UNIT II STATIC UML DIAGRAMS

Class Diagram— Elaboration – Domain Model – Finding conceptual classes and description classes – Associations – Attributes – Domain model refinement – Finding conceptual class Hierarchies – Aggregation and Composition - Relationship between sequence diagrams and use cases – When to use Class Diagrams

Class Diagram

Class diagram is a static diagram. It represents the static view of an application. Class diagram is not only used for visualizing, describing, and documenting different aspects of a system but also for constructing executable code of the software application.

Class diagram describes the attributes and operations of a class and also the constraints imposed on the system. The class diagrams are widely used in the modelling of object oriented systems because they are the only UML diagrams, which can be mapped directly with object-oriented languages.

Class diagram shows a collection of classes, interfaces, associations, collaborations, and constraints. It is also known as a structural diagram.

Types of Classes

- **Ones found during analysis:**
  - people, places, events, and things about which the system will capture information
  - ones found in application domain
- **Ones found during design**
  - specific objects like windows and forms that are used to build the system

Purpose of Class Diagrams

The purpose of class diagram is to model the static view of an application. Class diagrams are the only diagrams which can be directly mapped with object-oriented languages and thus widely used at the time of construction.

UML diagrams like activity diagram, sequence diagram can only give the sequence flow of the application; however class diagram is a bit different. It is the most popular UML diagram in the coder community.

The purpose of the class diagram can be summarized as −

- Analysis and design of the static view of an application.
- Describe responsibilities of a system.
- Base for component and deployment diagrams.
- Forward and reverse engineering.

Diagram of one class

The top part contains the name of the class. It is printed in bold and centered, and the first letter is capitalized.
The middle part contains the attributes of the class. They are left-aligned and the first
Letter is lowercase.
The bottom part contains the methods the class can execute. They are also left-aligned and the first letter is lowercase.

Class name in top of box

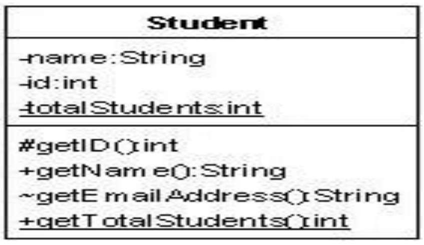
- write <<interface>> on top of interfaces' names
- use italics for an abstract class name

Attributes (optional)

- should include all fields of the object

Operations / methods (optional)

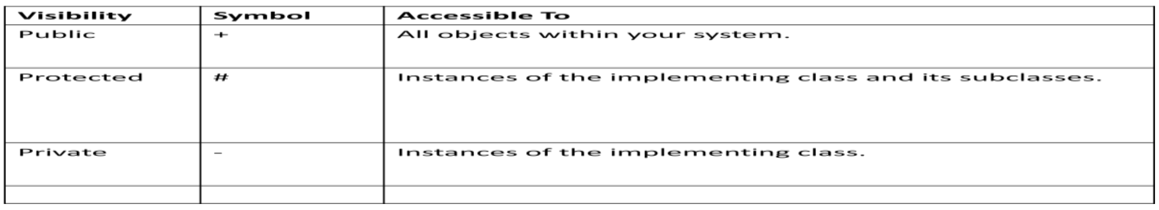
- may omit trivial (get/set) methods
- but don't omit any methods from an interface!
- should not include inherited methods



Attributes in a Class

Properties of the class about which we want to capture information. Represents a piece of information that is relevant to the description of the class within the application domain. Only add attributes that are primitive or atomic types

- Derived attribute
  - attributes that are calculated or derived from other attributes
  - denoted by placing slash (/) before name
- attributes (fields, instance variables)
  - ***visibility name*** **: data *type***



Visibility	Symbol	Accessible To
Public	+	All objects within your system.
Protected	#	Instances of the implementing class and its subclasses.
Private	-	Instances of the implementing class.

Operations in a Class

- Represents the actions or functions that a class can perform
- Describes the actions to which the instances of the class will be capable of responding
- Can be classified as a constructor, query, or update operation
- operations / methods
- ***visibility name* (*parameters*) : *return type***
- visibility: + public

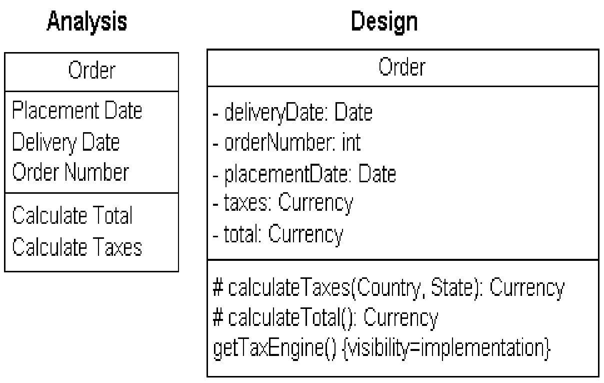
  # protected

  - private

  ~ package (default)
- underline static methods
- parameter types listed as (name: type)

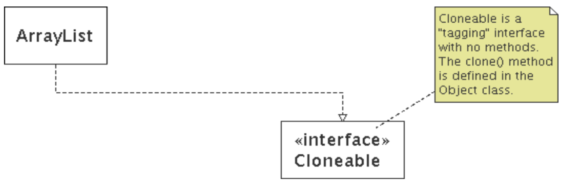
  example:

  + distance(p1: Point, p2: Point): double



Comments

- Represented as a folded note, attached to the appropriate class/method/etc by a dashed line.


ArrayList
Cloneable is a "tagging" interface with no methods. The clone() method is defined in the Object class.
«interface» Cloneable

ELABORATION

Elaboration is an activity in which the information about the requirements is expanded and refined. This information is gained during inception.The elaboration consists of several modeling and refinement tasks. Several user scenarios are created and refined.Each scenario is parsed and various classes are identified. The classes are business entities that are visible to the end user.The attributes and the services (methods) of these classes are defined, and the relationship among these classes is identified.Various UML diagrams are developed during this task.Elaboration is conducted within two are more iterations. Each iteration is timeboxed and executes for two to three weeks.Elaboration is neither a design step nor a phase in which a fully developed model for implementation is prepared.Executable architecture or the architectural baseline of the system is prepared.

