# UNIT 3: CASE STUDY

## ELABORATION

- ❒ Elaboration is an activity in which the information about the requirements is expanded and refined.
- ❒ This phase consists of several modelling and refinement tasks.
- ❒ During elaboration, each user scenario is parsed and various classes are identified.
- ❒ The Domain Model is primarily created during elaboration iterations. The identification of conceptual classes is part of an investigation of the problem domain.

## DOMAIN MODEL

- ★ A **domain model** is a visual representation of conceptual classes or real-world objects in a domain of interest.
- ★ A domain model illustrates meaningful conceptual classes in a problem domain
- ★ **It is the most important artefact to create during object-oriented analysis.**
- ★ A domain model is a representation of real-world conceptual classes, not of software components.
- ★ They have also been called **conceptual models**, **domain object models, and analysis object models.**
- ★ Using UML notation, a domain model is illustrated with a set of **class diagrams** in which no operations are defined. It may show:
    - domain objects or conceptual classes
    - associations between conceptual classes
    - attributes of conceptual classes

### Features of a domain model

•**Domain classes** – each domain class denotes a type of object.

•**Attributes** – an attribute is the description of a named slot of a specified type in a domain class; each instance of the class separately holds a value.

•**Associations** – an association is a relationship between two (or more) domain classes that describes links between their object instances. Associations can have roles, describing the multiplicity and participation of a class in the relationship.

•**Additional rules** – complex rules that cannot be shown with symbolically can be shown with attached notes.

## Steps to create a Domain Model

1. Identify candidate conceptual classes

2. Draw them in a UML domain model

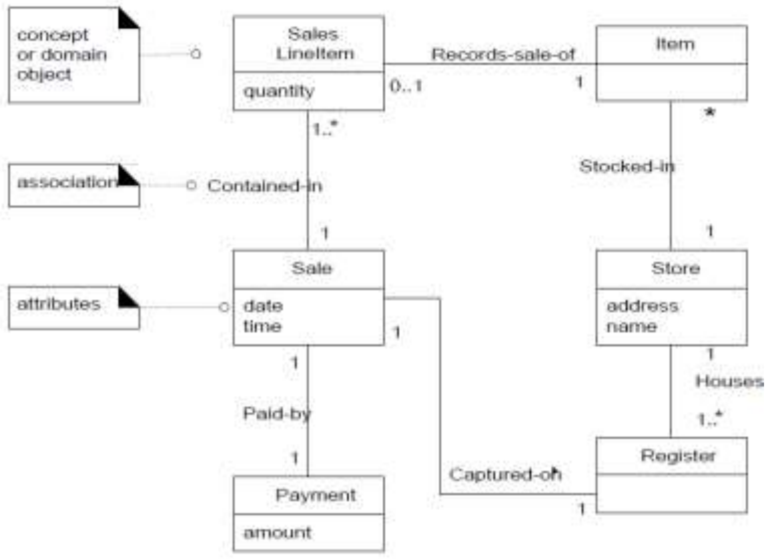
3. Add associations necessary to record the relationships that must be retained

4. Add attributes necessary for information to be preserved

5. Use existing names for things, the vocabulary of the domain

## Partial Domain model

It illustrates that the conceptual class of Payment and Sale are significant in this domain, that a Payment is related to a Sale in a way that is meaningful to note, and that a Sale has a date and time.



It also depicts an abstraction of the conceptual classes, because there are many things one could communicate about registers, sales, and so forth.
**Thus, the domain model may be considered a visual dictionary of the noteworthy abstractions, domain vocabulary, and information content of the domain.**

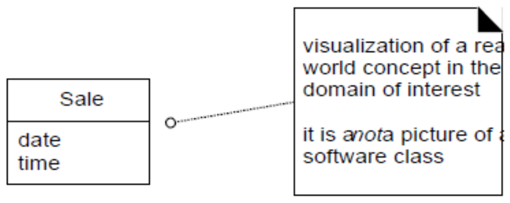

Sale
date
time
visualization of a real world concept in the domain of interest
it is a not a picture of a software class

Figure 10.2 A domain model shows real-world conceptual classes, not software classes.

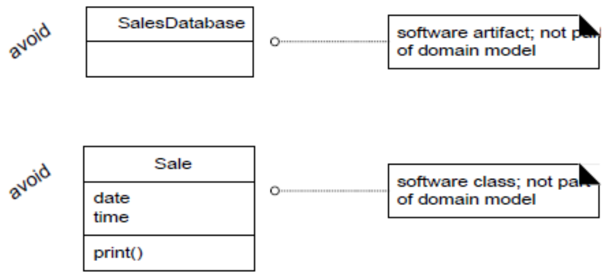

avoid
SalesDatabase
software artifact; not part of domain model
avoid
Sale
date
time
print()
software class; not part of domain model

Figure 10.3 A domain model does not show software artifacts or classes.

