - All low-level DFDs are based on the processes identified at the upper level.
  - The input and output data flow in all low-level DFDs must be aligned properly (balancing).
    - i.e., the input and output of Process 2 in Diagram 0 are retained in the Diagram 1

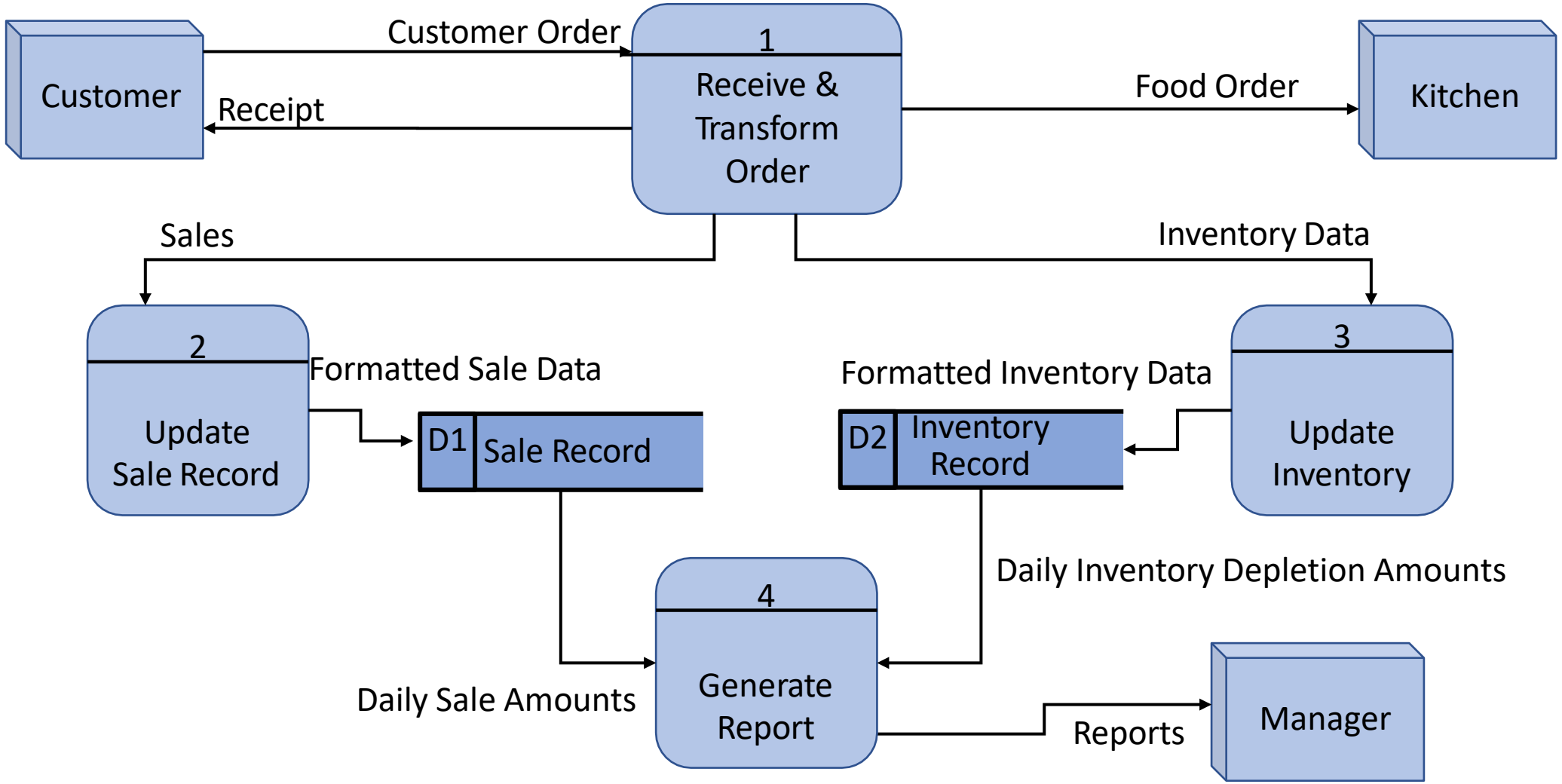


DFD Example – Restaurant Food Ordering System (Context Diagram)



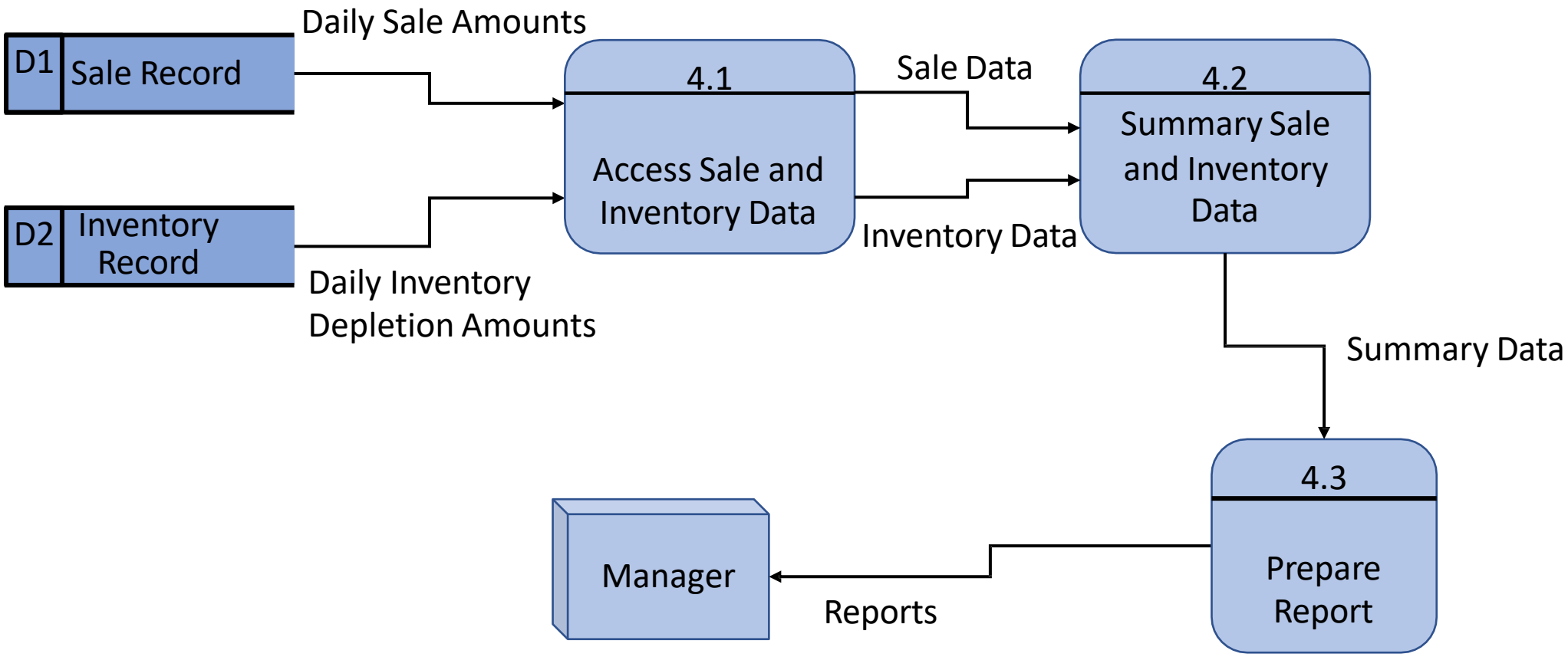
Context Diagram

DFD Example – Restaurant Food Ordering System (Level-0)



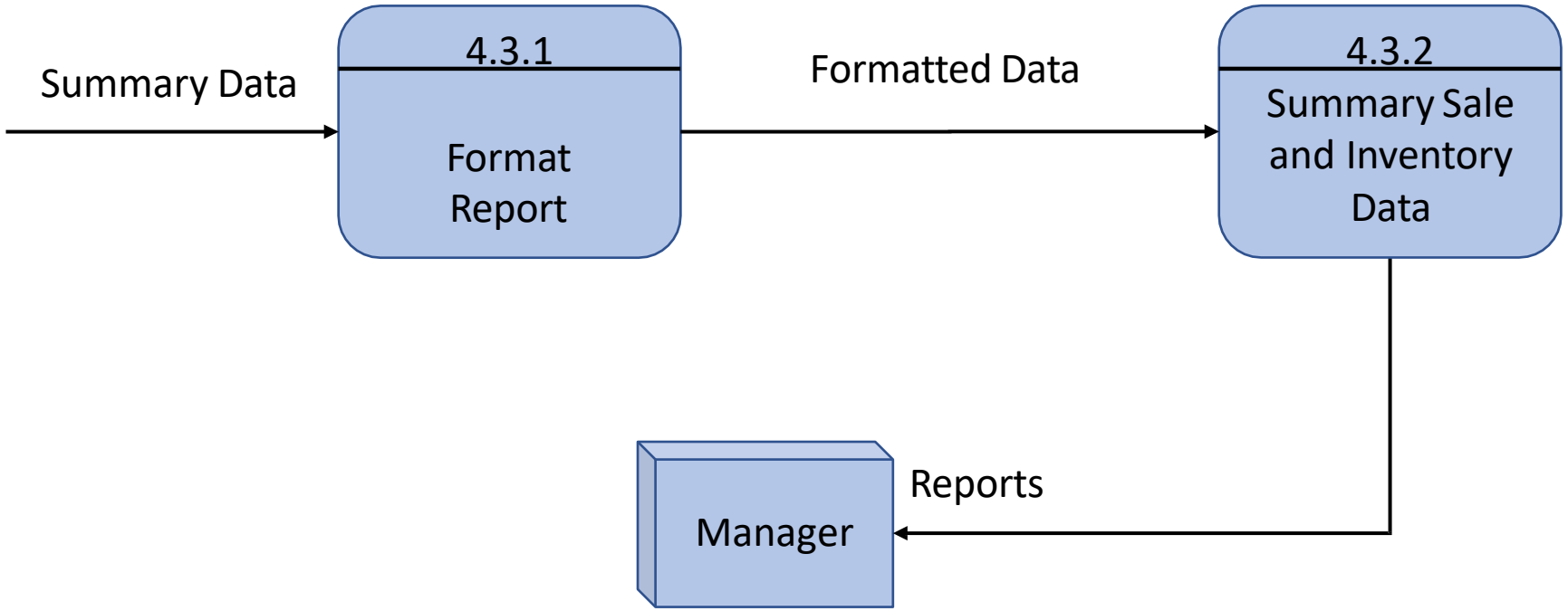
Level-0

DFD Example – Restaurant Food Ordering System (Level-1 <Process 4>)



Level-1

DFD Example – Restaurant Food Ordering System (Level-2 <Process 4.3>)



Level-2

DFD Summary

- Data Flow Diagrams are used for modelling data and processes.
- DFDs show the data movement and transformation in the system.
- DFDs organize the level of details in hierarchies.
  - Context diagram represents the system scope and its external dependencies.
  - Diagram 0 shows the major processes, data stores, external entities and data flows in the system.
  - Low-level diagrams (Diagram 1, 2, etc.) show additional details of the system.
- DFDs should accurately reflect the information system being modelled.
- Complete sets of DFDs should extend to the *primitive* level, where every component reflect certain irreducible properties. This is an iterative process.

Problems and Exercises

Using the example of a retail clothing store in a mall, list relevant data flows, data stores, processes, and sources/sinks. Observe several sales transactions. Draw a context diagram and a level-0 diagram that represent the selling system at the store. Explain why you chose certain elements as processes versus sources/sinks. (submit to Teams)

Conceptual Data Modelling

"A detailed model that captures the overall structure of organizational data and is independent of any database management system or other implementation considerations."

--Valacich, J.S., George, J.F. and Valacich, J.S., 2017. Modern systems analysis and design. Boston: Pearson.

- Data modelling develops the definition, structure, and relationships within the data.
- A data model explains what the organization does and the rules that govern the work performed in the organization.
- A data model does not care how or when data are processed or used.
- Conceptual data modelling is often performed with other requirements analysis and systems analysis activities, such as process modeling and logic modelling.
  - The works and activities are commonly coordinated and shared through project dictionary or repository maintained by a common Computer-Aided Software Engineering (CASE) software tool.
- The most common technique used for data modelling is Entity-Relationship (E-R) diagramming.

The Importance of Data Modelling

- The characteristics of data are important in the design of databases, programs, computer screens and printed reports.
- Data are the most complex aspects of many modern information systems.
  - Transaction processing systems, such as an order processing system, can have considerable process complexity in validating data, reconciling errors, and coordinating the movement of data.
- The characteristics about data, such as format and entity relationship, are relatively stable and are largely the same across organizations in the same business domain.
  - An information system design based on a data orientation, rather than a process or logic orientation, should have a longer useful life.

Gathering Information for Conceptual Data Modelling

- Conceptual data models can be developed from scratch, refined from existing data models, or purchased (standard data models for a particular business domains).

- Data modelling is often done from a combination of perspectives.
  - The top-down approach
    - Derives the business rules for a data model from an understanding of the nature of the business (interview, asking specific questions)
  - The bottom-up approach
    - Reviewing specific business documents, such as reports and receipts, etc.