ELABORATION – ACTIVITIES

- **Define the architecture.**

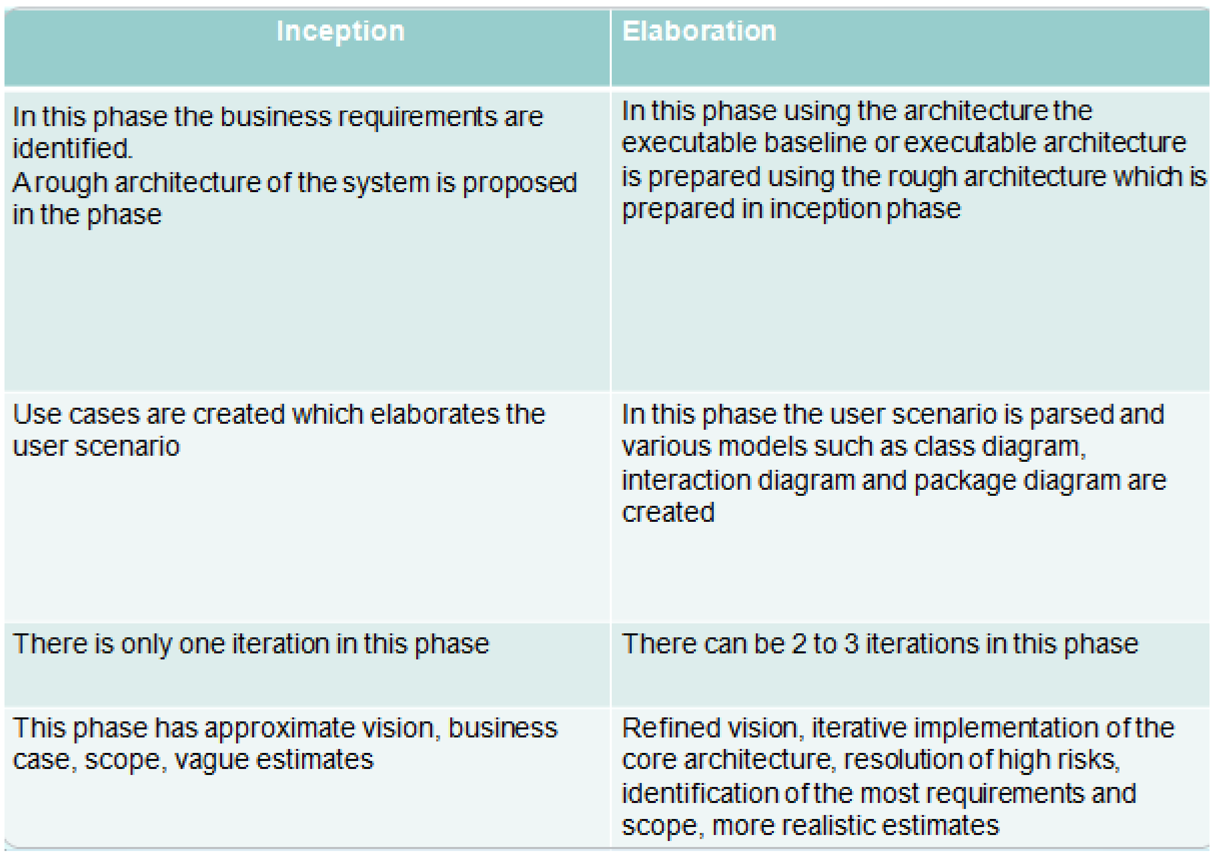
Project plan is defined. The process, infrastructure and development environment are described**.**

- **Validate the architecture.**

- **Baseline the architecture.**

To provide a stable basis for the bulk of the design and implementation effort in the construction phase.

DIFFERENCE BETWEEN INCEPTION AND ELABORATION



Inception	Elaboration
In this phase the business requirements are identified. Arough architecture of the system is proposed in the phase	In this phase using the architecture the executable baseline or executable architecture is prepared using the rough architecture which is prepared in inception phase
Use cases are created which elaborates the user scenario	In this phase the user scenario is parsed and various models such as class diagram, interaction diagram and package diagram are created
There is only one iteration in this phase	There can be 2 to 3 iterations in this phase
This phase has approximate vision, business case, scope, vague estimates	Refined vision, iterative implementation of the core architecture, resolution of high risks, identification of the most requirements and scope, more realistic estimates

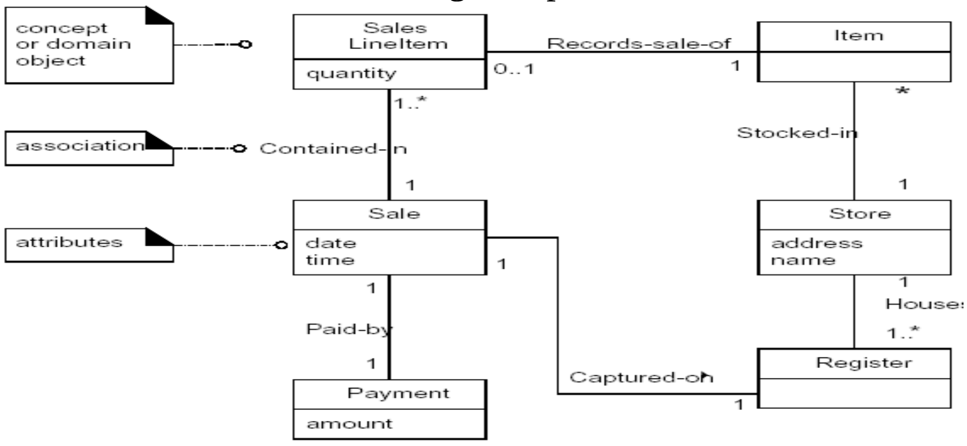
INTRODUCTION TO DOMAIN MODEL

A domain model captures the most important types of objects in the context of the Business. The domain model represents the 'things' that exist or events that transpire in the business environment. A Domain model is the important and classic model in OO analysis. It illustrates noteworthy concepts in a domain. **A Domain Model is a Visual representation of Conceptual Classes or real – situation objects in a domain.**

- Domain models have also been called Conceptual models, domain object models, and analysis object models
- A domain model is illustrated with a set of Class diagrams in which no operations are defined. It provides a conceptual perspective.
- It may show
  - Domain objects or conceptual classes
  - Associations b/w conceptual classes
  - Attributes of conceptual classes

It can act as a source of inspiration for designing some software objects and will be an input to several artifacts explored in the case studies.

- **A domain model is a representation of real-world conceptual classes**
  - not a representation of software components.
  - not a set of diagrams describing software classes,
  - not software objects with responsibilities.
- **A domain model** is a visual representation of conceptual classes or real-world objects in a domain of interest [MO95, Fowler96]
  - They have also been called conceptual models, domain object models, and analysis object models.
- The UP defines a Domain Model as one of the artifacts that may be created in the Business Modeling discipline.