## Conceptual Classes

The domain model illustrates conceptual classes or vocabulary in the domain. More formally, 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.

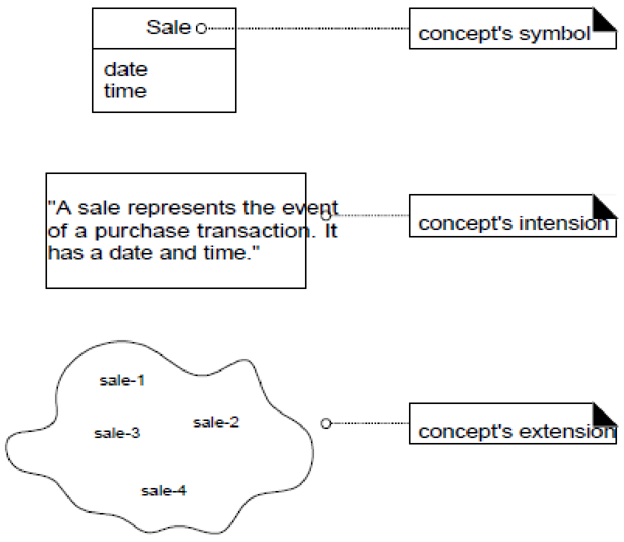


Figure 10.4 A conceptual class has a symbol, intension, and extension.

## Conceptual Classes in the Sale Domain

For example, in the real-world domain of sales in a store, there are the conceptual classes of Store, Register, and Sale. Therefore, our domain model, shown in diagram below, may include Store, Register, and Sale.

Store
Register
Sale

Figure 10.5 Partial domain model in the domain of the store.

## Conceptual Class identification

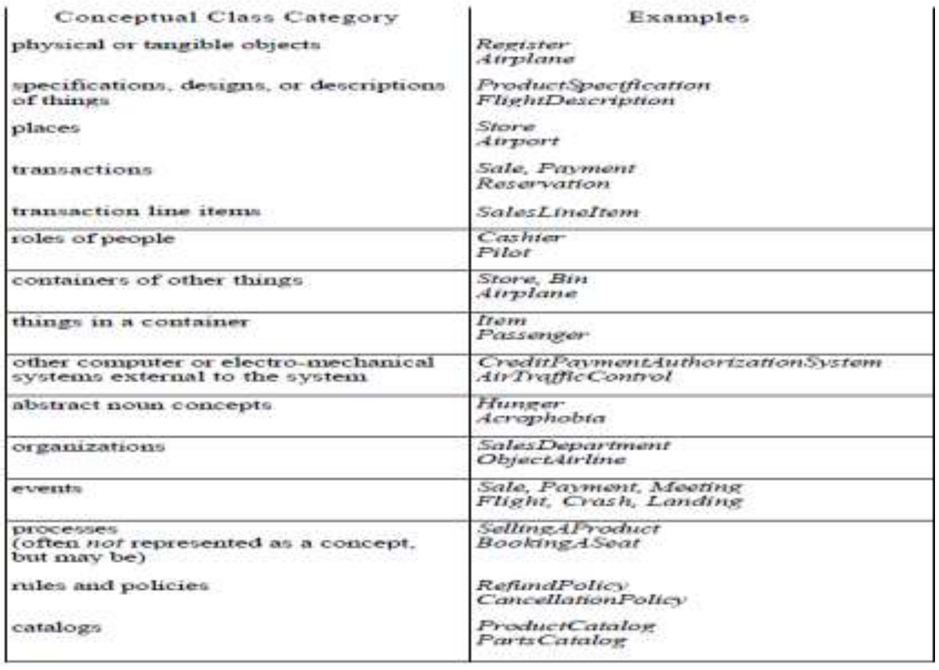
Our goal is to create a domain model of interesting or meaningful conceptual classes in the domain of interest (sales). In this case, that means concepts related to the use case Process Sale.
In iterative development, one incrementally builds a domain model over several iterations in the elaboration phase.

## Strategies to Identify Conceptual Classes

Two techniques are presented in the following sections:
1. Use a conceptual class category list.
2. Identify noun phrases.

## 1. Use a Conceptual Class Category List

Start the creation of a domain model by making a list of candidate conceptual classes.
For airline domain, the conceptual classes category list is as follows:



| Conceptual Class Category | Examples |
|---|---|
| physical or tangible objects | *Register* *Airplane* |
| specifications, designs, 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* |
| abstract noun concepts | *Hunger* *Acrophobia* |
| organizations | *SalesDepartment* *ObjectAirline* |
| events | *Sale, Payment, Meeting* *Flight, Crash, Landing* |
| processes (often *not* represented as a concept, but may be) | *SellingAProduct* *BookingASeat* |
| rules and policies | *RefundPolicy* *CancellationPolicy* |
| catalogs | *ProductCatalog* *PartsCatalog* |