The Top-Down Approach – Typical Interview Questions

Determine data entities and their relationships: *What are the subjects/objects of the business?* What types of people, places, things, materials, events, etc. are used or interact in this business, about which data must be maintained? How many instances of each object might exist?
Determine primary key: *What unique characteristic (or characteristics) distinguishes each object from other objects of the same type?* Might this distinguishing feature change over time or is it permanent? Might this characteristic of an object be missing even though we know the object exists?
Determine attributes and secondary keys: *What characteristics describe each object?* On what basis are objects referenced, selected, qualified, sorted, and categorized? What must we know about each object in order to run the business?
Determine cardinality and time dimensions of data: *Over what period of time are you interested in these data?* Do you need historical trends, current "snapshot" values, and/or estimates or projections? If a characteristic of an object changes over time, must you know the obsolete values?
Determine supertypes, subtypes, and aggregations: *Are all instances of each object the same?* That is, are there special kinds of each object that are described or handled differently by the organization? Are some objects summaries or combinations of more detailed objects?
Determine relationships and their cardinality and degree: *What events occur that imply associations among various objects?* What natural activities or transactions of the business involve handling data about several objects of the same or a different type

Reference: Gottesdiener, E., 1999. Turning rules into requirements. Application Development Trends, 6(7).
Herbst, H., 1997. Business rule-oriented conceptual modeling. Springer Science & Business Media.
Witt, G., 2012. Writing effective business rules. Elsevier.

The Bottom-Up Approach

Your ACCOUNT Number: 12345 Your PIN: 12345 Route: FRI

DAPENG DONG
CS MU
CORK CITY
CO. CORK

INVOICE
INVOICE DATE: **02/04/2021**
INVOICE NO. : **DW 12345**

Description	Period Of Cover	Price incl VAT	VAT Rate
Service Charge	01/04/2021 - 30/06/2021	€67.25	13.50%
Text Communication Charge (FOC to APP Users)	01/04/2021 - 30/06/2021	€1.35	13.50%

VC	Rate	VAT Excl	VAT
1	13.50%	€60.44	€8.16

VAT Excl:	€60.44
VAT Total:	€8.16
Invoice Total:	**€68.60**
Balance brought forward as of 02/04/2021:	**€0.00**

GreenStar Bill

- If we were designing a billing system for an GreenStar, we can determine that the following data entities should be of interest:
- ACCOUNT NUMBER
- PIN
- ROUTE
- NAME
- ADDRESS
- INVOICE
  - INVOICE DATE
  - INVOICE NUMBER
- DESCRIPTION
- PERIOD OF COVER
- PRICE
- VAT RATE
- TOTAL
- BALANCE BROUGHT FORWARD

These data entities should be included in DFDs.

Entity-Relationship Modelling

- An E-R model is a detailed logical representation of data for an organization or for a business area.

- An E-R Diagram is a graphical representation of an E-R model.
  - The notation of ERDs may vary from organization to organization

- E-R modelling is common a method used in database design.

- The basic E-R modeling notation uses three main constructs:
  - *data entity*, *relationships*, and their associated *attributes*.

E-R Model – Entities

- An entity is a person, place, object, event, or concept in the user environment.
  - Person: STUDENTS, LECTURER, CLEARK, etc.
  - Place: STORE, WAREHOUSE, DEPARTMENT, etc.
  - Object: SENSOR, PRODUCT, CAR, etc.
  - Event: SALE, REGISTRATION, PURCHASE, etc.
  - Concept: COURSE, ACCOUNT, STOCK, etc.
- An entity has its own identity that distinguishes it from other entity.
- An entity type (or entity class) is a collection of entities that share common properties or characteristics (similar to the *class* in OOD).
- An entity instance is a single occurrence of an entity type (similar to the *object* in OOD)
  - E.g., there is one LECTURER entity type in university, but there may be hundreds (or thousands) of instances of this type stored in the university's staff management database.

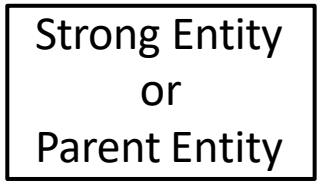
Naming Entity Types

- An entity type name should be a singular noun.
  - E.g., STUDENT, LECTURER, ACCOUNT, etc.
- An entity name should be descriptive and specific to the organization.
  - E.g., a PURCHASE ORDER for orders placed with suppliers is different from CUSTOMER ORDER that are placed by customers. Both entity types should not be named ORDER.
- Event entity types should be named for the result of the event, not the activity or process of the event.
  - E.g., the event of project managers assigning a developer to work on a project results in an ASSIGNMENT entity type.

Defining Entity Types

- An entity type definition should include:
  - what the unique characteristic(s) is (are) for each instance of the entity type.
  - what entity instances are included and not included in the entity type.
  - when an instance of the entity type is created and removed.
  - when an instance might change into an instance of another entity type.
    - E.g., "*a bid for a construction company becomes a contract once it is accepted*."
  - what history is to be kept about entity instances.

ERD Diagrams for Entities



- Independent from other entities
- A strong entity often has weak entities that depend on them.
- A strong entity has a primary key.

- A weak entity depends on other entity type.
- A weak entity does not have primary keys.
- A weak entity has no meaning in the diagram without its parent entity.

- An associative entity relates the instances of server entity types.
- An associative entity often contain attributes specific to the relationship between those entity instances.

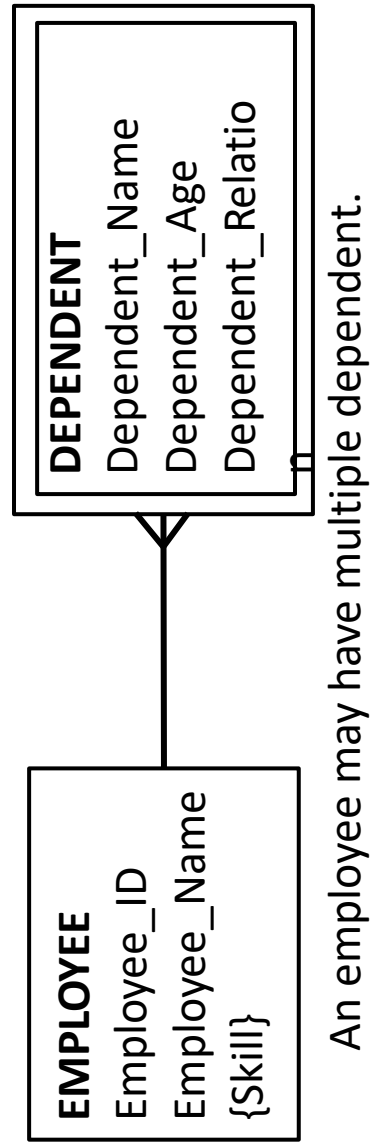
E-R Model – Attributes

- Each entity type has a set of attributes (similar to the class attributes in OOD).
- An attribute is a property or characteristic of an entity.
  - E.g., Entity type STUDENT may have attributes including Student_ID, Student_Name, Major, Contact_Number, etc.
- Every entity type must have an attribute or set of attributes that distinguishes one instance from other instances of the same type. The attribute or the set of the attributes are called *Candidate Key*.
- An attribute that may contain multiple values is called a *Multivalued Attribute*.
- Several attributes that repeat together are called a *Repeating Group*.
- Use a pair of curly brackets to enclose a repeating group or a multivalued attribute.
- An attribute that must have a value for every entity instance is called a *Required Attribute.*
- An attribute that may not have a value for every entity instance is called an *Optional Attribute.*
- An attribute that has meaningful component part are called a *Composite Attribute*.
- An attribute's value that can be computed from other data in the database is called a *Derived Attribute*.

E-R Model – Attribute Example

Entity Type

EMPLOYEE	Entity Name (All Capitalized)
Employee_ID	Candidate Key (Underlined) & Required (Bold)
Employee_Name (First_Name, Last_Name)	Composite Attribute & Required
{Dependent_Name, Dependent_Age, Dependent_Relation}	Repeating Group & Required
{Skill}	Multivalued Attribute & Required
Date_of_Birth	Optional Attribute
[Employee_Age]	Derived & Optional Attribute



E-R Model – Relationships

- A relationship is an association between t he instances of one or more entity types that is of interest to the organization.
- An association usually means that an event has occurred or that there exists some natural linkage between entity instances.
- Relationships are labelled with verb phrases.
- The number of entity types that participate in a relationship is called the *degree* of a relationship.
  - Unary relationship (or recursive relationship), Binary relationship, and Ternary relationships are common.
  - Higher-degree of relationships are possible, but they are rarely encountered in practice.

Basic Relationships

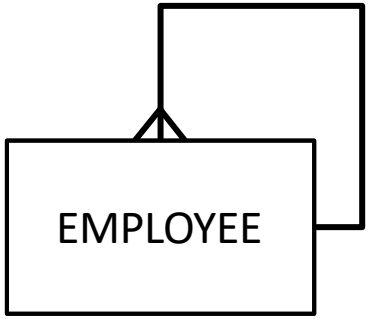
One: ——

Many: —<

Degree of a Relationship

Unary Relationships

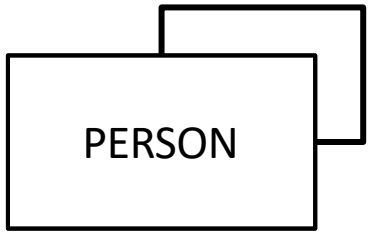
Manages



A manager (an employee) manages multiple employees

One-to-Many

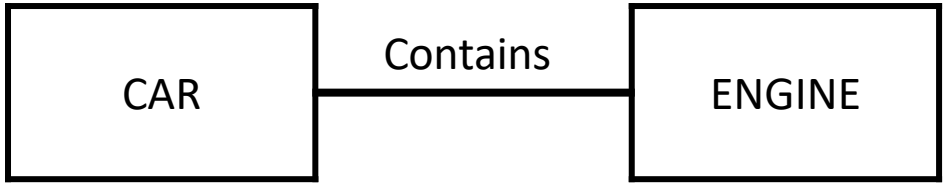
Is_married_to



A person is married to another person

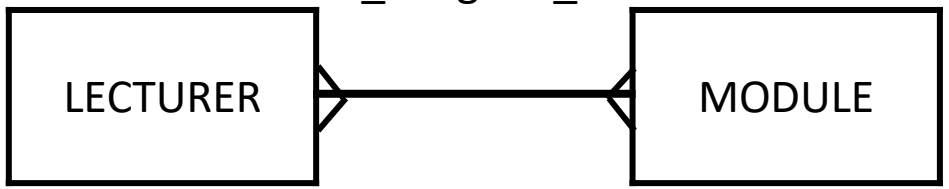
One-to-One

Binary Relationships



A car contains one engine

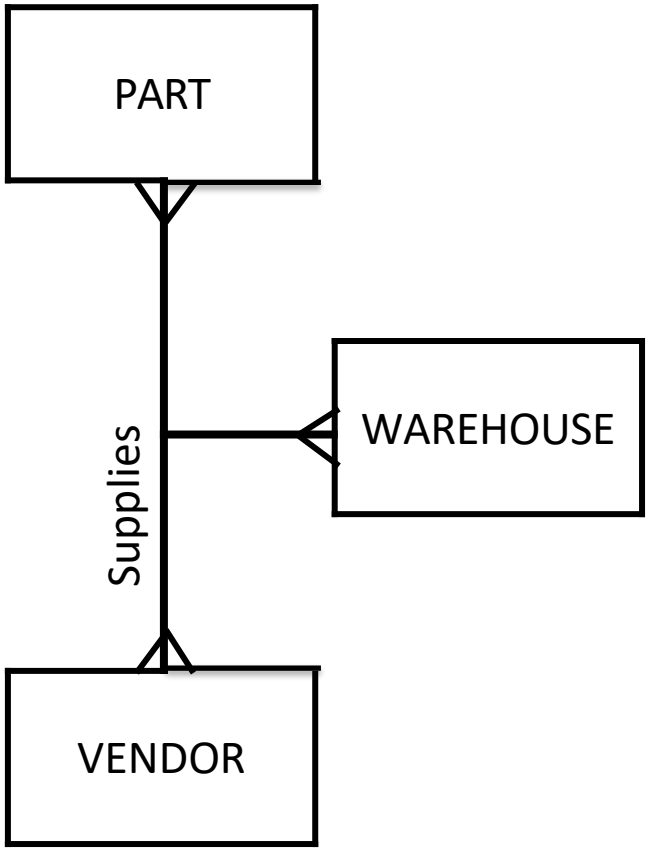
One-to-One



A lecture(s) is assigned to module(s)

Many-to-Many

Ternary Relationships



The relationship Supplies tracks the quantity of a given part that is shipped by vendors to selected warehouses.

Reference: Valacich, J.S., George, J.F. and Valacich, J.S., 2017. Modern systems analysis and design. Boston: Pearson.

Cardinalities in Relationships

The number of instances of entity ***B*** that can (or must) be associated with each instance of entity ***A***.