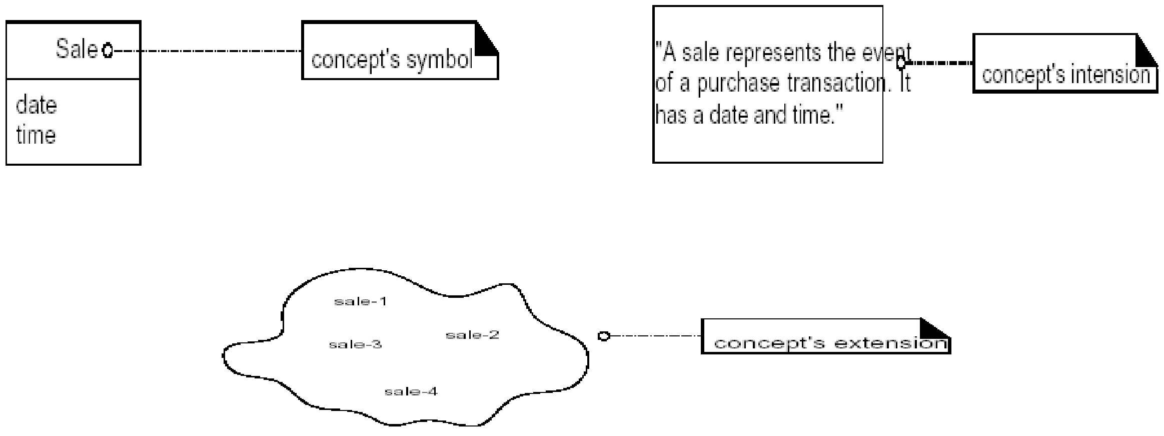
- Software problems can be complex;
    - Decomposition (divide-and-conquer) is a common strategy to deal with this complexity by division of the problem space into comprehensible units.
    - In structured analysis, the dimension of decomposition is by processes or *functions.*
    - However, in object-oriented analysis, the dimension of decomposition is fundamentally by things or entities in the domain.
- A central distinction between object-oriented and structured analysis is: division by conceptual classes (objects) rather than division by functions.
- A primary analysis task is to identify different concepts in the problem domain and document the results in a domain model

Domain Models are not models of software components. The following elements are not suitable in a domain model:

- **Software artifacts, such as a window or a database, unless the domain being modeled is of software concepts, such as a model of graphical user interfaces.**
- **Responsibilities or methods.**

A conceptual class may be considered in terms of its symbol, intension, and extension.

- **Symbol—words or images representing a conceptual class.**
- **Intension—the definition of a conceptual class.**
- **Extension—the set of examples to which the conceptual class applies.**



DOMAIN MODEL AS A VISUAL DICTIONARY

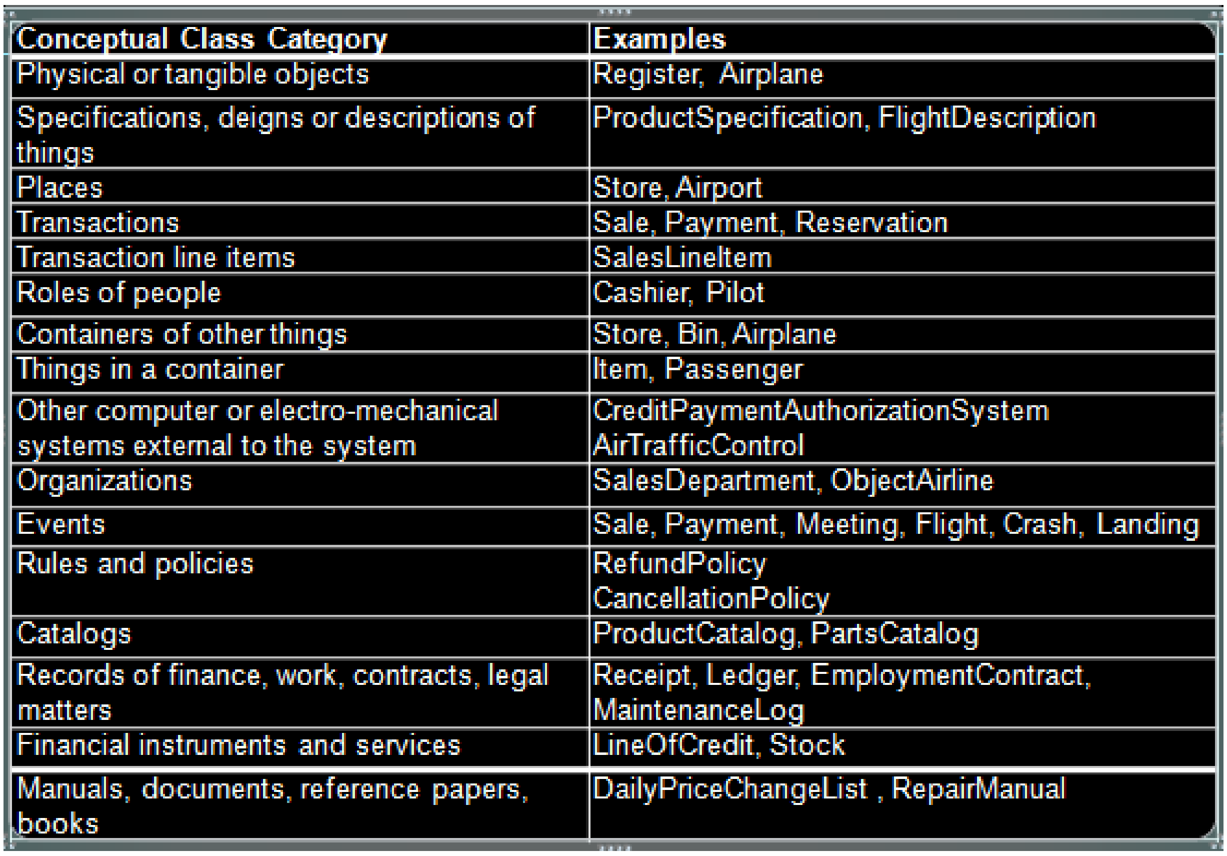
The information in domain model can be expressed in plain text.But the conceptual classes and their relationship with other classes can be expressed using visual language using a domain model. Hence Domain model is referred as visual dictionary.