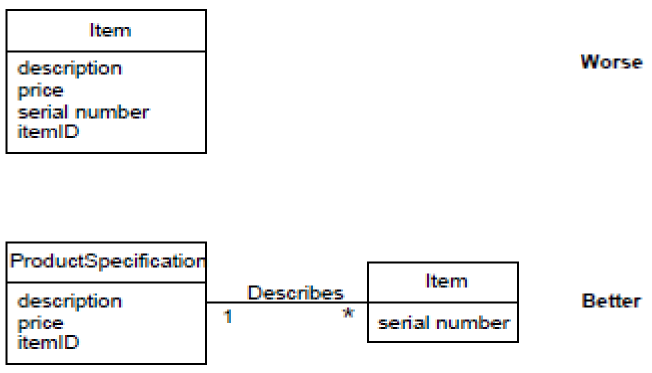
## Candidate Conceptual Classes for the Sales Domain

From the Conceptual Class Category List and noun phrase analysis, a list is generated of candidate conceptual classes for the domain.

- Register
- Item
- Store
- Sale
- Payment
- ProductCatalog
- ProductSpecification
- SalesLineItem
- Cashier
- Customer
- Manager

## DESCRIPTION CLASSES

- A description class contains information about something else. For example, ProductSpecification .
- Description or specification objects are strongly related to the things theydescribe. In a domain model, it is common to state that an XSpecification Describes an X

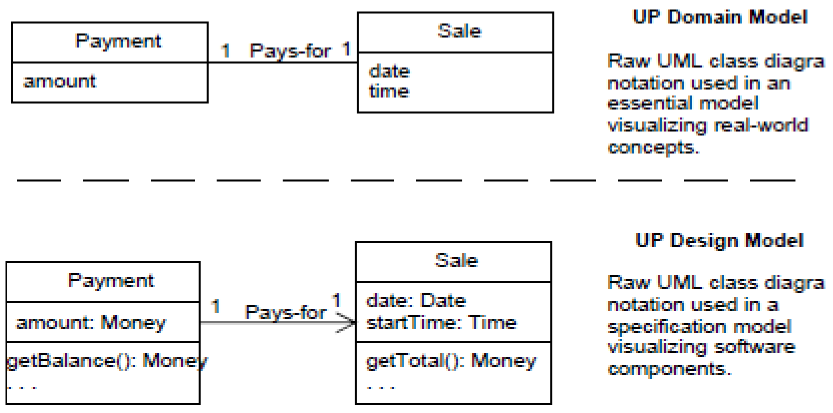


Description classes are used when:

- 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 Model and Design model:

For example, in the UP, when the UML boxes are drawn in the Domain Model, they may be called **domain concepts** or **conceptual classes;** the Domain Model offers a conceptual perspective. In the UP, when UML boxes are drawn in the Design Model, they are officially called **design classes;** the Design Model offers a specification or implementation perspective, as desired by the modeler.



Payment
amount
1 Pays-for 1
Sale
date
time
UP Domain Model
Raw UML class diagra notation used in an essential model visualizing real-world concepts.
Payment
amount: Money
getBalance(): Money
. . .
1 Pays-for 1
Sale
date: Date
startTime: Time
getTotal(): Money
. . .
UP Design Model
Raw UML class diagra notation used in a specification model visualizing software components.

## Domain Model for NextGEN POS Case study:

The list of conceptual classes generated for the NextGen POS domain may be represented graphically (see Figure 10.11) to show the start of the Domain Model.

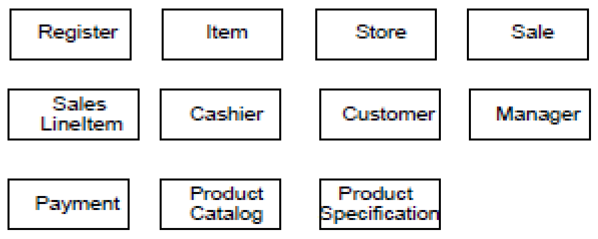


Figure 10.11 Initial Domain Model.

## ASSOCIATIONS

An **association** is a relationship between types that indicates some meaningful and interesting connection.

In the UML associations are defined as "the semantic relationship between two or more classifiers that involve connections among their instances."

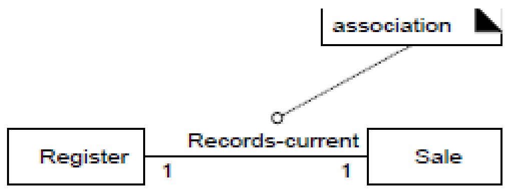


Figure 11.1 Associations.

## The UML Association Notation

- 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.
- 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.

