CLASS DIAGRAM

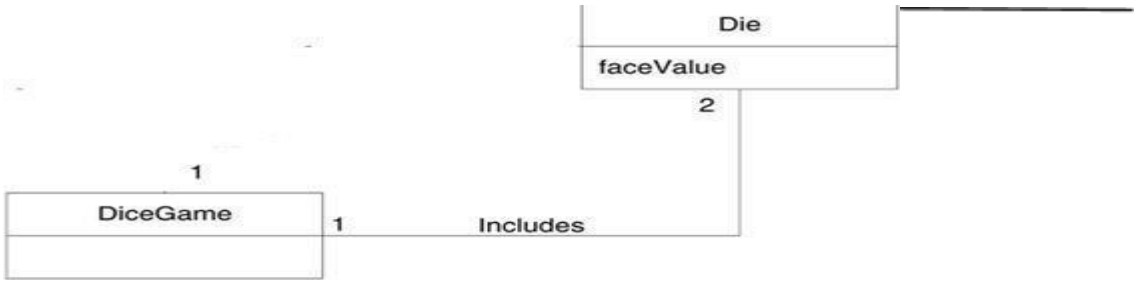
Introduction

- The UML includes class diagrams to illustrate classes, interfaces, and their associations. They are used for static object modeling.
- Used for static object modeling. It is used to depict the classes within a model.
- It describes responsibilities of the system, it is used in forward and reverse engineering
- Keywords used along with class name are {abstract, interface, actor}

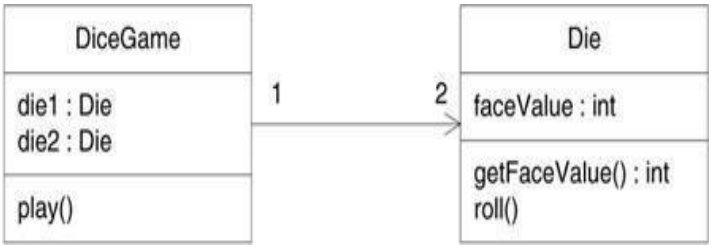
Definition: Design Class Diagram

The class diagram can be used to visualize a domain model. we also need a unique term to clarify when the class diagram is used in a software or design perspective. A common modeling term for this purpose is design class diagram (DCD).

UML class diagrams in two perspectives Domain Model

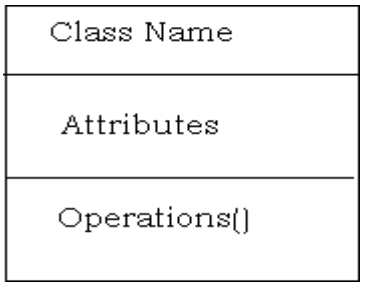


Design Model



Class Diagram Representation

Class is represented as rectangular box showing class name, attributes , operations.



The main elements of class are

1. ***Attributes***
2. ***Operations & Methods***
3. ***Relationship between classes***

1.Attributes (refer pg no 26)

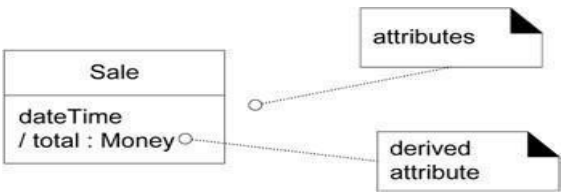
An **attribute** is a logical data value of an object. Attributes of a classifier also called structural properties in the UML. The full format of the attribute text notation is:

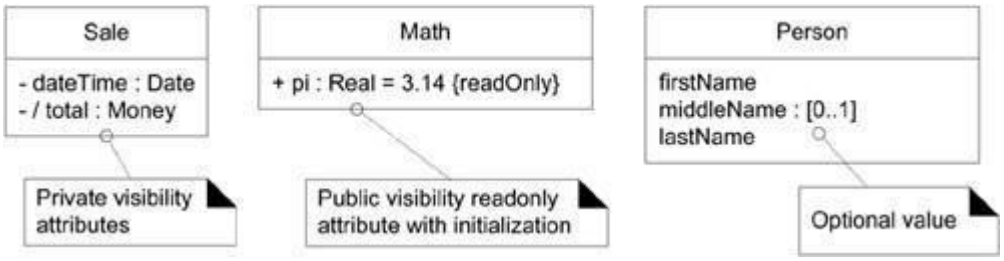
Syntax:

visibility name: type multiplicity = default {property-string}

visibility marks include + (public), - (private)