FINDING CONCEPTUAL CLASSES AND DESCRIPTION CLASSES

- **Strategies to Identify Conceptual Classes.**
    - Use a conceptual class category list.
    - Finding conceptual classes with Noun Phrase Identification

CONCEPTUAL CLASS

A *conceptual class* is a real-world concept or thing; a conceptual or essential perspective. A conceptual class is *not* an *implementation* class, such as a class that can be implemented in an OO language such as Java or C++. That comes much later.



Conceptual Class Category	Examples
Physical or tangible objects	Register, Airplane
Specifications, deigns or descriptions of things	ProductSpecification, FlightDescription
Places	Store, Airport
Transactions	Sale, Payment, Reservation
Transaction line items	SalesLineItem
Roles of people	Cashier, Pilot
Containers of other things	Store, Bin, Airplane
Things in a container	Item, Passenger
Other computer or electro-mechanical systems external to the system	CreditPaymentAuthorizationSystem AirTrafficControl
Organizations	SalesDepartment, ObjectAirline
Events	Sale, Payment, Meeting, Flight, Crash, Landing
Rules and policies	RefundPolicy CancellationPolicy
Catalogs	ProductCatalog, PartsCatalog
Records of finance, work, contracts, legal matters	Receipt, Ledger, EmploymentContract, MaintenanceLog
Financial instruments and services	LineOfCredit, Stock
Manuals, documents, reference papers, books	DailyPriceChangeList , RepairManual

How to Make a Domain Model

1. List the candidate conceptual classes using the Conceptual Class Category List and noun phrase identification techniques related to the current requirements under consideration.
2. Draw them in a domain model.
3. Add the associations necessary to record relationships for which there is a need to preserve some memory.
4. Add the attributes necessary to fulfill the information requirements

On Naming and Modeling Things

Use the vocabulary of the domain when naming conceptual classes and attributes.
For example, if developing a model for a library, name the customer a *"Borrower"* or *"Patron"*—the terms used by the library staff.

A domain model may exclude conceptual classes in the problem domain not pertinent to the requirements.

For example, we may exclude *Pen* and *PaperBag* from our domain model (for the current set of requirements) since they do not have any obvious noteworthy role. The domain model should exclude things *not* in the problem domain under consideration.

SPECIFICATION OR DESCRIPTION CONCEPTUAL CLASSES

Add a specification or description conceptual class (for example, *Product Specification)* when:

- There needs to be a description about an item or service, independent of the current existence of any examples of those items or services.
- Deleting instances of things they describe (for example, *Item)* results in a loss of information that needs to be maintained, due to the incorrect association of information with the deleted thing.
- It reduces redundant or duplicated information.

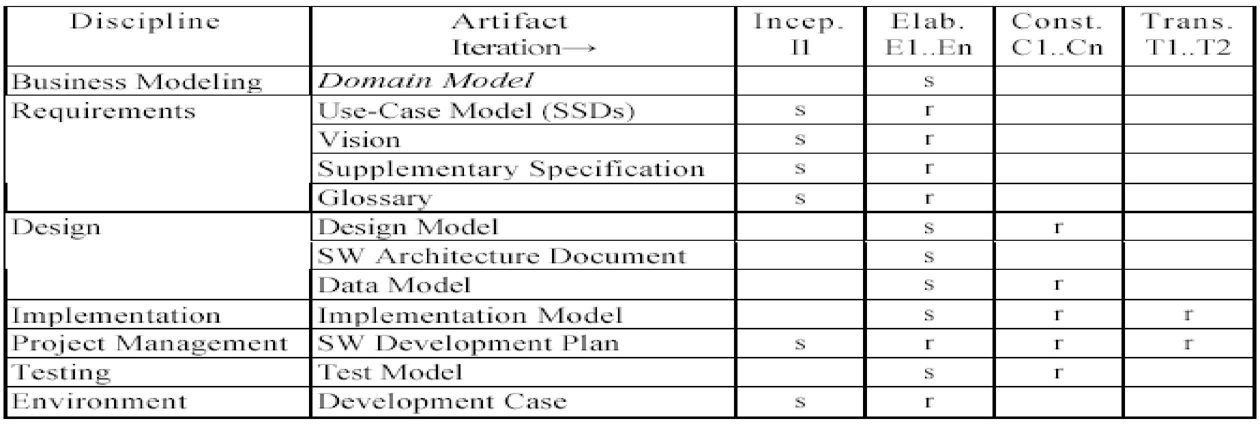
Domain Models within the UP

- **Domain models are not strongly motivated in inception,**

Since inception's purpose is not to do a serious investigation, but rather to decide if the project is worth deeper investigation in an elaboration phase.

- **The Domain Model is primarily created during elaboration iterations,**

When the need is highest to understand the noteworthy concepts and map some to software classes during design work.



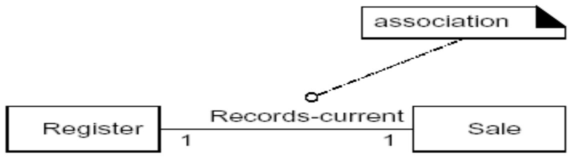
Discipline	Artifact Iteration→	Incep. I1	Elab. E1..En	Const. C1..Cn	Trans. T1..T2
Business Modeling	*Domain Model*		s		
Requirements	Use-Case Model (SSDs)	s	r		
	Vision	s	r		
	Supplementary Specification	s	r		
	Glossary	s	r		
Design	Design Model		s	r	
	SW Architecture Document		s		
	Data Model		s	r	
Implementation	Implementation Model		s	r	r
Project Management	SW Development Plan	s	r	r	r
Testing	Test Model		s	r	
Environment	Development Case	s	r		

Table 10.2 Sample UP artifacts and timing, s - start; r - refine

ASSOCIATIONS

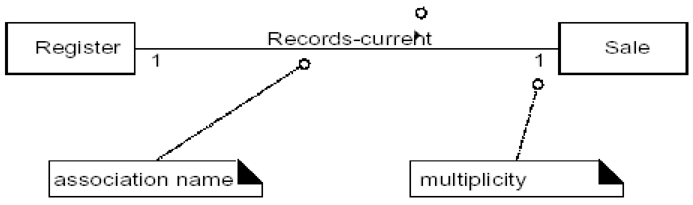
- The UML Class diagram is used to visually describe the problem domain in terms of types of object (classes) related to each other in different ways. **There are three primary inter-object relationships: association, aggregation, and composition.**
- Using the right relationship line is important for placing implicit restrictions on the visibility and propagation of changes to the related classes, matter which play major role in reducing system complexity.
- An association is a relationship between types (or more specifically, instances of those types) that indicates some meaningful and interesting connection
- In the UML associations are defined as **"the semantic relationship between two or more classes that involve connections among their instances**.