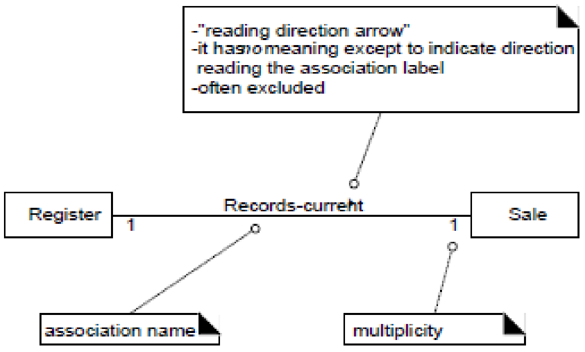
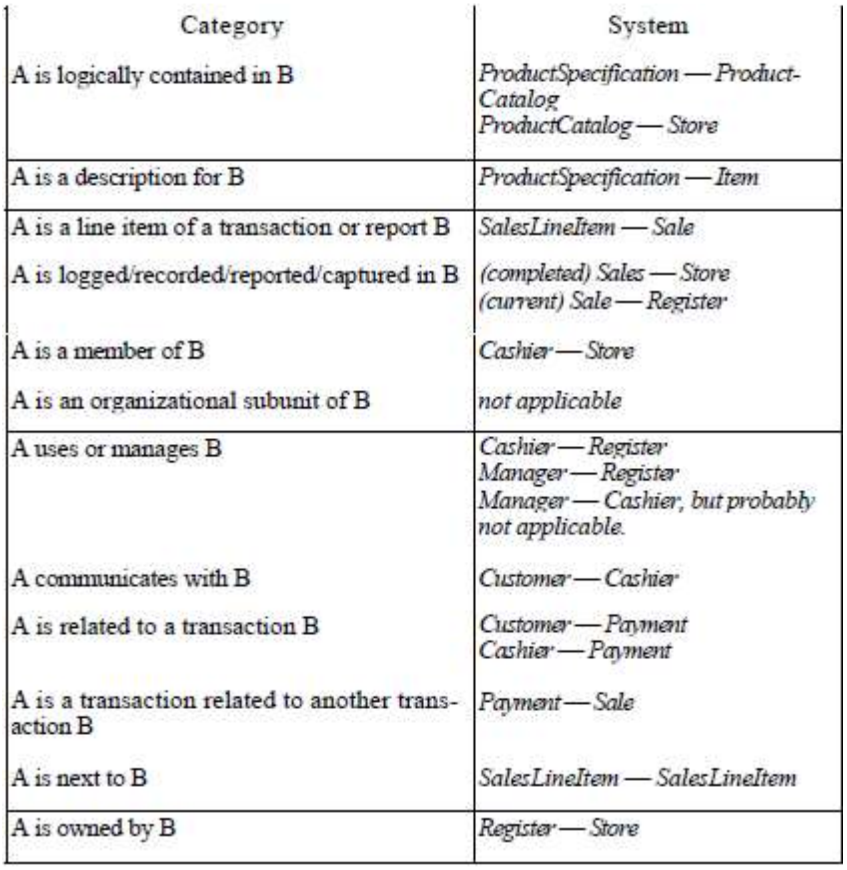

-"reading direction arrow"
-it has no meaning except to indicate direction reading the association label
-often excluded
Register
Records-current
Sale
1
1
association name
multiplicity

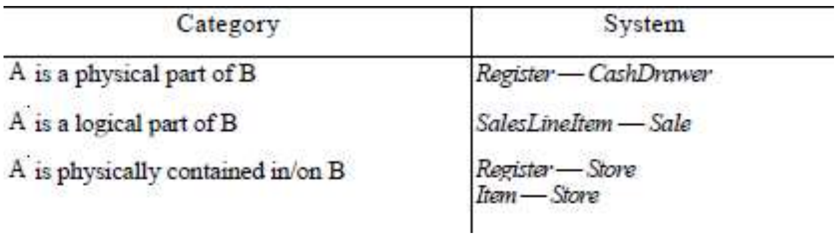
Figure 11.2 The UML notation for associations.

## Common Association List:

The common Association List for the NextGEN POS Case Study is :



| Category | System |
|---|---|
| A is logically contained in B | *ProductSpecification — Product-Catalog*<br>*ProductCatalog — Store* |
| A is a description for B | *ProductSpecification — Item* |
| A is a line item of a transaction or report B | *SalesLineItem — Sale* |
| A is logged/recorded/reported/captured in B | *(completed) Sales — Store*<br>*(current) Sale — Register* |
| A is a member of B | *Cashier — Store* |
| A is an organizational subunit of B | *not applicable* |
| A uses or manages B | *Cashier — Register*<br>*Manager — Register*<br>*Manager — Cashier, but probably not applicable.* |
| A communicates with B | *Customer — Cashier* |
| A is related to a transaction B | *Customer — Payment*<br>*Cashier — Payment* |
| A is a transaction related to another transaction B | *Payment — Sale* |
| A is next to B | *SalesLineItem — SalesLineItem* |
| A is owned by B | *Register — Store* |



| Category | System |
|---|---|
| A is a physical part of B | *Register — CashDrawer* |
| A is a logical part of B | *SalesLineItem — Sale* |
| A is physically contained in/on B | *Register — Store*<br>*Item — Store* |

## Association Guidelines:

• Focus on those associations for which knowledge of the relationship needs to be preserved for some duration.
• 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.