Examples:

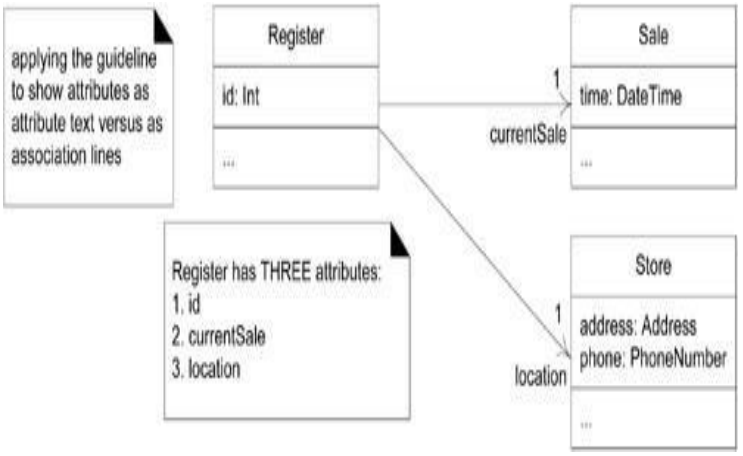




different formats

-classOrStaticAttribute:int
+ publicAttribute : string
- privateAttribute
assumedPrivateAttribute
isIntializedAttribute : Bool = true
aCollection:VeggieBurger [*]
attributeMayLegallyBeNull : String[0..1]
finalConstantattribute : int = 5 { readonly}
/derivedAttribute

Guideline: Use the attribute text notation for data type objects and the association line notation for others.



2. Operations and Methods

Operations: One of the compartments of the UML class box shows the signatures of operations. Assume the version that includes a return type. Operations are usually assumed public if no visibility is shown. both expressions are possible

An operation is not a method. A UML **operation** is a declaration, with a name, parameters, return type, exceptions list, and possibly a set of constraints of pre-and post-conditions. methods are implementations.

Syntax:

visibility name (parameter-list): return-type {property-string}

Example:

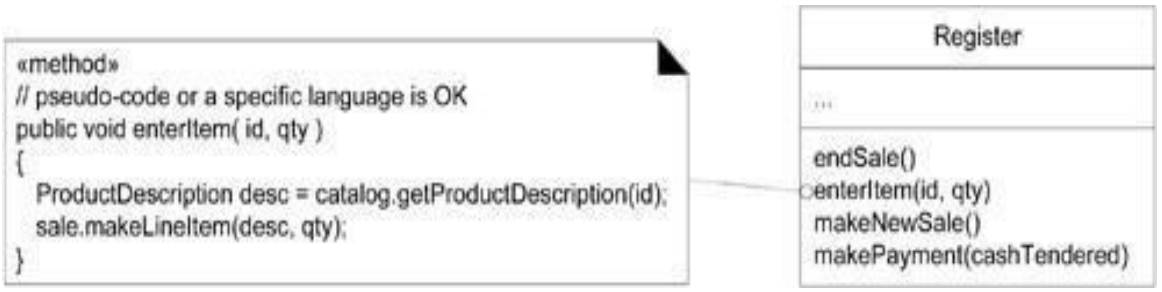
UML REPRESENTATION	+ getPlayer(name : String) : Player {exception IOException}
JAVA CODING	public Player getPlayer(String name) throws IOException

```
+ classOrStaticMethod()
+ publicMethod()
assumedPublicMethod()
- privateMethod()
# protectedMethod()
~ packageVisibleMethod()
«constructor» SuperclassFoo( Long )
methodWithParms(parm1 : String, parm2 : Float)
methodReturnsSomething() : VeggieBurger
methodThrowsException() {exception IOException}
abstractMethod()
abstractMethod2() { abstract }   // alternate
finalMethod() { leaf }  // no override in subclass
synchronizedMethod() { guarded }
```

To Show Methods in Class Diagrams:

A UML **method** is the implementation of an operation. A method may be illustrated several ways, including:

- in interaction diagrams, by the details and sequence of messages
- in class diagrams, with a UML note symbol stereotyped with «method»

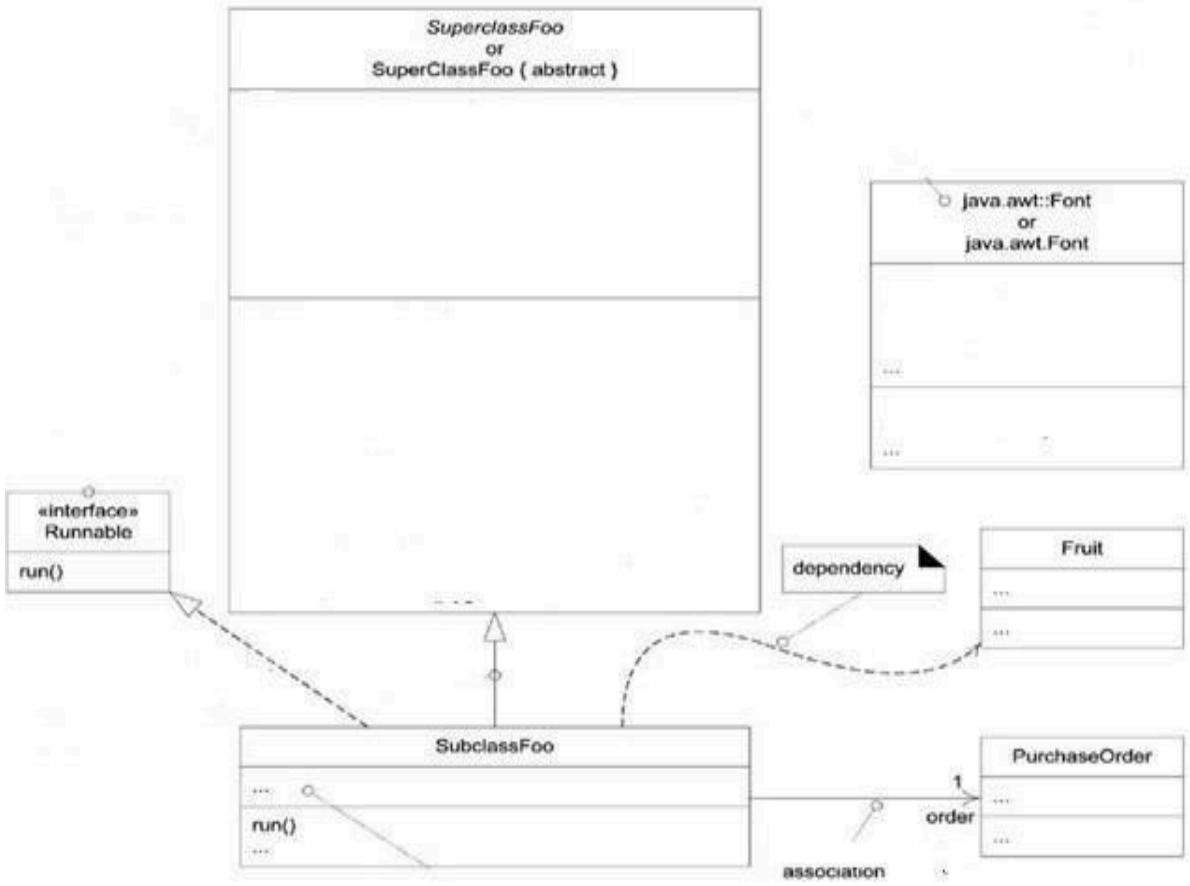


3.Relationship between classes

There are different relationship exists between classes. They are

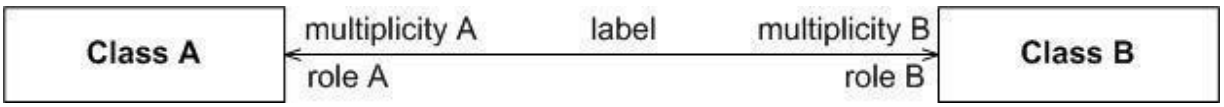
A. Association
B. Generalization & specialization
C. Composition and aggregation
D. Dependency