- ***Criteria for Useful Associations***
  - Associations for which knowledge of the relationship needs to be preserved for some duration ("need-to-know" associations).
  - Associations derived from the Common Associations List.



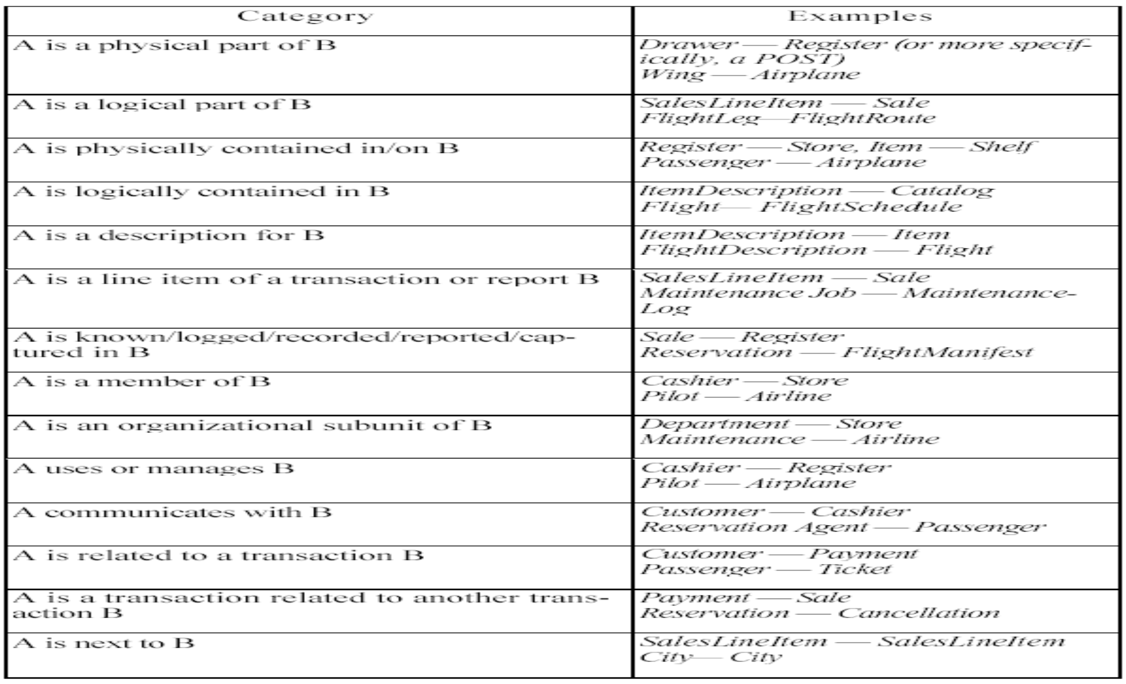
- An association is represented as a line between classes with an association name.
  - The association is inherently bidirectional, meaning that from instances of either class, logical traversal to the other is possible.
  - This traversal is purely abstract; it is not a statement about connections between software entities.

- The ends of an association may contain a multiplicity expression indicating the numerical relationship between instances of the classes.
- An optional "reading direction arrow" indicates the direction to read the association name; it does not indicate direction of visibility or navigation.



FINDING ASSOCIATIONS

- **Start the addition of associations by using the Common Associations List**
  - It contains common categories that are usually worth considering.
- **Here are some high-priority association categories that are invariably useful to include in a domain model:**
  - A is a *physical or logical part* of B.
  - A is *physically or logically contained* in/on B.
  - A is *recorded in* B.



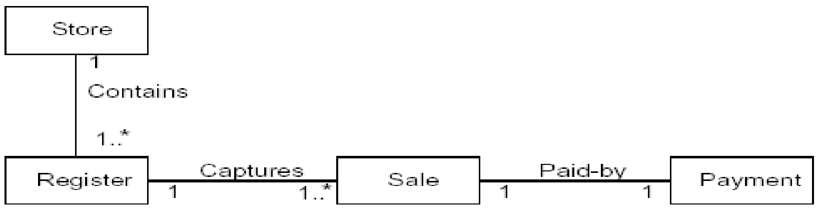
Category	Examples
A is a physical part of B	*Drawer — Register (or more specifically, a POST)* *Wing — Airplane*
A is a logical part of B	*SalesLineItem — Sale* *FlightLeg—FlightRoute*
A is physically contained in/on B	*Register — Store, Item — Shelf* *Passenger — Airplane*
A is logically contained in B	*ItemDescription — Catalog* *Flight— FlightSchedule*
A is a description for B	*ItemDescription — Item* *FlightDescription — Flight*
A is a line item of a transaction or report B	*SalesLineItem — Sale* *Maintenance Job — Maintenance-Log*
A is known/logged/recorded/reported/captured in B	*Sale — Register* *Reservation — FlightManifest*
A is a member of B	*Cashier — Store* *Pilot — Airline*
A is an organizational subunit of B	*Department — Store* *Maintenance — Airline*
A uses or manages B	*Cashier — Register* *Pilot — Airplane*
A communicates with B	*Customer — Cashier* *Reservation Agent — Passenger*
A is related to a transaction B	*Customer — Payment* *Passenger — Ticket*
A is a transaction related to another transaction B	*Payment — Sale* *Reservation — Cancellation*
A is next to B	*SalesLineItem — SalesLineItem* *City— City*

ASSOCIATION GUIDELINES

- Focus on those associations for which knowledge of the relationship needs to be preserved for some duration ("need-to-know" associations).
- It is more important to identify *conceptual classes* than to identify associations.
- Too many associations tend to confuse a domain model rather than illuminate it. Their discovery can be time-consuming, with marginal benefit.
- Avoid showing redundant or derivable associations.

NAMING ASSOCIATIONS

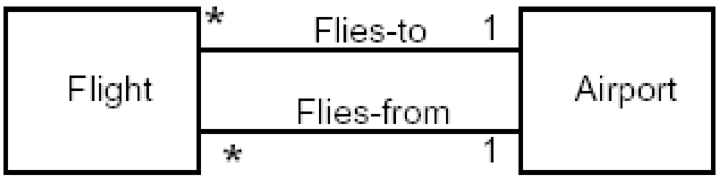
Name an association based on a *TypeName-VerbPhrase-TypeName* format where the verb phrase creates a sequence that is readable and meaningful in the model context.