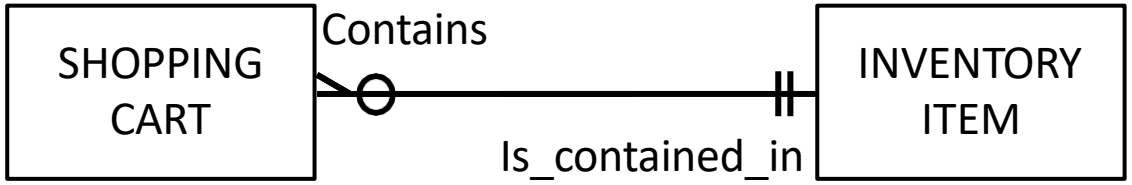
Mandatory Many
(One or many, 1..*)

Optional One
(Zero or one, 0..1)

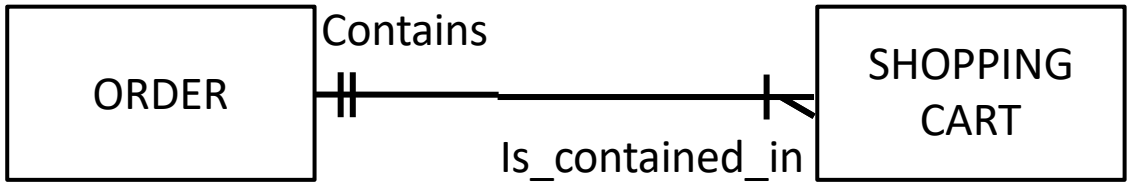
Mandatory One
(One and only one, 1)

Optional Many
(Zero, or one, or many, 0..*)

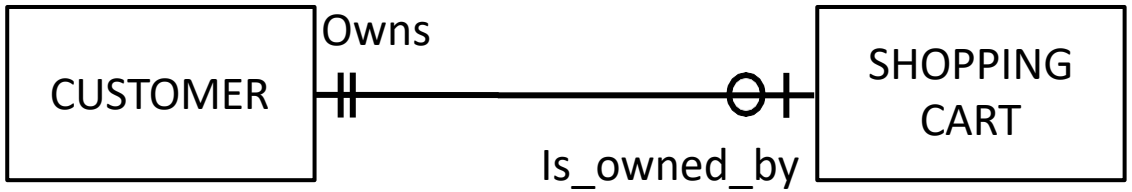
The Crow's Foot Symbols



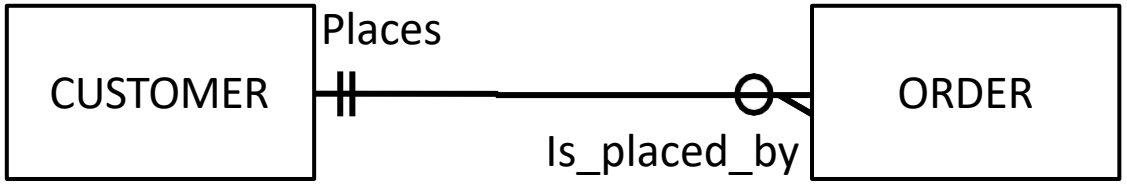
An inventory item can be contained in zero, or one, or many shopping cart instances.



An order must contain one or more shopping cart instances.



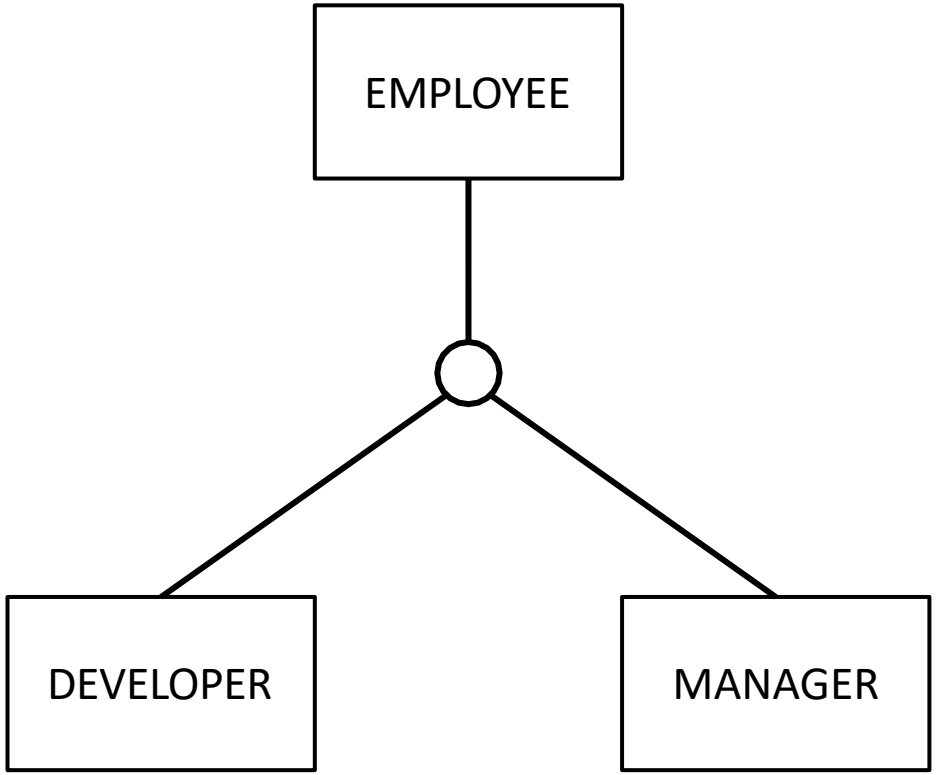
A customer can own zero or one shopping cart instance.



A customer can place zero, or one, or many orders

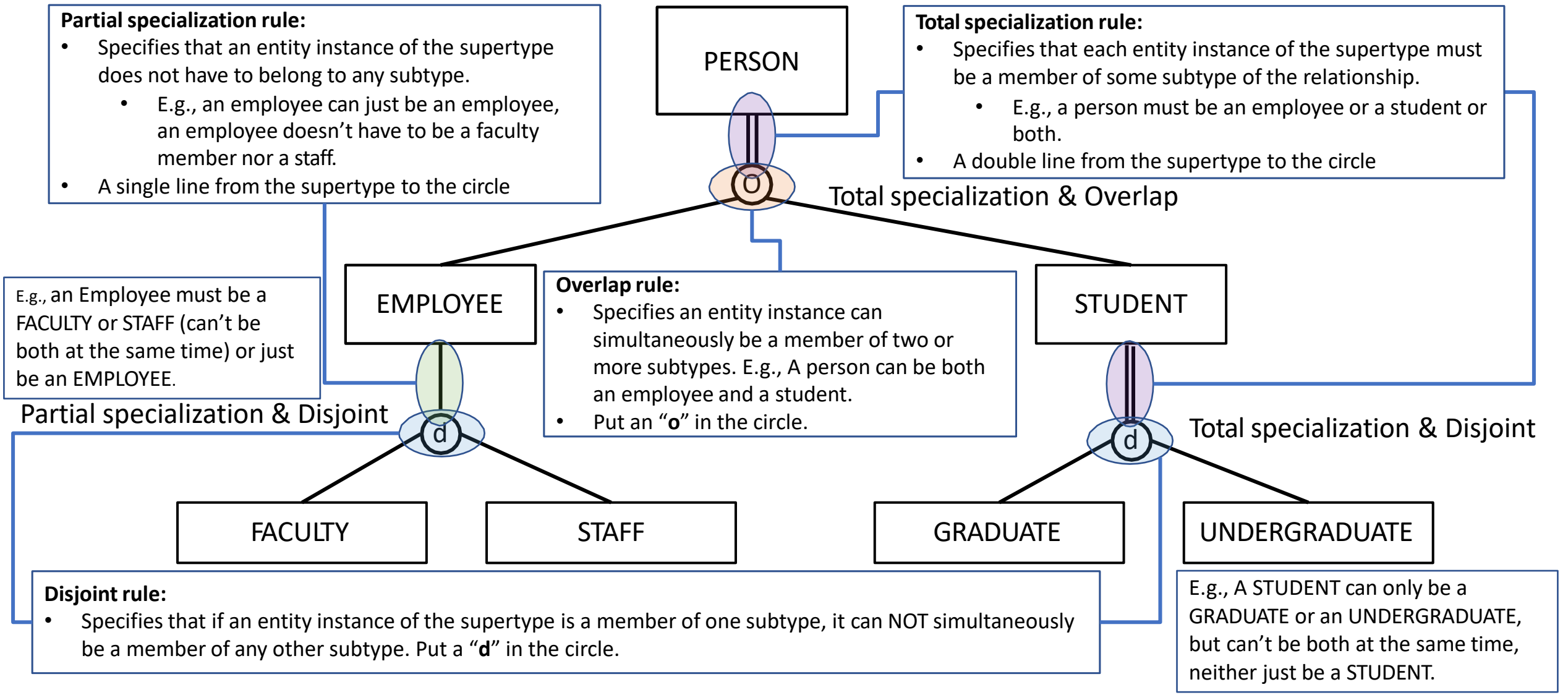
Supertypes and Subtypes

- Subtype (similar to the child classes in OOD)
  - *A subgroup of the entities in an entity type that is meaning full to the organization and that shares common attributes or relationships distinct from other subgroupings.*
- Supertype (similar to the super classes in OOD)
  - *A generic entity type that has a relationship with one or more subtypes*.



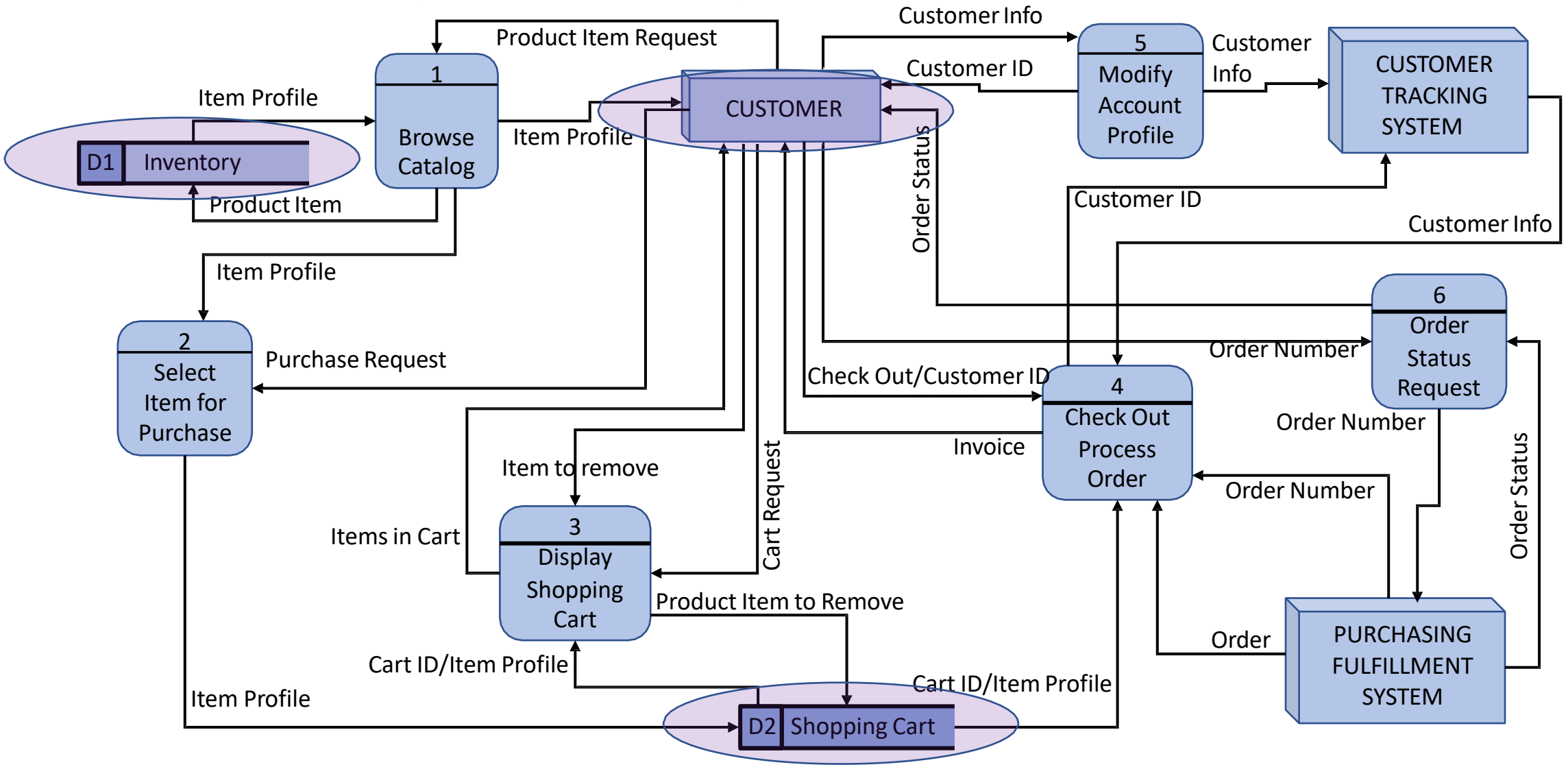
Inheritance in OOD

Rules Govern Supertype/Subtype Relationships



The Connections between DFDs and Data Models

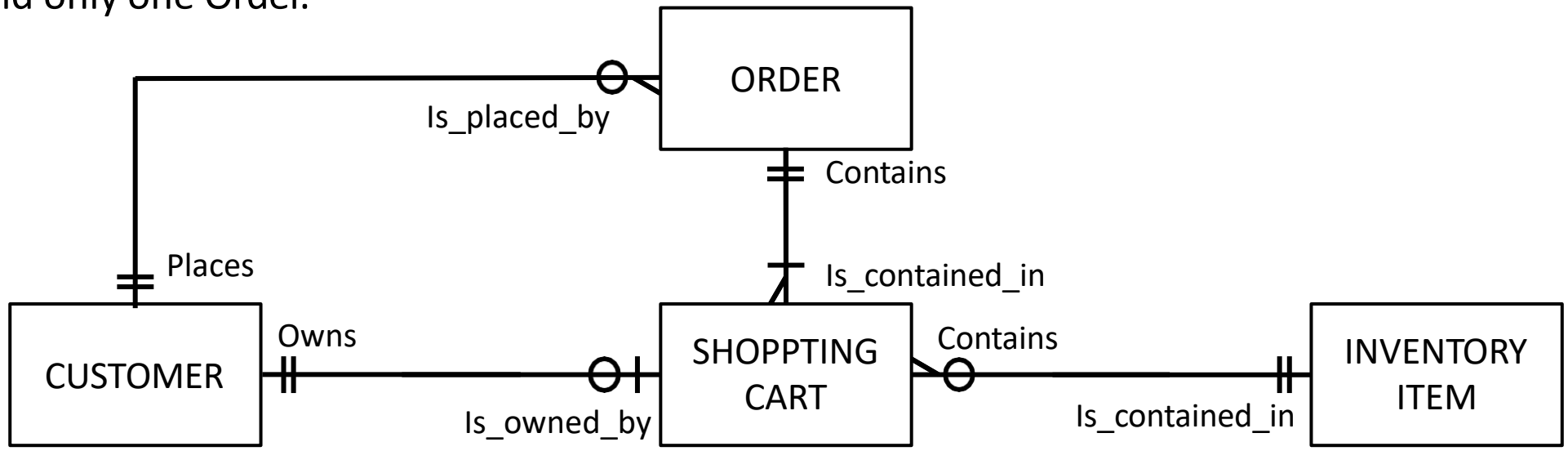
- Data elements included in data flows (DFDs) also need to be present in the data model, and vice versa.
- Each data store in a process model must map to data entities.