E. Interface realization

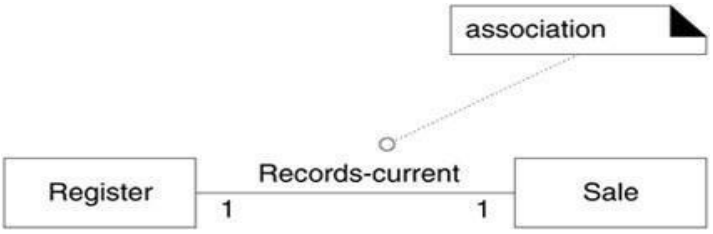


A) Association (refer page no 21)

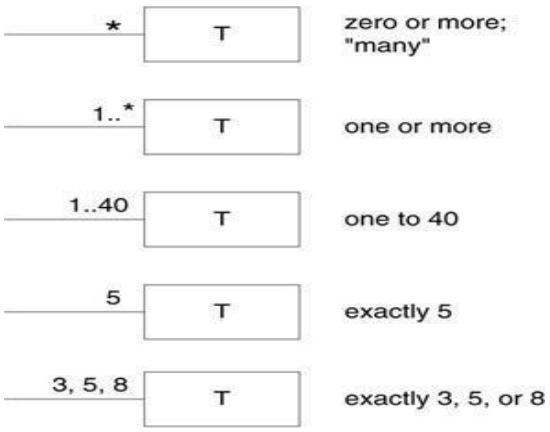
An **association** is a relationship between classes. The semantic relationship between two or more classifiers that involve connections among their instances.



Example



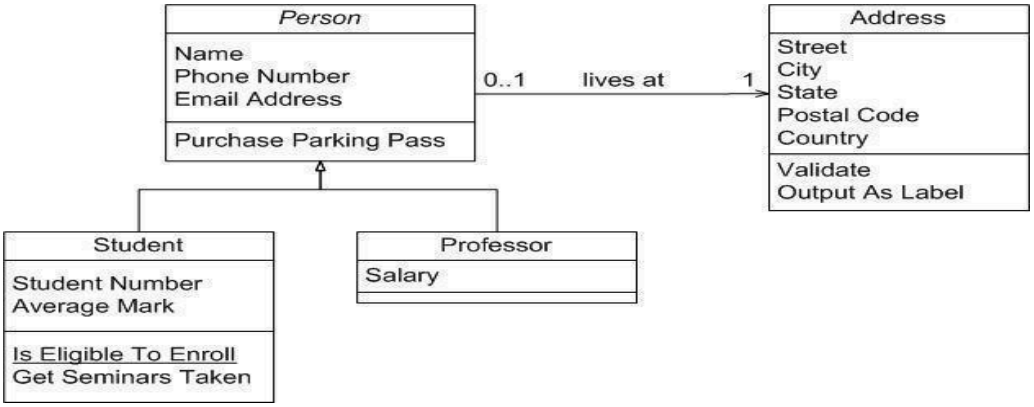
For example, a single instance of a class can be associated with "many" (zero or more, indicated by the *) Item instances.



B) Generalization & Specialization

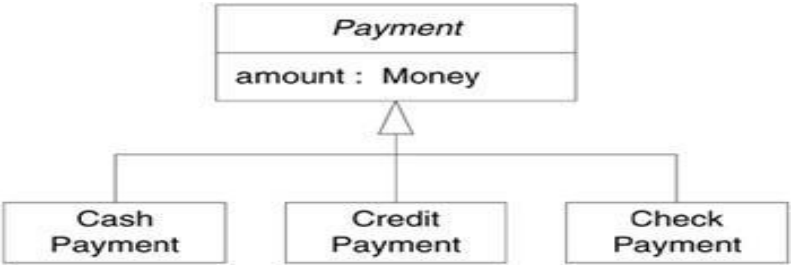
Generalization is the activity of identifying commonality among concepts and defining superclass (general concept) and subclass (specialized concept) relationships.