MULTIPLE ASSOCIATIONS BETWEEN TWO TYPES

Two types may have multiple associations between them;

- During domain modeling, an association is *not* a statement about data flows, instance variables, or object connections in a software solution;
- it is a statement that a relationship is meaningful in a purely conceptual sense—in the real world.



PARTIAL DOMAIN MODEL OF NEXT GEN POS SYSTEM

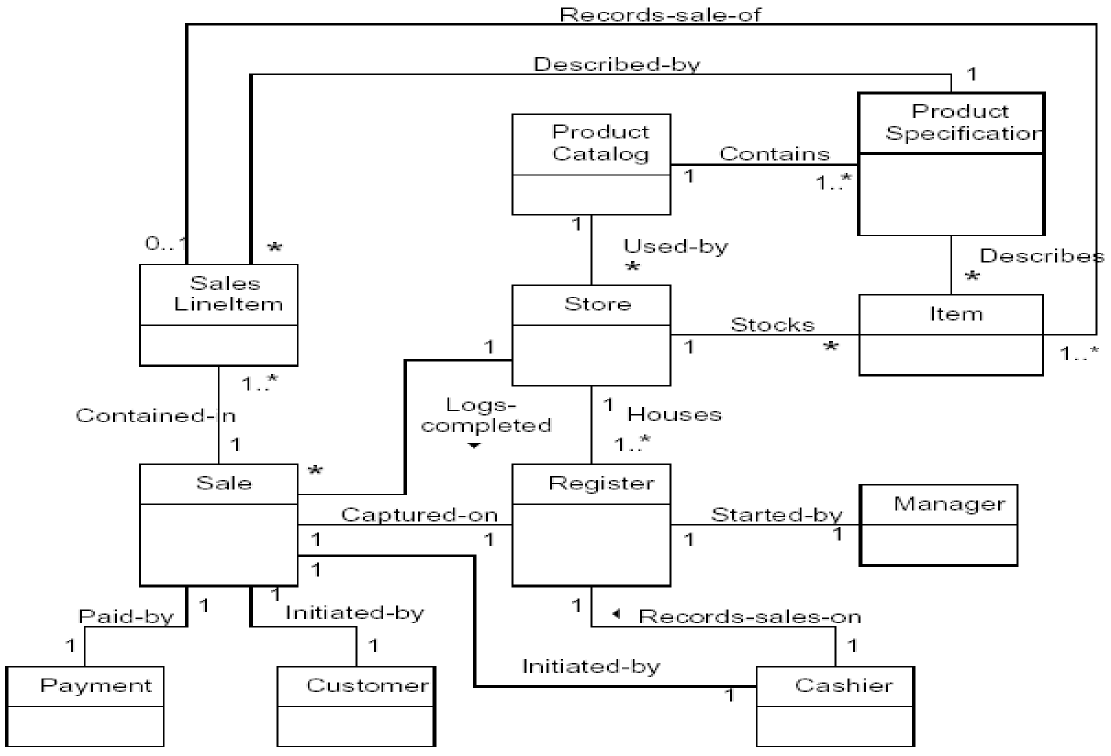
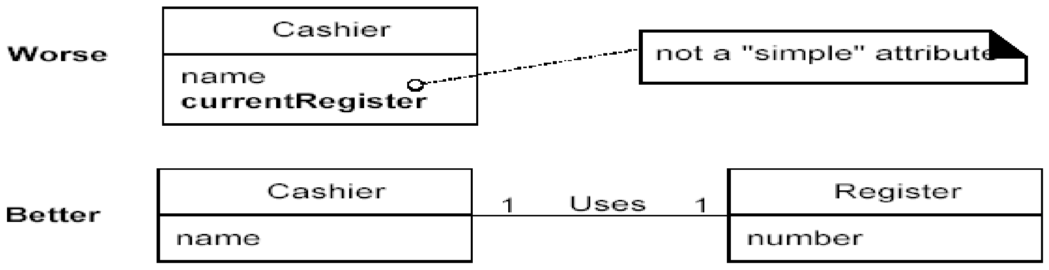


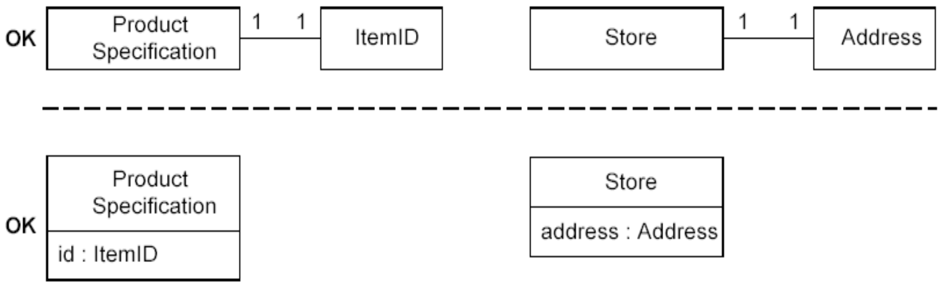
Figure 11.8 A partial domain model.

ADDING ATTRIBUTES

- **Attributes VS. Associations**
  The attributes in a domain model should preferably be simple attributes or data types.

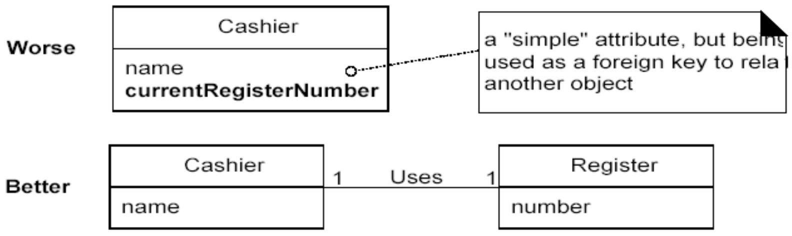


- **Non-primitive Data Type Classes**
  If the attribute class is a data type, it may be shown in the attribute box.