Reference: Valacich, J.S., George, J.F. and Valacich, J.S., 2017. Modern systems analysis and design. Boston: Pearson.

Data Stores and External Entities to E-R Model

1. Each Customer *owns* zero or one Shopping Cart instances; each Shopping Cart instance *is owned by* one and only one Customer.
2. Each Shopping Cart instance *contains* one and only one Inventory item; each Inventory item *is contained in* zero or many Shopping Cart instances.
3. Each Customer *places* zero, or one, or many Orders; each Order *is placed by* one and only one Customer.
4. Each Order *contains* one to many Shopping Cart instances; each Shopping Cart instance *is contained in* one and only one Order.



Reference: Valacich, J.S., George, J.F. and Valacich, J.S., 2017. Modern systems analysis and design. Boston: Pearson.

OOA Summary

- During analysis and the early stage of design, two tasks are of importance:
  - **Identify the classes** that form the vocabular of the problem domain
  - **Create object models** that reflect the requirements of the problem
- Object-oriented analysis common approaches
  - Textual/Use Case Analysis
  - Classical Analysis
  - Behavior analysis
  - CRC (Class-Responsibility-Collaboration) Cards

  - Structured analysis
    - Process Modelling (DFDs - Data Flow Diagrams)
    - Entity-Relationship Modelling (ERDs - Entity-Relationship Diagrams)

OOA Summary

- Textual analysis is used to identify potential objects, attributes, operations, and relationships by reviewing the use cases and their associated descriptions.
- Classical analysis derives classes and objects from the principle of classical categorization that focus on tangible things in the problem domain.
- Behavior analysis focuses on dynamic behavior as the primary sources of classes and objects that forms classes based on groups of objects that have similar behavior.
- CRC cards analysis focuses on the assignment of responsibilities and the identification of collaborations of classes.
- Data Flow Diagrams are used for modelling data and processes.
- DFDs show the data movement and transformation in the system.
- DFDs organize the level of details in hierarchies.
  - Context diagram represents the system scope and its external dependencies.
  - Diagram 0 shows the major processes, data stores, external entities and data flows in the system.
  - Low-level diagrams (Diagram 1, 2, etc.) show additional details of the system.
- Conceptual data modelling is based on certain constructs about the structure, not use, of data.

Designing Objects with Responsibilities -- The GRASP Principles

"Think of software objects as similar to people with responsibilities who collaborate with other people to get work done."

--Wirfs-Brock, R. and McKean, A., 2003. Object design: roles, responsibilities, and collaborations. Addison-Wesley Professional.

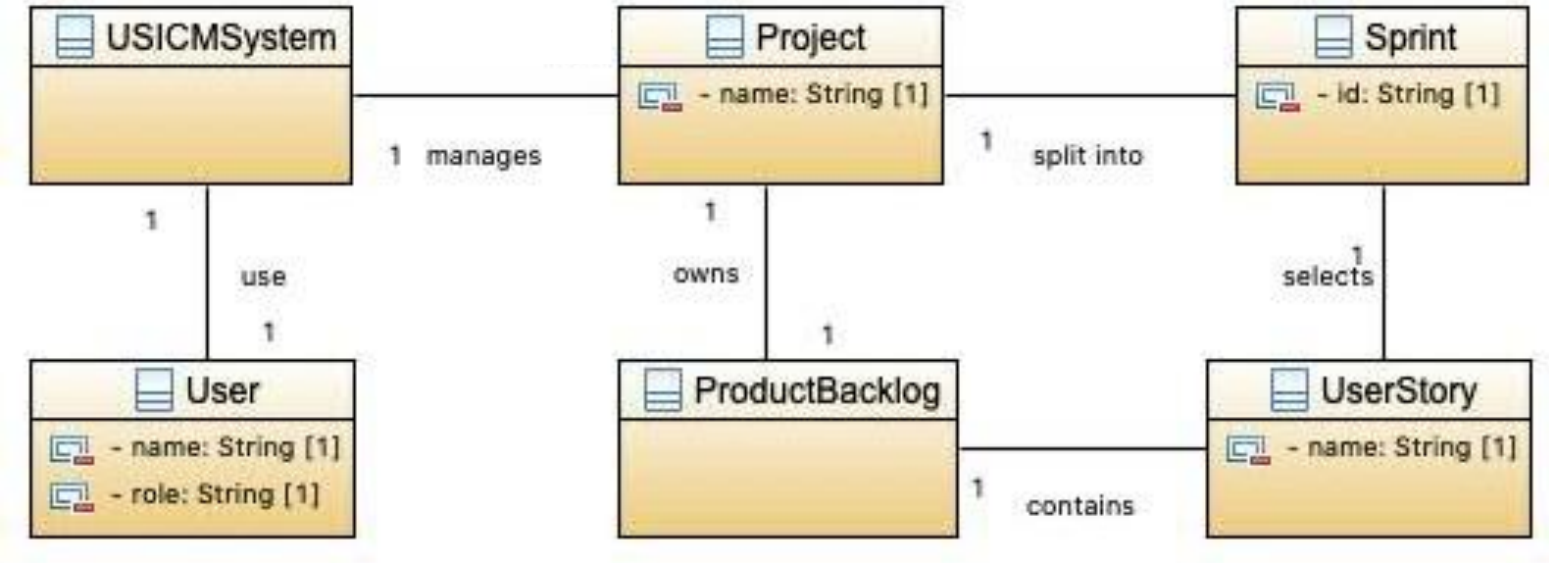
- The GRASP (General Responsibility Assignment Software Pattern or Principles) is a learning aid for OO Design with responsibilities.

GRASP		
Creator	**Low Coupling**	**Indirection**
Information Expert	**Polymorphism**	**Pure Fabrication**
Controller	**High Cohesion**	**Protected Variations**

Domain Model

In software engineering, a domain model is a conceptual model of the domain that incorporates both behaviour and data.

> ***"A domain model is a visual representation of conceptual classes or real-situation objects in a domain."***
>
> *--Martin, J. and Odell, J.J., 1997. Object-oriented methods (UML ed.,) a foundation. Prentice-Hall, Inc..*



A domain model provides a conceptual perspective:

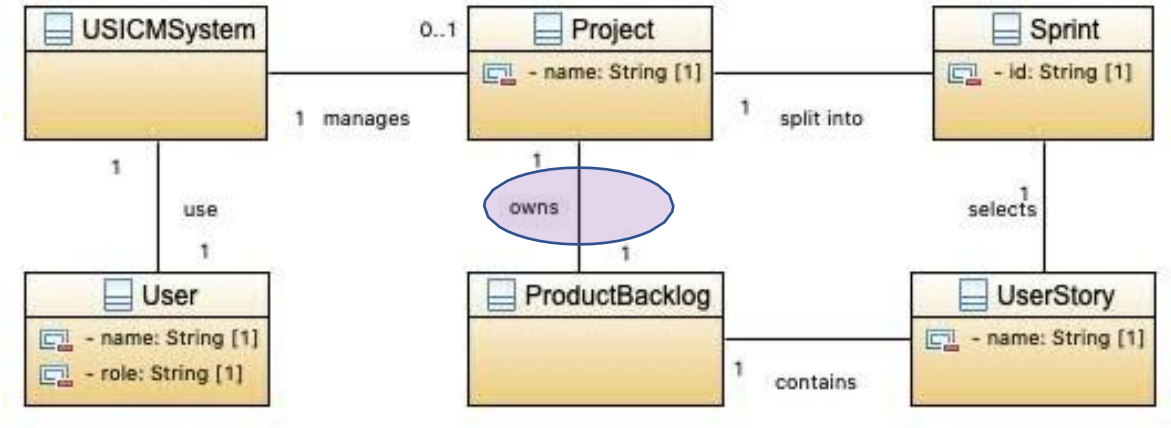
- Domain objects
- Conceptual classes
- Relationships between conceptual classes
- Attributes of conceptual classes

Domain Model == Conceptual Model == Domain Object Model == Analysis Object Model

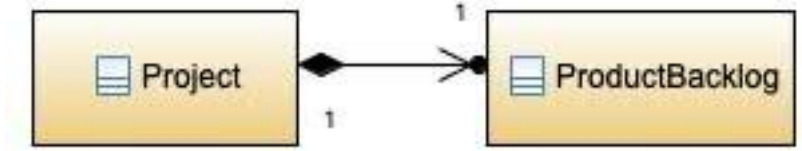
GRASP – Creator

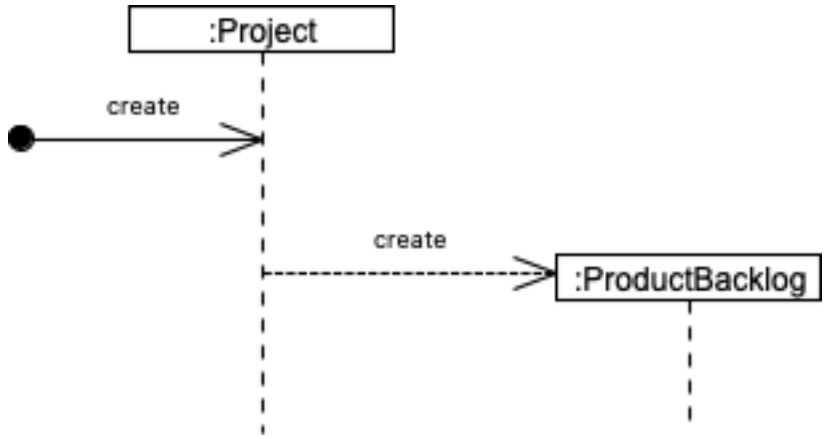
Creator refers to the object that instantiates a class.

Problem:	**Who create an object A?**
Solution:	Assign class B the responsibility to create an instance of class A if one of following is true: • B contains or compositely aggregates A. • B records A. • B closely uses A. • B has the initializing data for A.



- Who should create an object of the "*ProductBacklog*"?

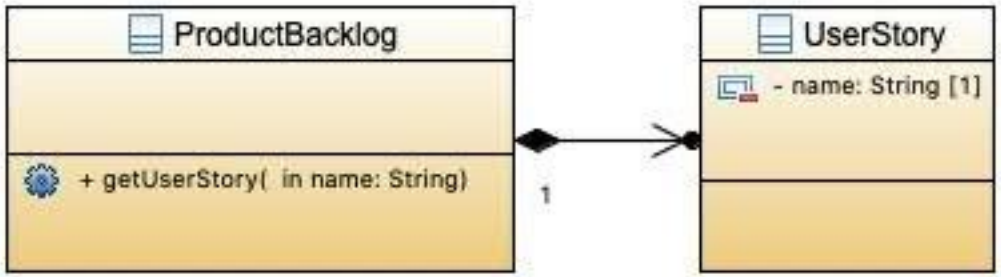


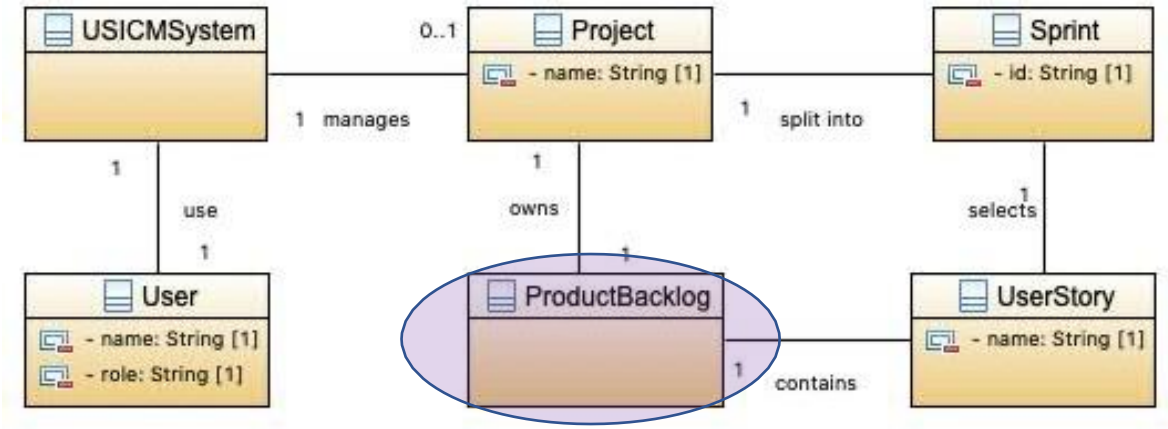


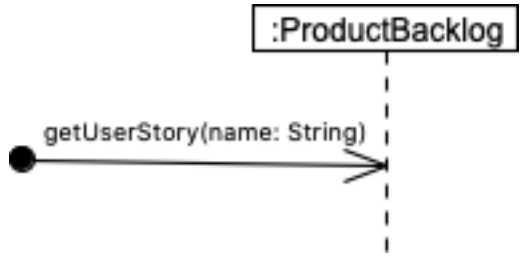
GRASP – Information Expert

Problem:	**What is a basic principle by which to assign responsibilities to objects?**
Solution:	Assign a responsibility to the class that has the information needed to fulfill it.