## Roles and Multiplicity:

- ❖ Each end of an association is called a **role**.
- ❖ **Multiplicity** defines how many instances of a class A can be associated with one instance of a class B.
- ❖ •e.g.: a single instance of a Store can be associated with “many”(zero or more) Item instances.

## Naming Associations

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

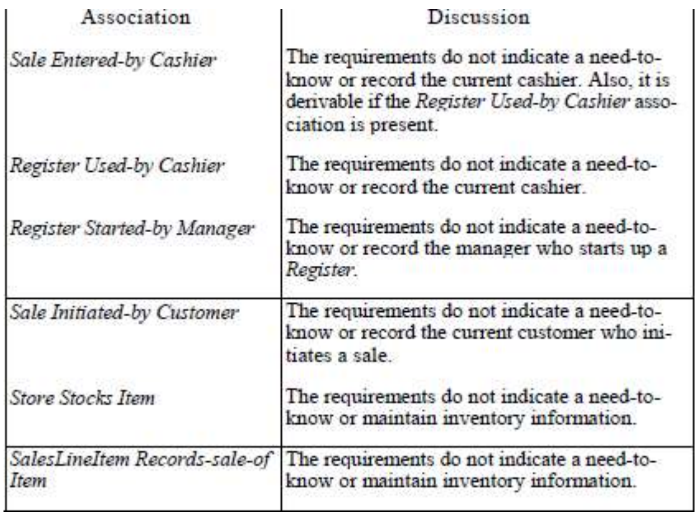
## NextGEN POS Domain Model Associations

We can now add associations to our POS domain model.
The following sample of associations is justified in terms of a need-to-know. It is based on the use cases currently under consideration.

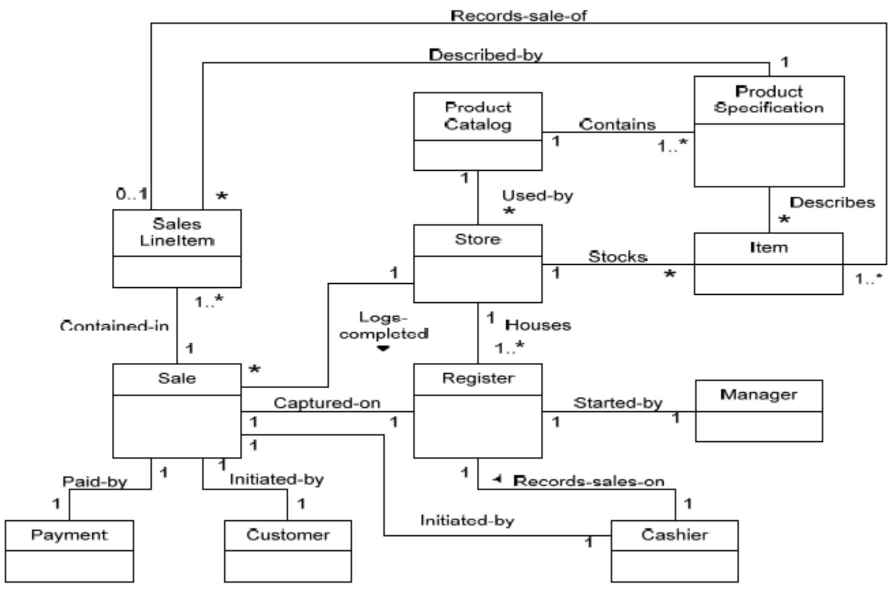
- o **Register Records Sale**
- o **Sale Paid-by Payment**
- o **ProductCatalog Records ProductSpecification**

To know the current sale, generate a total, print a receipt.
To know if the sale has been paid, relate the amount tendered to the sale total, and print a receipt.
To retrieve an ProductSpecification, given an itemID.

Applying the Category of Associations Checklist
We will run through the checklist, based on previously identified types, considering the current use case requirements.



| Association | Discussion |
|---|---|
| *Sale Entered-by Cashier* | The requirements do not indicate a need-to-know or record the current cashier. Also, it is derivable if the *Register Used-by Cashier* association is present. |
| *Register Used-by Cashier* | The requirements do not indicate a need-to-know or record the current cashier. |
| *Register Started-by Manager* | The requirements do not indicate a need-to-know or record the manager who starts up a *Register*. |
| *Sale Initiated-by Customer* | The requirements do not indicate a need-to-know or record the current customer who initiates a sale. |
| *Store Stocks Item* | The requirements do not indicate a need-to-know or maintain inventory information. |
| *SalesLineItem Records-sale-of Item* | The requirements do not indicate a need-to-know or maintain inventory information. |



## ATTRIBUTES

**An attribute** is a logical data value of an object.

For example, a receipt (which reports the information of a sale) normally includes a date and time, and management wants to know the dates and times of sales for a variety of reasons. Consequently, the Sale conceptual class needs a date and time attribute.