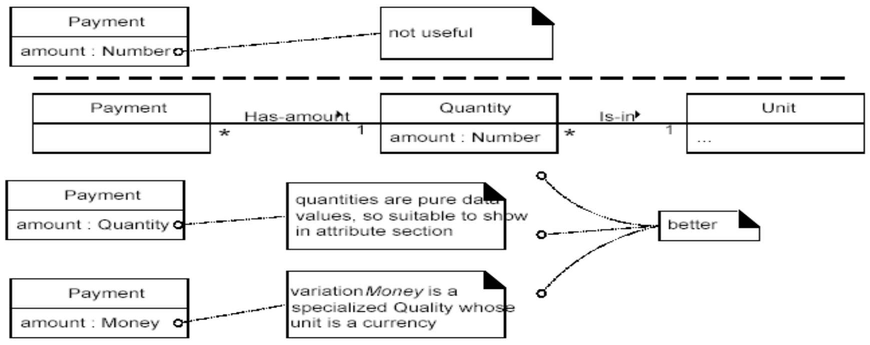
- **No Attributes as Foreign Keys**

Attributes should not be used to relate conceptual classes in the domain model.
The most common violation of this principle is to add a kind of foreign key attribute, as is typically done in relational database designs, in order to associate two types.


Worse
Cashier
name
currentRegisterNumber
a "simple" attribute, but being used as a foreign key to relate another object
Better
Cashier
name
1
Uses
1
Register
number

- **Modeling Attribute Quantities and Units**

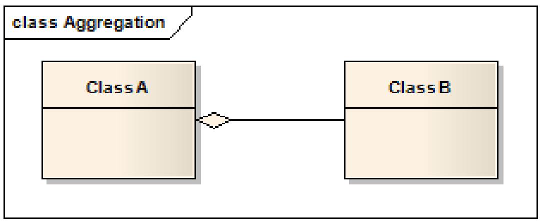
Most numeric quantities should not be represented as plain numbers.



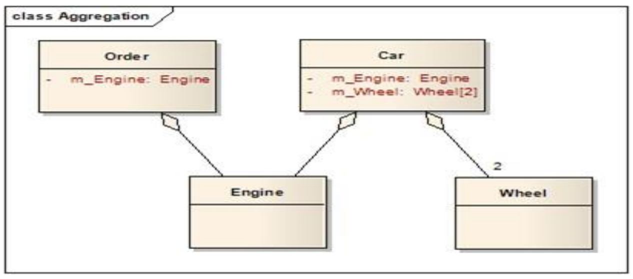
AGGREGATION (SHARED ASSOCIATION)

In cases where there's a part-of relationship between Class A (whole) and Class B (part), we can be more specific and use the aggregation link instead of the association link, taking special notice that Class B can also be aggregated by other classes in the application (therefore aggregation is also known as shared association).

- **Phrase:- "is part of"**
- **symbolized by a clear white diamond**



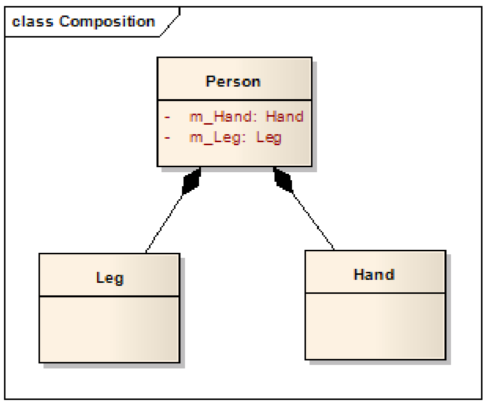
So basically, the aggregation link doesn't state in any way that Class A owns Class B nor that there is a parent-child relationship (when parent deleted all its child's are being deleted as a result) between the two. Actually, quite the opposite! The aggregation link usually used to stress the point that Class A is not the exclusive container of Class B, as in fact Class B has another container.



COMPOSITION (NOT-SHARED ASSOCIATION)

In cases where in addition to the part-of relationship between ClassA and ClassB - there's a strong life cycle dependency between the two, meaning that when ClassA is deleted then ClassB is also deleted as a result, we should be more specific and use the composition link instead of the aggregation link or the association link.

- **PHRASE -"is entirely made of"**
- symbolized by a black diamond

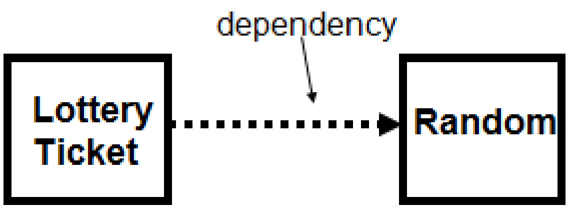


The composition link shows that a class (container, whole) has exclusive ownership over other class/s (parts), meaning that the container object and its parts constitute a parent-child/s relationship.

Unlike association and aggregation, in the composition relationship, the composed class cannot appear as a return type or parameter type of the composite class, thus changes in the composed class cannot be propagated to the rest of the system. Consequently, usage of composition limits complexity growth as the system grows.

DEPENDENCY:

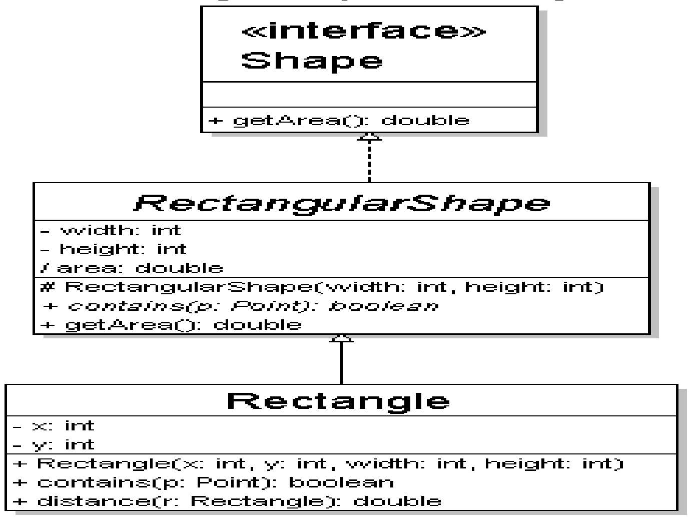
- **PHRASE**-"uses temporarily"
- Symbolized by dotted line
- often is an implementation detail, not an intrinsic part of that object's state.



GENERALIZATION RELATIONSHIPS

Generalization (inheritance) relationships

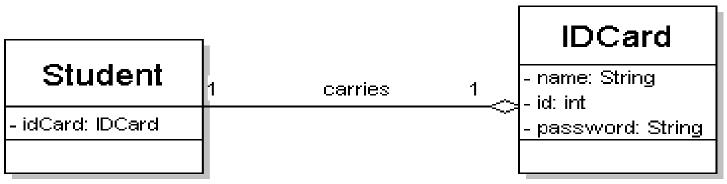
- hierarchies drawn top-down with arrows pointing upward to parent
- **line/arrow styles differ, based on whether parent is a(n):**
  - **<u>class</u>:**
    solid line, black arrow
  - **<u>abstract class</u>:**
    solid line, white arrow
  - **<u>interface</u>:**
    dashed line, white arrow
- We often don't draw trivial / obvious generalization relationships, such as drawing the Object class as a parent.