- Who should take the responsibility for *"getUserStory(name: String)"*?





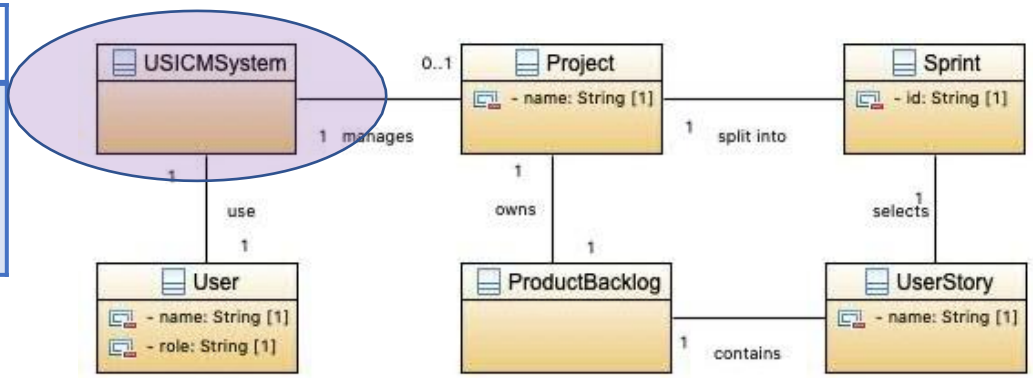


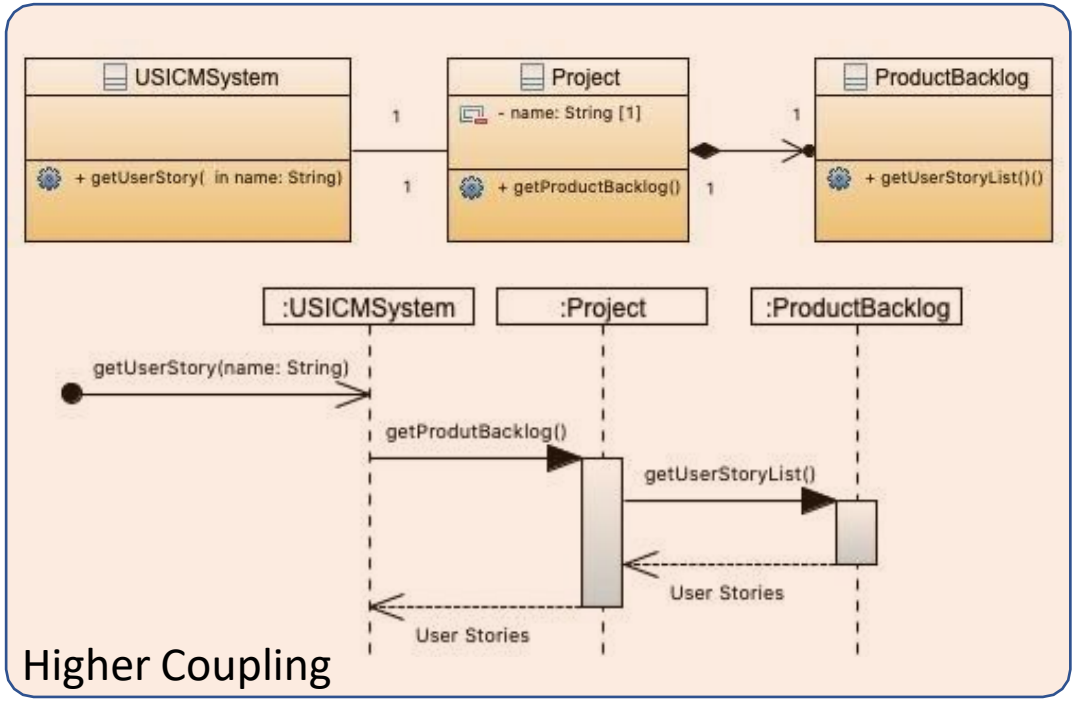
GRASP – Low Coupling

***Coupling** is a measure of how strongly one element is connected to, has knowledge of, or depends on other elements.*

Problem:	**How to minimize the impact of change?**
Solution:	Assign responsibilities so that unnecessary coupling remains low. Use this principle to evaluate alternatives.

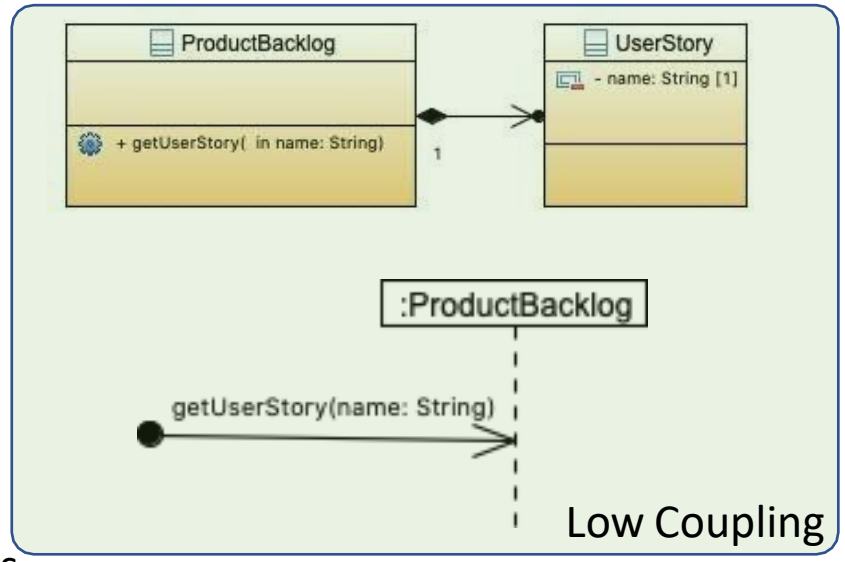
- Who should take the responsibility for *"getUserStory(name: String)"*?





Higher Coupling

VS.



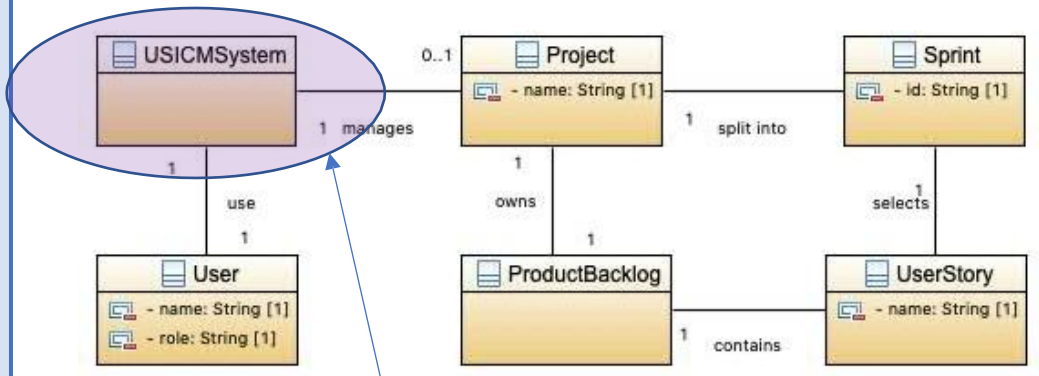
Low Coupling

Sequence Diagrams

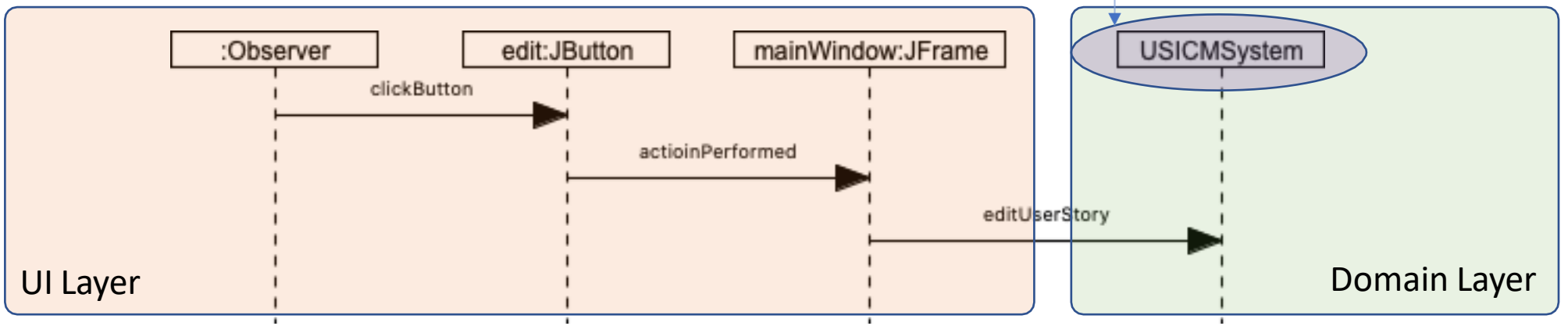
GRASP – Controller

Problem:	**What first object beyond the UI layer receives and coordinates a system operation?**
Solution:	Assign the responsibility to an object representing one of following choices: • Represents the overall system, a “root object”, a device that the software is running within, or a major subsystem (variations of a façade controller). • Represents a use case scenario within which the system operation occurs (a use case or session controller).

- Who should handle the *events* or *actions* triggered by the users at the UI layer?



Treated as a ”controller/coordinator/session/handler”.



The Model-View Separation Principle

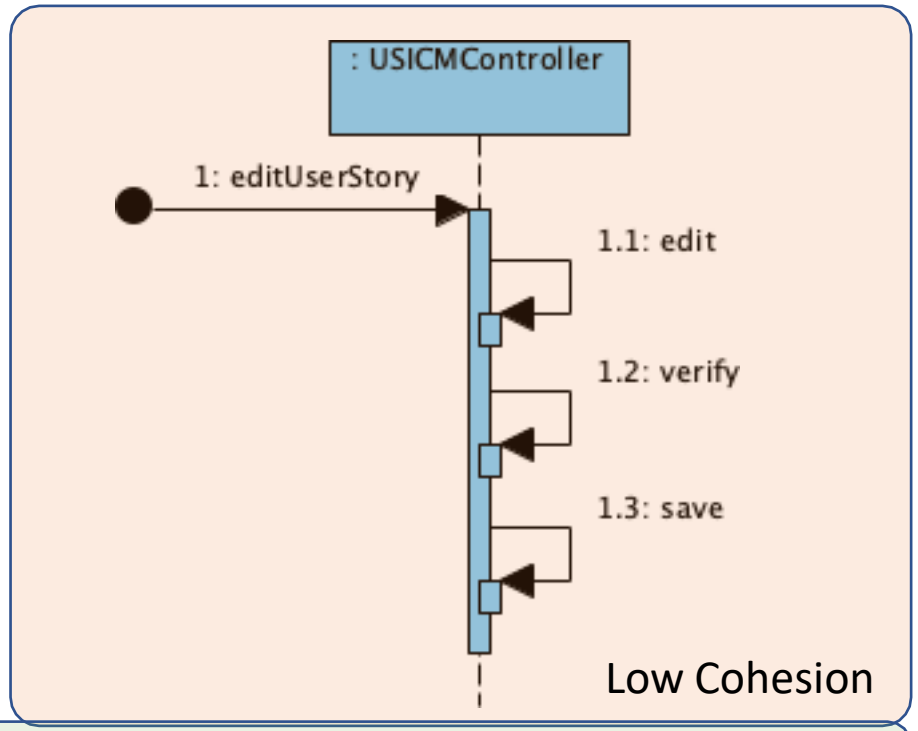
- What kind of visibility should other packages have to the UI layer?
  - Don’t connect or couple non-UI objects directly to UI objects.
    - E.g., don’t let the ***Project*** object (a non-UI domain object) have a reference to a Java *JFrame* window object.
  - Don’t put application logic in the UI object methods.
    - E.g., don’t implement a *getUserStory( )* in a *JFrame* window object.

- How should non-window classes communicate with windows?
  - Use the ***Observer*** pattern, where the domain objects send messages to the UI objects.