### UML Attribute Notation

Attributes are shown in the second compartment of the class box

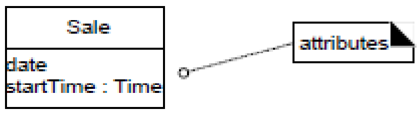
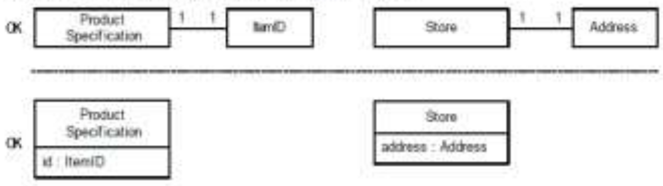


Figure 12.1 Class and attributes.

- The attributes in a domain model should preferably be simple attributes or data types.
- Very common attribute data types include: Boolean, Date, Number, String(Text), Time.
- Other common types include: Address, Color, Geometries (Point, Rectangle), Phone Number, Social Security Number,

- Represent as a non-primitive class if :
  - It is composes of separate sections
    - Phone number, name, ...
  - There are operations associated with it.
  - It has other attributes
  - It is a quantity with a unit
- That means :
  - Address and Quantity are data types that can be considered as separate classes.
  - But it depends on what you wants to emphasize in the diagram
    - It may be shown in the attribute of the class box
  - A domain model is a tool of communication



## Attributes in the NextGEN POS model:

The attributes chosen reflect the requirements for this iteration—the Process *Sale* scenarios of this iteration.

| | |
|---|---|
| *Payment* | *amount*—To determine if sufficient payment was provided, and to calculate change, an amount (also known as "amount tendered") must be captured. |
| *Product-Specification* | *description*—To show the description on a display or receipt. |
| | *id*—To look up a *ProductSpecification*, given an entered itemID, it is necessary to relate them to a *id*. |
| | *price*—To calculate the sales total, and show the line item price. |
| *Sale* | *date, time*—A receipt is a paper report of a sale. It normally shows date and time of sale. |
| *SalesLineItem* | *quantity*—To record the quantity entered, when there is more than one item in a line item sale (for example, *five* packages of tofu). |
| *Store* | *address, name*—The receipt requires the name and address of the store. |

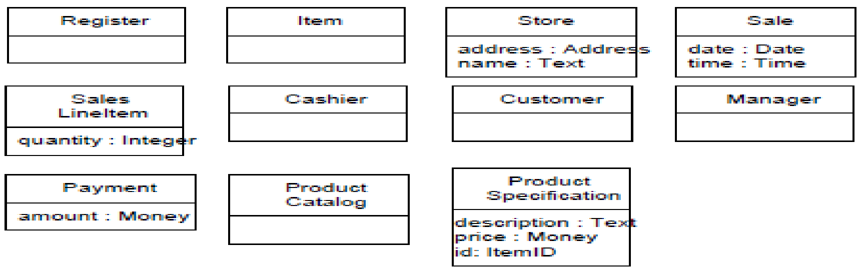


Figure 12.7 Domain model showing attributes.

## DOMAIN MODEL REFINEMENT

- Refine the domain model with generalizations, specializations, association classes, time intervals, composition, and packages.
- Identify domain super classes and subclasses relevant to the current investigation, and illustrate them in the Domain Model.
- Generalization and specialization are fundamental concepts in domain modeling that support an economy of expression

### Generalisation and Specialization

- **Generalization** is the activity of identifying commonality among concepts and defining superclass (general concept) and subclass (specialized concept) relationships.
- It is a way to construct taxonomic classifications among concepts which are then illustrated in class hierarchies.
- Identifying a superclass and subclasses is of value in a domain model because their presence allows us to understand concepts in more general, refined and abstract terms.
- **Superclass** Payment represents a more general concept, and the **subclasses** more specialized ones.

  In the given example, the concepts CashPayment, CreditPayment, and Check Payment are all very similar.

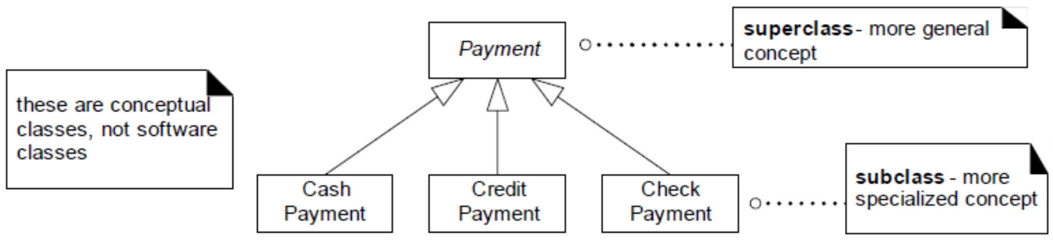

Payment
superclass - more general concept
these are conceptual classes, not software classes
Cash Payment
Credit Payment
Check Payment
subclass - more specialized concept

Figure 26.1 Generalization-specialization hierarchy.