«interface»
Shape
+ getArea(): double
RectangularShape
- width: int
- height: int
/ area: double
# RectangularShape(width: int, height: int)
+ contains(p: Point): boolean
+ getArea(): double
Rectangle
- x: int
- y: int
+ Rectangle(x: int, y: int, width: int, height: int)
+ contains(p: Point): boolean
+ distance(r: Rectangle): double

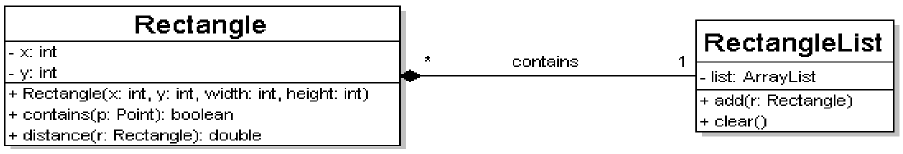
Associational (usage) relationships

1. Multiplicity (how many are used)
   - * ⇒ 0, 1, or more
   - 1 ⇒ 1 exactly
   - 2..4 ⇒ between 2 and 4, inclusive
   - 3..* ⇒ 3 or more
2. name (what relationship the objects have)
3. navigability (direction)

- one-to-one
  - each student must carry exactly one ID card



- one-to-many
  - one rectangle list can contain many rectangles


Rectangle
- x: int
- y: int
+ Rectangle(x: int, y: int, width: int, height: int)
+ contains(p: Point): boolean
+ distance(r: Rectangle): double
*
contains
1
RectangleList
- list: ArrayList
+ add(r: Rectangle)
+ clear()

Hierarchy

Hierarchy is the ranking or ordering of abstraction. Through hierarchy, a system can be made up of interrelated subsystems, which can have their own subsystems and so on until the smallest level components are reached. It uses the principle of "divide and conquer". Hierarchy allows code reusability.

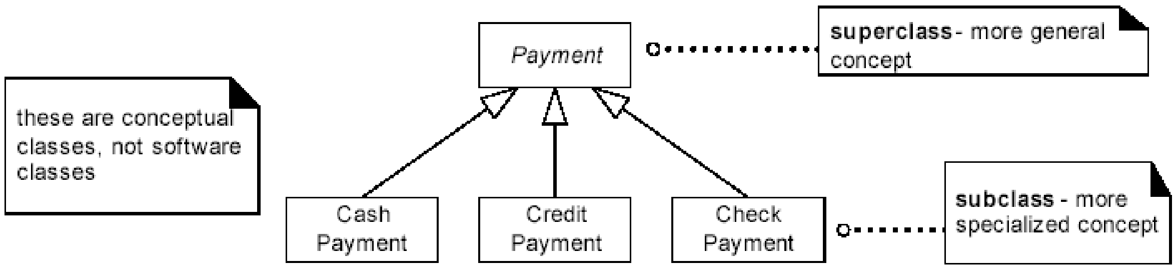
The two types of hierarchies in OOAD are −

- **"IS–A" hierarchy** − It defines the hierarchical relationship in inheritance, whereby from a super-class, a number of subclasses may be derived which may again have subclasses and so on. For example, if we derive a class Rose from a class Flower, we can say that a rose "is–a" flower.

- **"PART–OF" hierarchy** − It defines the hierarchical relationship in aggregation by which a class may be composed of other classes. For example, a flower is composed of sepals, petals, stamens, and carpel. It can be said that a petal is a "part–of" flower.

Generalization-specialization class hierarchy

The concepts *Cash Payment*, *Credit Payment*, and *Check Payment* are all very similar.

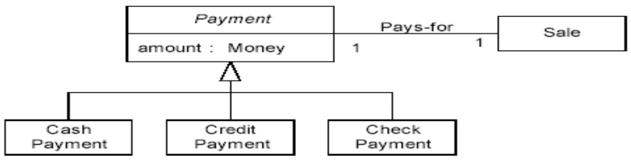


these are conceptual classes, not software classes
Payment
superclass - more general concept
Cash Payment
Credit Payment
Check Payment
subclass - more specialized concept

100% Rule

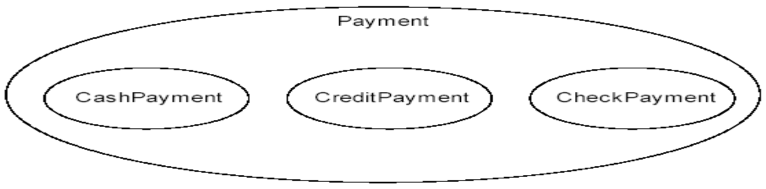
The subclass must conform to 100% of the super class's:

- attributes
- Associations

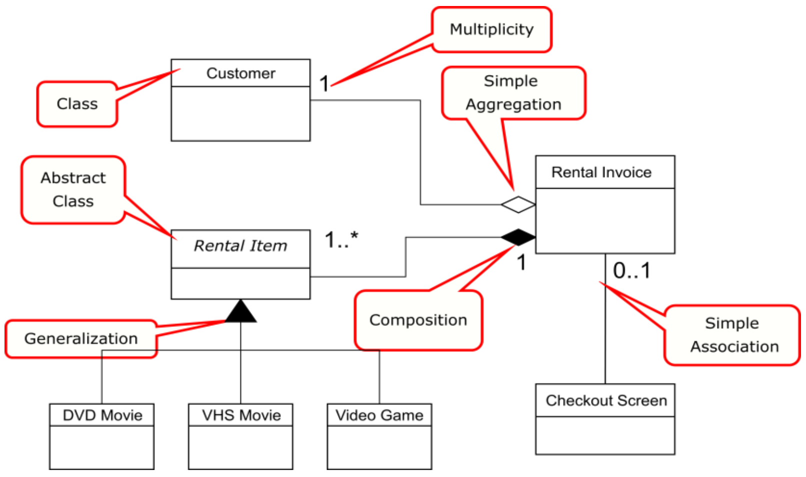


Is-a Rule

All the members of a subclass set must be members of their super class set.



CLASS DIAGRAM EXAMPLE



WHEN TO USE CLASS DIAGRAM?

Class diagrams are used for –

- Describing the static view of the system.
- Showing the collaboration among the elements of the static view.
- Describing the functionalities performed by the system.
- Construction of software applications using object oriented languages.