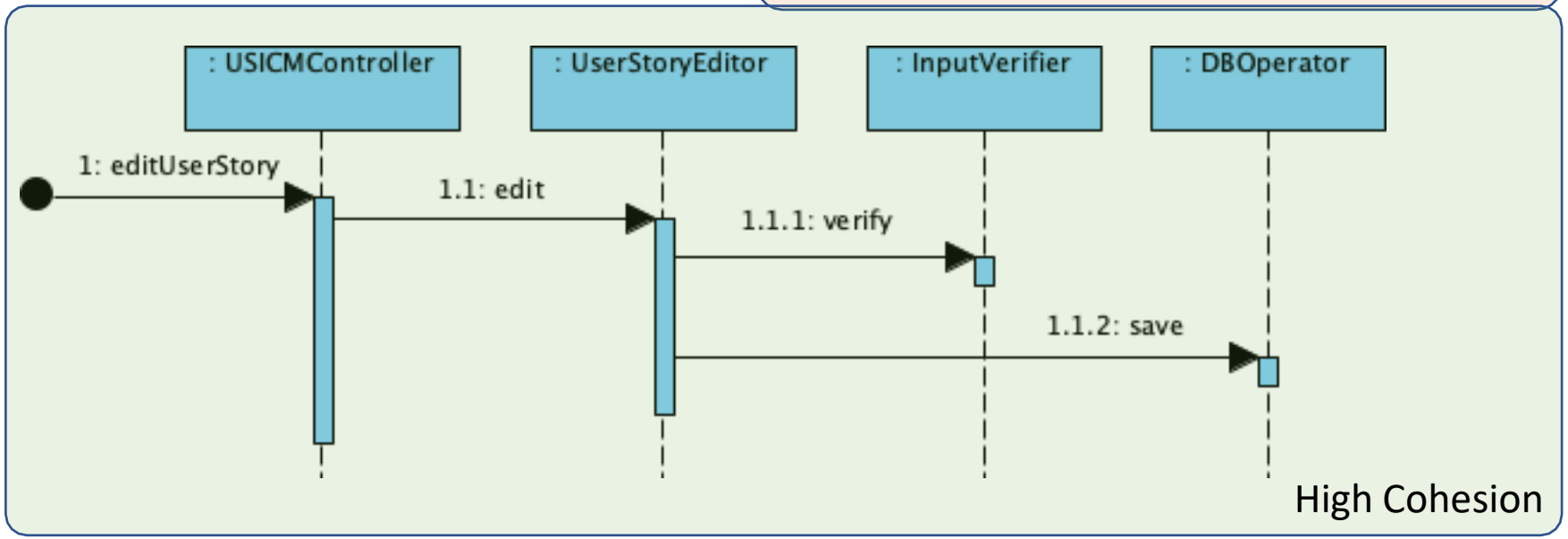
- The principle allows the domain classes encapsulate the information and behavior related to application logic; the window classes are only responsible for input, output, and GUI events, but don’t maintain application data or directly provide application logic.

GRASP – High Cohesion

Problem:	**How to keep objects focused, understandable, and manageable, and as a side effect, support Low Coupling?**
Solution:	Assign responsibility so that cohesion remain high. Use this to evaluate alternatives.

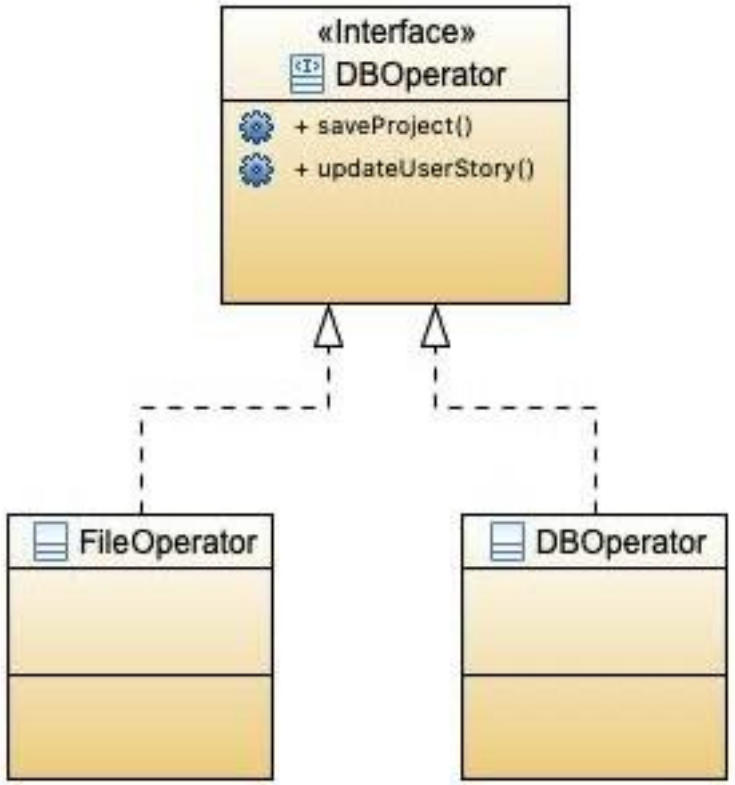


Cohesion measures how functionally related the operations of a software element are, and how much work a software element is doing.



GRASP – Polymorphism

Problem:	• **How handle alternatives based on type?** • **How to create pluggable software components?**
Solution:	When related alternatives or behaviors vary by type, assign responsibility for the behavior using polymorphic operations to the types for which the behavior varies.
Benefits:	• Extensions required for new variations are easy to add. • New implementations can be introduced without affecting clients.



Polymorphism (one of the uses): giving the same name to services in different object when the services are similar or related.

GRASP – Pure Fabrication, Indirection and Protected Variations

	Pure Fabrication	Indirection	Protected Variations
Problem:	What object should have the responsibility, when you don't want to violate High Cohesion and Low Coupling, or other goals, but solutions offered are not appropriate?	Where to assign a responsibility, to avoid direct coupling between two or more things? How to de-couple objects so that low coupling is supported, and reuse potential remains higher?	How to design objects, subsystems, and systems so that the variations or instability in these elements does not have an undesirable impact on other elements?
Solution:	Assign a highly cohesive set of responsibilities to an artificial or convenience class that does not represent a problem domain concept, to support high cohesion, low coupling and reuse. Such a class is a ***fabrication*** of the imagination.	Assign the responsibility to an intermediate object to mediate between other components or services so that thy are not directly coupled.	Identify points of predicted variation or instability; assign responsibilities to create a stable interface (a broad sense of an access view) around them.
Associated Design Patterns:	• Virtually all design patterns	• Adapter • Bridge • Façade • Observer • Mediator	• Most design principles (polymorphism, interfaces, data encapsulation) and most of the design patterns.

The SOLID Principles of OO Design

- SRP: Single Responsibility Principle
- OCP: Open/Close Principle
- LSP: Liskov Substitution Principle
-
- ISP: Interface Segregation Principle
- DIP: Dependency Inversion Principle

”The goal of the principles is the creation of mid-level software structures that tolerate change, are easy to understand, and are the basis of components that can be used in many software systems.”

--Martin, R.C., 2018. Clean architecture: a craftsman's guide to software structure and design. Prentice Hall.

Single Responsibility Principle

"A class should have only a single reason to change."

-- Weisfeld, M., 2008. The object-oriented thought process. Pearson Education.

"A module should be responsible to one, and only one, actor."

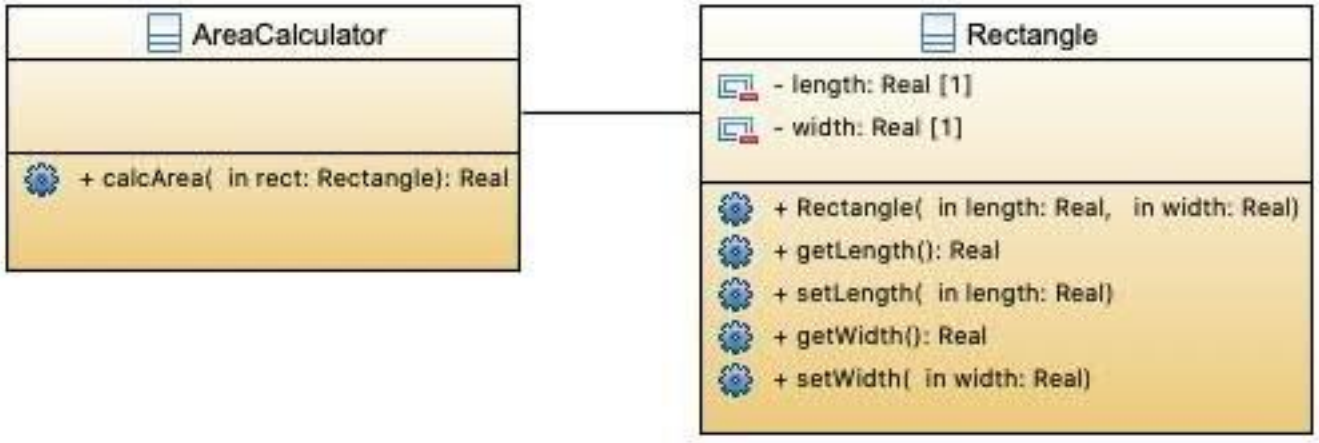
--Martin, R.C., 2018. Clean architecture: a craftsman's guide to software structure and design. Prentice Hall.

- Each class and module in a program should focus on single task.
- Every module or class should have responsibility over a single part of the functionality provided by the software.
- High Cohesion

Open/Close Principle (1)

Modules should be both open (for extension; adaptable) and closed (the modules is closed to modification in ways that affect clients).

-- Meyer, B. 1988. Object-Oriented Software Construction, first edition. Englewood Cliffs, NJ.: Prentice-Hall.

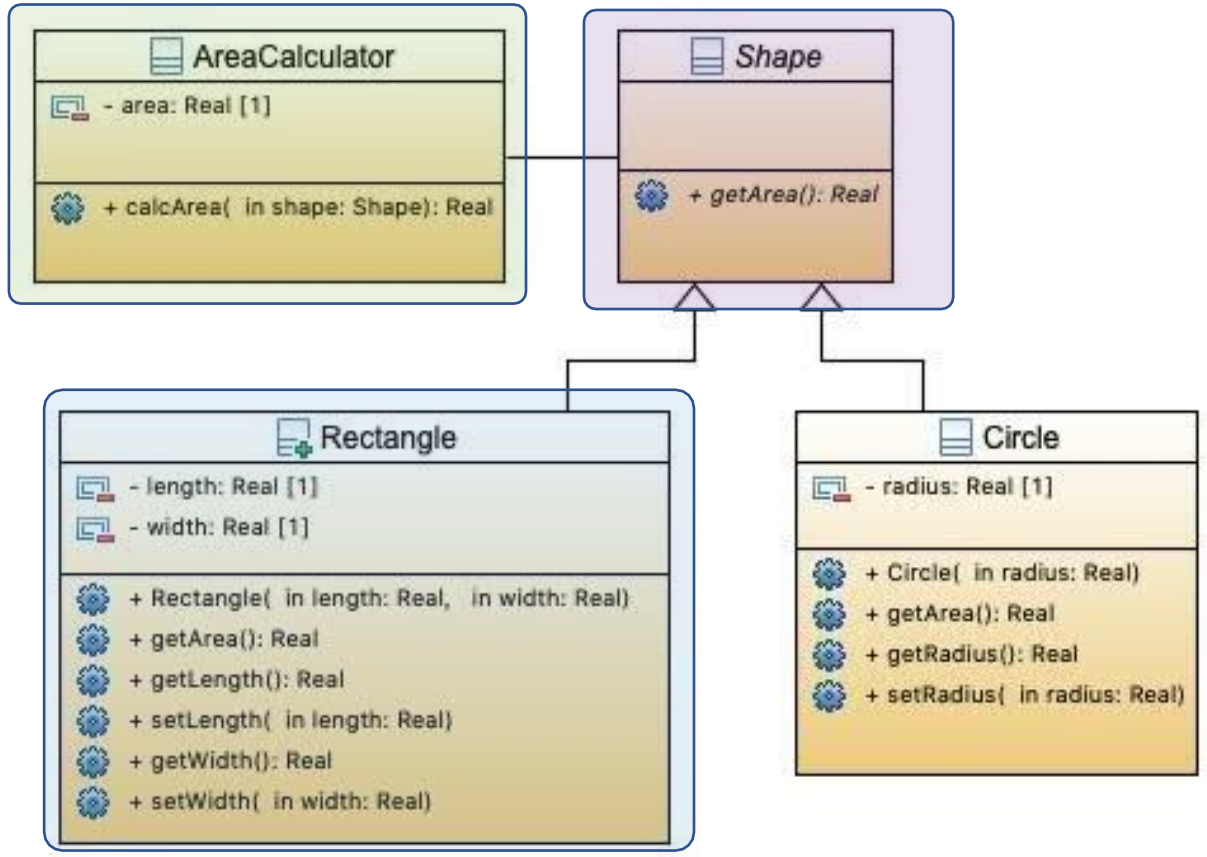



```
public class ClosedButNotOpen {
        public static void main(String[ ] args) {
                Rectangle rectangle = new Rectangle(5, 9);
                AreaCalculator ac = new AreaCalculator( );
                System.out.println(ac.calcArea(rectangle));
        }
}
```

```
public class AreaCalculator {
        public double calcArea(Rectangle rectangle) {
                return (rectangle.getLength() * rectangle.getWidth());
        }
}
```

```
public class Rectangle {
        private double length;
        private double width;
        public Rectangle(double length, double width) {
                this.length = length;
                this.width = width;
        }
        public double getLength() {
                return length;
        }
        public void setLength(double length) {
                this.length = length;
        }
        public double getWidth() {
                return width;
        }
        public void setWidth(double width) {
                this.width = width;
        }
}
```