For example, consider the superclass Payment and its subclasses (CashPayment, and so on). Assume the definition of Payment is that it represents the transaction of transferring money (not necessarily cash) for a purchase from one party

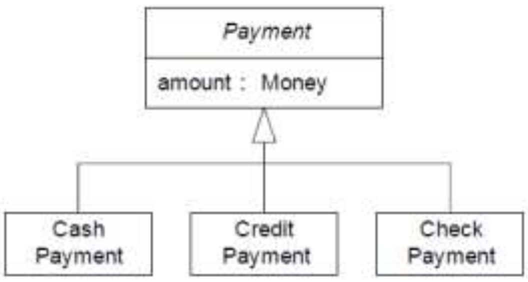


*Figure 26.3 Payment class hierarchy.*

## Generalisation guidelines:

• Identify superclasses and subclasses when they help us understand concepts in more general, abstract terms and reduce repeated information.
• Expand class hierarchy relevant to the current iteration and show them in the Domain Model.

### Conformance Rule:

All Payment subclasses must conform to having an amount and paying for a Sale. In general, this rule of conformance to a superclass definition is the 100% Rule:

*100% Rule*

100% of the conceptual superclass's definition should be applicable to the sub-class. The subclass must conform to 100% of the superclass's:

- attributes
- associations

Conceptual Sub Class Conformance

Informally, this expresses the notion that the conceptual subclass is a kind of superclass. CreditPayment is a kind of Payment. More tersely, is-a-kind-of is called is-a.

*Is-a Rule*

All the members of a subclass set must be members of their superclass set. In natural language, this can usually be informally tested by forming the statement: *Subclass is a Superclass*

A potential subclass should conform to the:

- 100% Rule (definition conformance)
- Is-a Rule (set membership conformance)

**Create a conceptual subclass of a superclass when:**
1. The subclass has additional attributes of interest.
2. The subclass has additional associations of interest.
3. The subclass concept is operated on, handled, reacted to, or manipulated differently than the superclass or other subclasses, in ways that are of interest.

**Create a conceptual superclass in a generalization relationship to subclasses when:**
- The potential conceptual subclasses represent variations of a similar concept.
- The subclasses will conform to the 100% and Is-a rules.
- All subclasses have the same attribute which can be factored out and expressed in the superclass.
- All subclasses have the same association which can be factored out and related to the superclass.

## FINDING CONCEPTUAL CLASS HIERARCHIES

### Payment Classes

Based on the above criteria for partitioning the Payment class, it is useful to create a class hierarchy of various kinds of payments. The justification for the superclass and subclasses is shown in Figure 26.7.

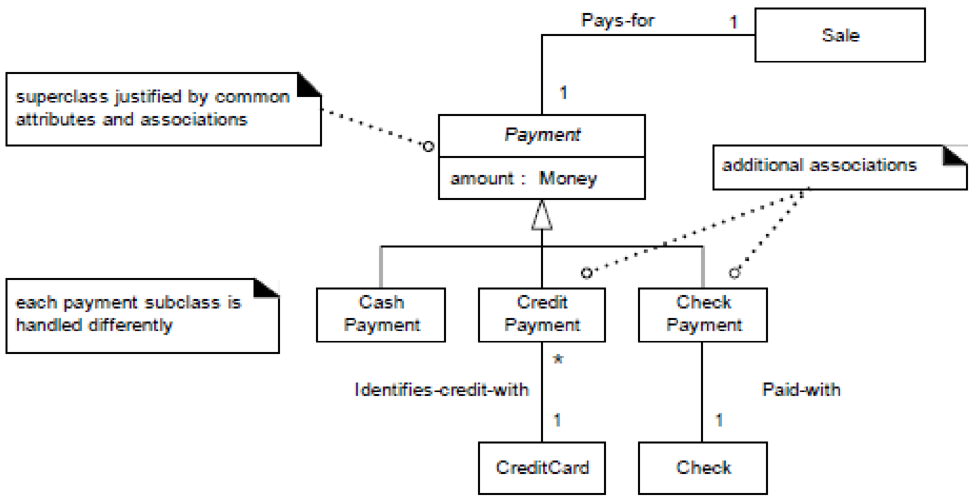


*Figure 26.7 Justifying Payment subclasses.*

**Authorization Service Classes**

Credit and check authorization services are variations on a similar concept, and have common attributes of interest. This leads to the class hierarchy in Figure 26.8.