Open/Close Principle (2)




```
public class OpenClose {
        public static void main(String[] args) {
                AreaCalculator ac = new AreaCalculator();
                Rectangle rectangle = new Rectangle(5, 9);
                System.out.println(ac.calcArea(rectangle));
                Circle circle = new Circle(5);
                System.out.println(ac.calcArea(circle));
        }
}
```

```
public class AreaCalculator {
        private double area;
        public double calcArea(Shape shape) {
                area = shape.getArea();
                return area;
        }
}
```

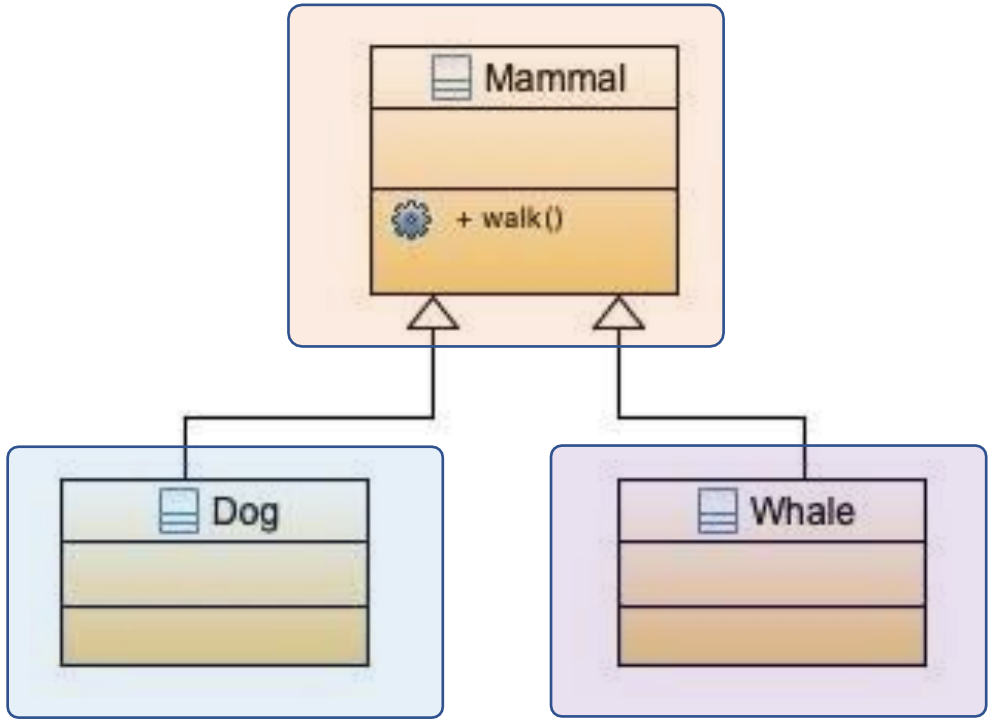
```
public abstract class Shape {
    public abstract double getArea( );
}
```

```
public class Rectangle extends Shape {
        private double length;
        private double width;
        public Rectangle(double length, double width) {
            this.length = length;
            this.width = width; }
        public double getLength( ) {
            return length;  }
        public void setLength(double length) {
            this.length = length; }
        public double getWidth( ) {
            return width; }
        public void setWidth(double width) {
            this.width = width; }
        @Override
        public double getArea( ) {
            return (length * width);  }
}
```

Liskov Substitution Principle (1)

- The ability to replace an instance of a superclass with an instance of one of its child classes.

> ***"If for each object o_1 of type S there is an object o_2 of type T such that for all program P defined in terms of T, the behaviour of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T."***
>
> *--Liskov, B., 1987. Data abstraction and hierarchy. OOPSLA'87.*




```
public class Mammal {
        public String walk( ) {
            return "I can walk";
        }
```

```
public class Dog extends Mammal {

}
```

```
public class Whale extends Mammal {

}
```

```
public class Liskov {
        public static void main(String[ ] args) {
        Mammal mammal = new Mammal( );
        System.out.println(mammal.walk( ) );
        Dog dog = new Dog();
        System.out.println(dog.walk( ));
        Whale whale = new Whale( );
        System.out.println(whale.walk( ));
        }
}
```

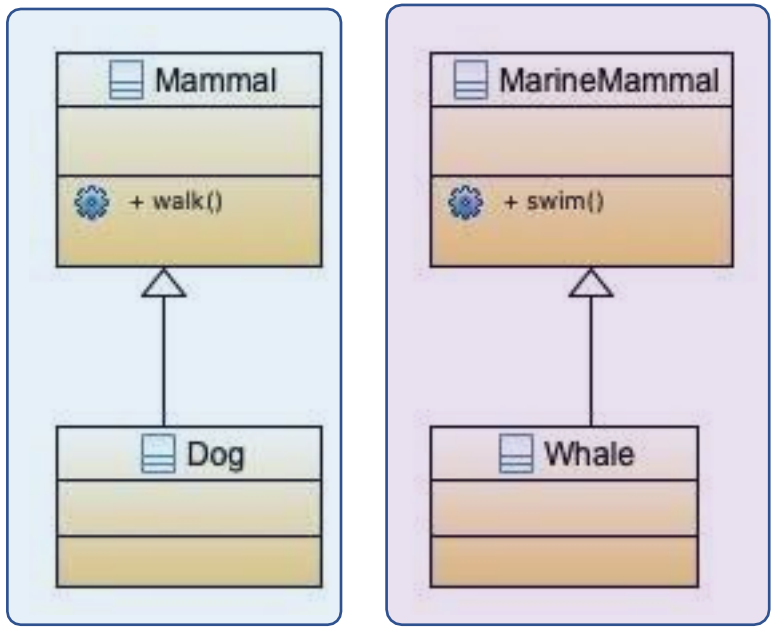
Can whale walk?

I can walk
I can walk
I can walk

Outputs

Liskov Substitution Principle (2)

- The ability to replace an instance of a superclass with an instance of one of its child classes.




```
public class Mammal {
        public String walk( ) {
            return "I can walk";
}

public class Dog extends Mammal {

}
```

```
public class MarineMammal {
        public String swim( ) {
            return "I can swim";
}

public class Whale extends MarineMammal {

}
```

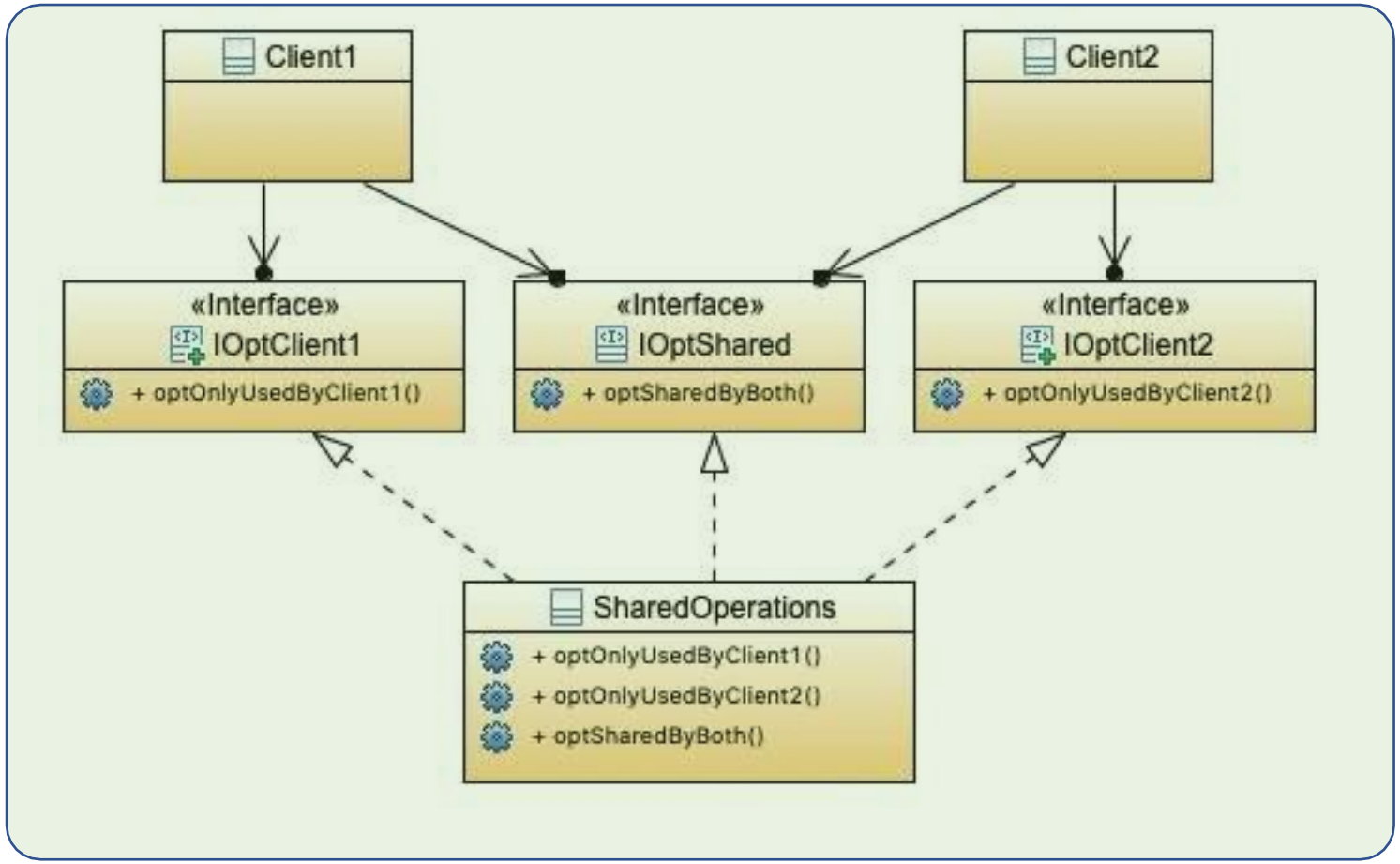
```
public class Liskov {
        public static void main(String[ ] args) {
        Mammal mammal = new Mammal( );
        System.out.println(mammal.walk( ) );
        Dog dog = new Dog();
        System.out.println(dog.walk( ));

        MarineMammal marineMammal = new MarineMammal( );
        System.out.println(marineMammal.swim( ) );
        Whale whale = new Whale( );
        System.out.println(whale.swim( ));
        }
}
```

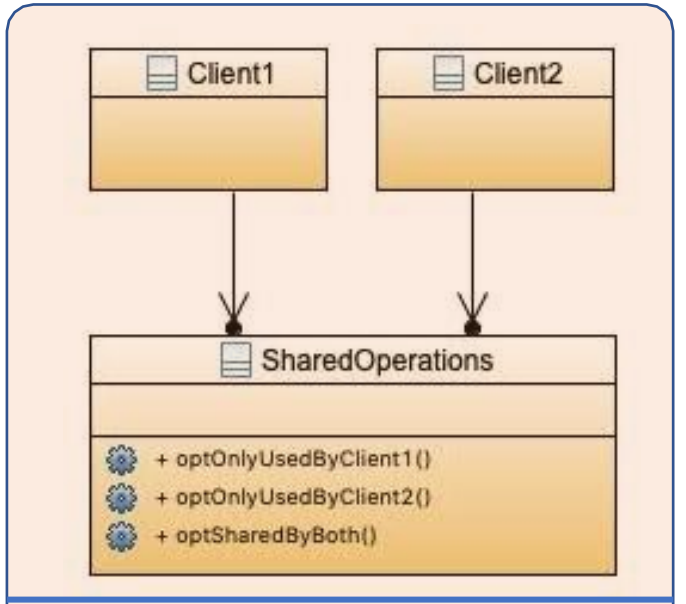
Outputs

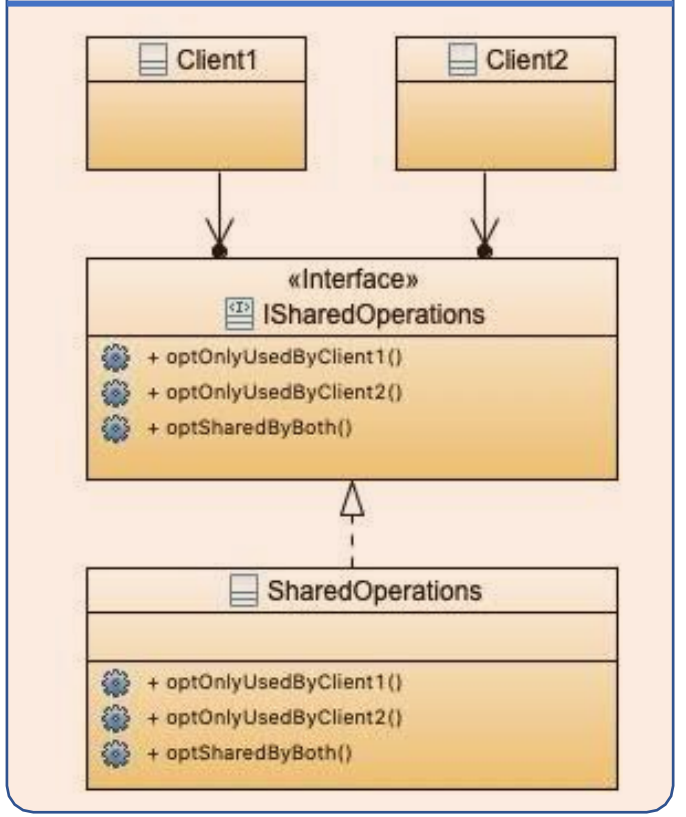
```
I can walk
I can walk
I can swim
I can swim
```