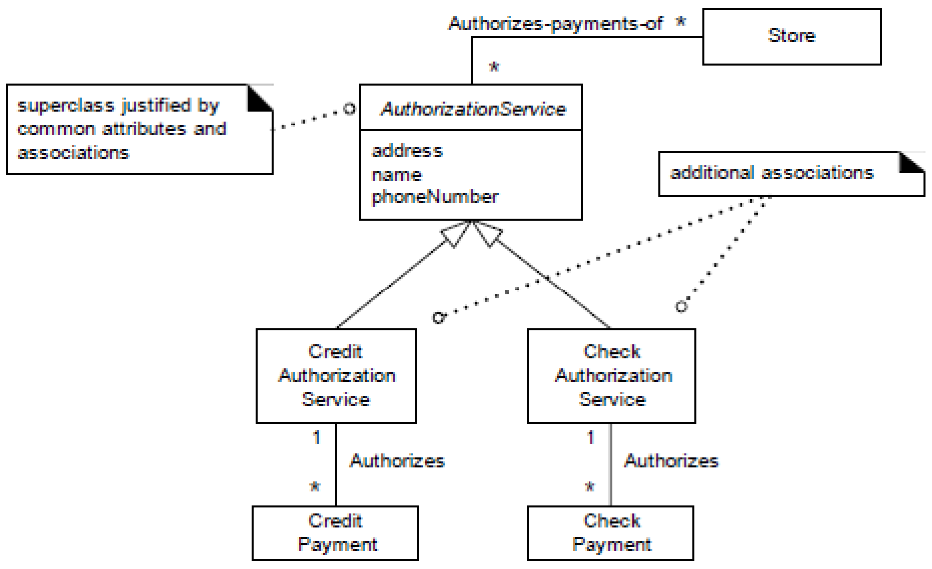


*Figure 26.8 Justifying the AuthorizationService hierarchy.*

## AGGREGATION AND COMPOSITION

**Aggregation** is a kind of association used to model whole-part relationships between things. The whole is called the **composite.**

For instance, physical assemblies are organized in aggregation relationships, such as a Hand aggregates Fingers.

### Aggregation in the UML

Aggregation is shown in the UML with a hollow or filled diamond symbol at the composite end of a whole-part association

Aggregation is a property of an association role.

The association name is often excluded in aggregation relationships since it is typically thought of as Has-part.

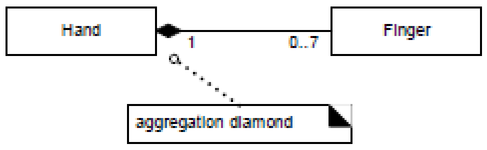


*Figure 27.5 Aggregation notation.*

### Consider showing aggregation when:

• The lifetime of the part is bound within the lifetime of the composite — there is a create-delete dependency of the part on the whole.

• There is an obvious whole-part physical or logical assembly.
• Some properties of the composite propagate to the parts, such as the location.
• Operations applied to the composite propagate to the parts, such as destruction, movement, and recording.

## *Aggregation in the POS Domain Model*

In the POS domain, the *SalesLineItems* may be considered a part of a composite *Sale;* in general, transaction line items are viewed as parts of an aggregate transaction (see Figure 27.8). In addition to conformance to that pattern, there is a create-delete dependency of the line items on the *Sale*—their lifetime is bound within the lifetime of the *Sale.*

By similar justification, *ProductCatalog* is an aggregate of *Product-Specifications.*

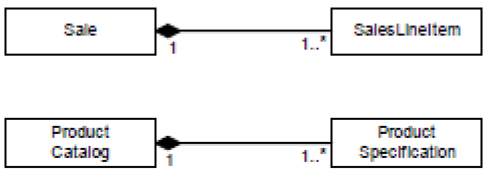


*Figure 27.8 Aggregation in the point-of-sale application.*

**Composite aggregation, or composition,** means that the part is a member of only one composite object, and that there is an existence and disposition dependency of the part on the composite.
For example, a hand is in a composition relationship to a finger.

In the Design Model, composition and its existence dependency implication indicates that composite software objects create (or caused the creation of) the part software objects (for example, Sale creates SalesLineItem).

But in the Domain Model, since it does not represent software objects, the notion of the whole creating the part is seldom relevant (a real sale does not create a real sales line item).

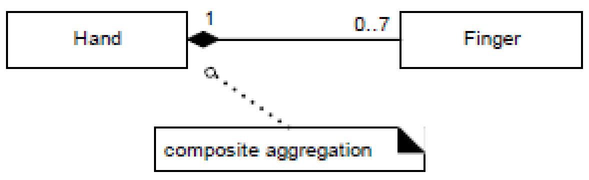


*Figure 27.6 Composite aggregation.*

**Inception , Use Case Modelling, Relating Usecases** -> Refer Technical , chap 30 in Craig Larman 3rd editionss