Interface Segregation Principle

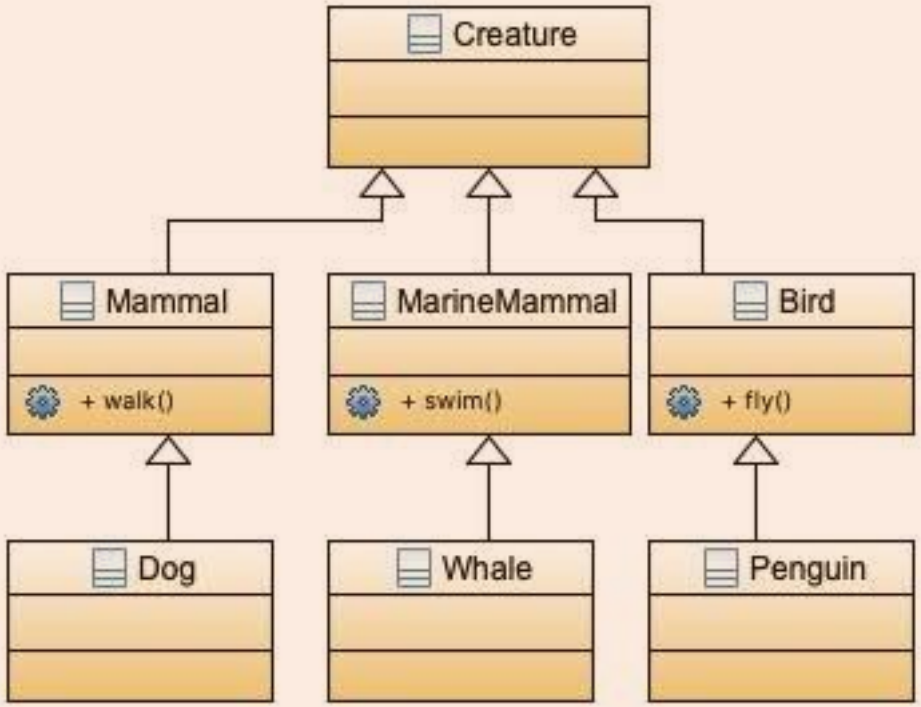
"Clients should not be forced to depend upon interfaces that they do not use."





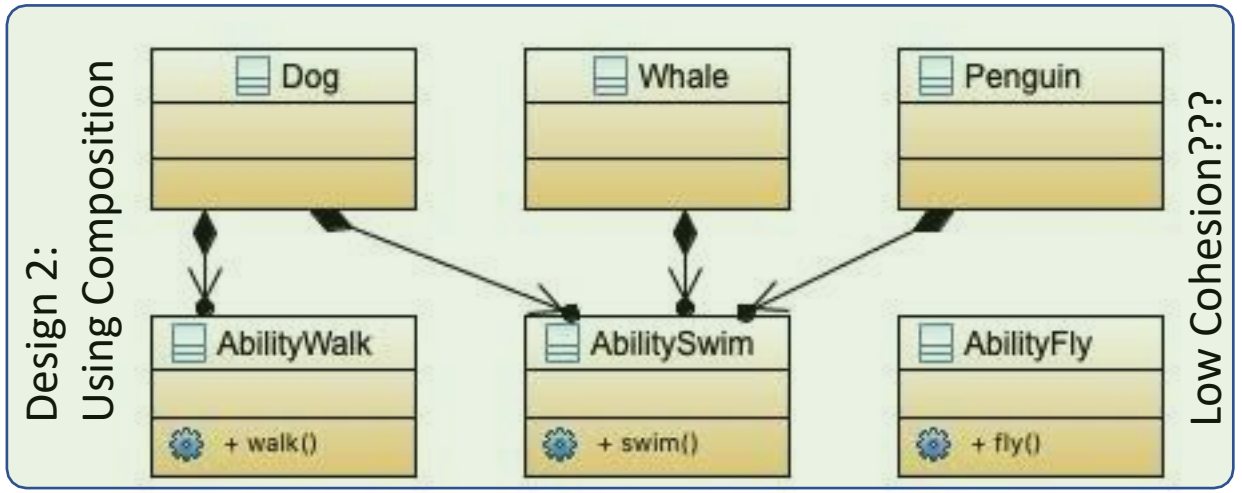


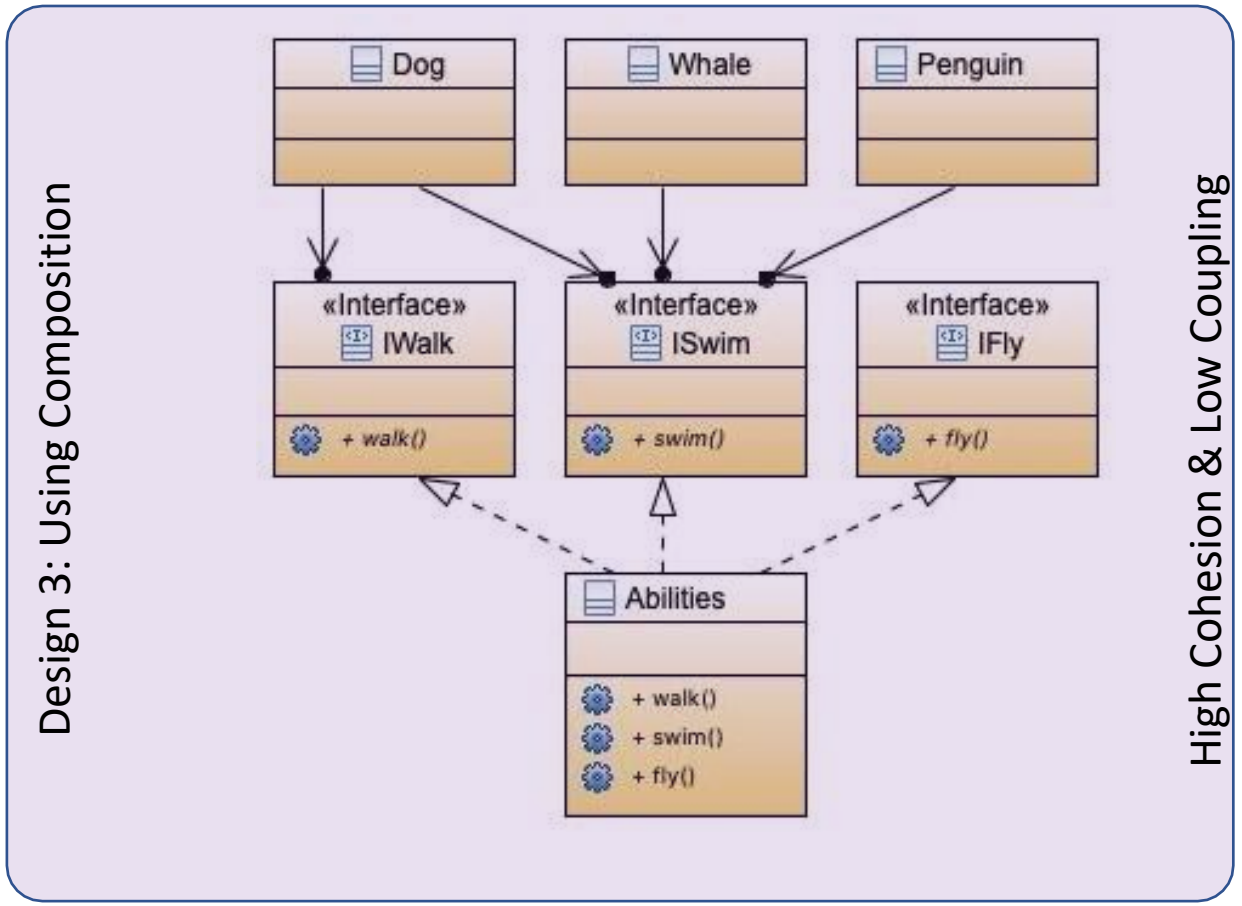
Composition vs. Inheritance



Design 1: Using inheritance

- technically correct, but logically incorrect
  - E.g., can a penguin fly? Can a dog swim?
- hard to model complex relationships



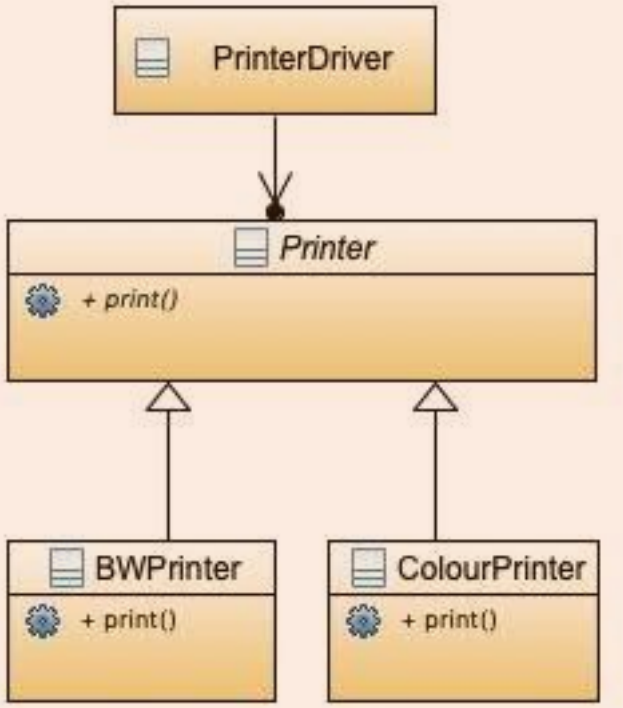


Dependency Inversion Principle

"Code should depend on abstractions."
-- Weisfeld, M., 2008. The object-oriented thought process. Pearson Education.

"Abstractions should not depend upon details. Details should depend upon abstractions"

- *Every change to an abstract interface corresponds to a change to its concrete implementations. Conversely, changes to concrete implementations do not always require changes to the interfaces that they implementation.*




```
public class BWPrinter extends Printer {
        public void print() {
                System.out.println("BW");
        }
}
```

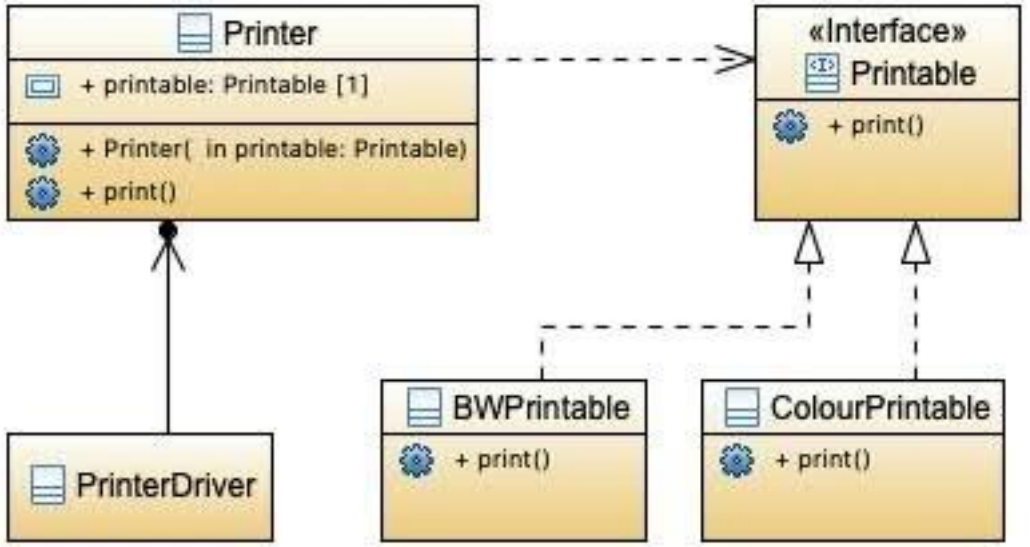
```
public class ColourPrinter extends Printer {
        public void print() {
                System.out.println("Colour");
        }
}
```

```
public abstract class Printer {
        public abstract void print();
}
```

```
public class PrinterDriver {
        public static void main(String[ ] args) {
                Printer cp = new ColourPrinter( );
                cp.print();
                Printer bwp = new BWPrinter( );
                bwp.print();
        }
}
```

Inheritance creates strong dependencies (high-coupling).

DIP – Dependency Injection




```
public class Printer {
        Printable printable;
        public Printer(Printable printable) {
                this.printable = printable;
        }
        public void print() {
                this.printable.print();
        }
}
```

```
public interface Printable {
        public void print();
}
```

```
public class PrinterDriver {
        public static void main(String[ ] args) {
                Printer cp = new Printer(new ColourPrintable( ));
                cp.print();

                Printer bwp = new Printer(new BWPrintable( ));
                bwp.print();
        }
}
```

```
public class BWPrintable implements Printable {
        public void print() {
                System.out.println("BW");
        }
}
```

```
public class ColourPrintable implements Printable {
        public void print() {
                System.out.println("Colour");
        }
}
```

Summary

- The GRASP principles can be used as a tool to help master the basics of OOD and understanding responsibility assignment in object design.
- GRASP follows the idea of responsibility-driven design, thinking about how to assign responsibilities to collaborating objects.
- GRASP and SOLID largely overlap.

Favour composition over inheritance;
Program to Interface, not Implementation.

Practice

In Object-Oriented design, modelling complex relationships between entities is hard. The following class diagram models creatures using inheritance. It is technically correct, but logically incorrect, as a bat is a mammal but can fly, and a penguin is a bird, but cannot fly. Redesign the class diagram using class composition so that it is logically and technically correct.