Ex1:



In the above example person is the generalized class and specialized classes are student and professor

Ex2:

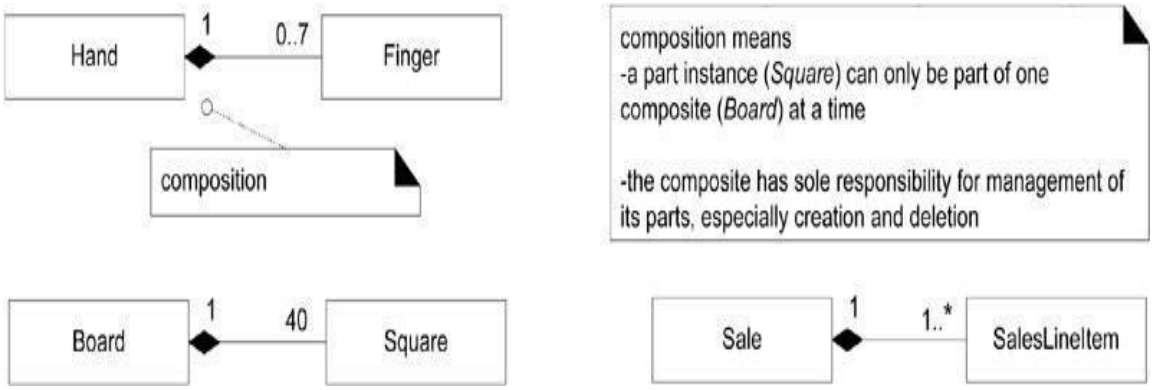


In the above example payment is the generalized class and specialized classes are cash payment , credit payment and check payment .

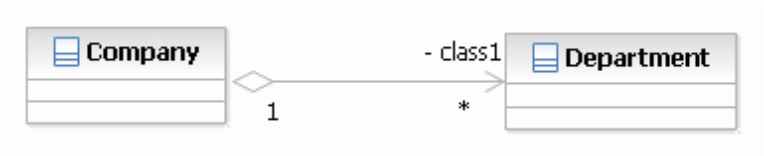
C) Composition and Aggregation

Composition, also known as composite aggregation, is a strong kind of whole-part aggregation and is useful to show in some models. A composition relationship implies that

1) an instance of the part (such as a Square) belongs to only one composite instance (such as one Board) at a time,
2) the part must always belong to a composite (no free-floating Fingers)
3) the composite is responsible for the creation and deletion of its parts either by itself creating/deleting the parts, or by collaborating with other objects.



Aggregation is a vague kind of association in the UML that loosely suggests whole-part relationships .Aggregation implies a relationship where the child can exist independently of the parent. Example: Class (parent) and Student (child). Delete the Class and the Students still exist.



For example, a Department class can have an aggregation relationship with a Company class, which indicates that the department is part of the company. Aggregations are closely related to compositions.

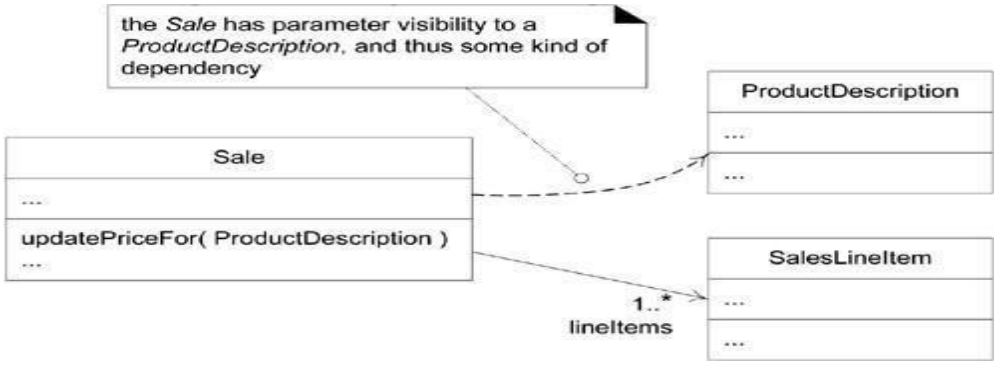
D) Dependency

A general dependency relationship indicates that a client element (of any kind, including classes, packages, use cases, and so on) has knowledge of another supplier element and that a change in the supplier could affect the client.

Dependency can be viewed as another version of **coupling**, a traditional term in software development when an element is coupled to or depends on another.

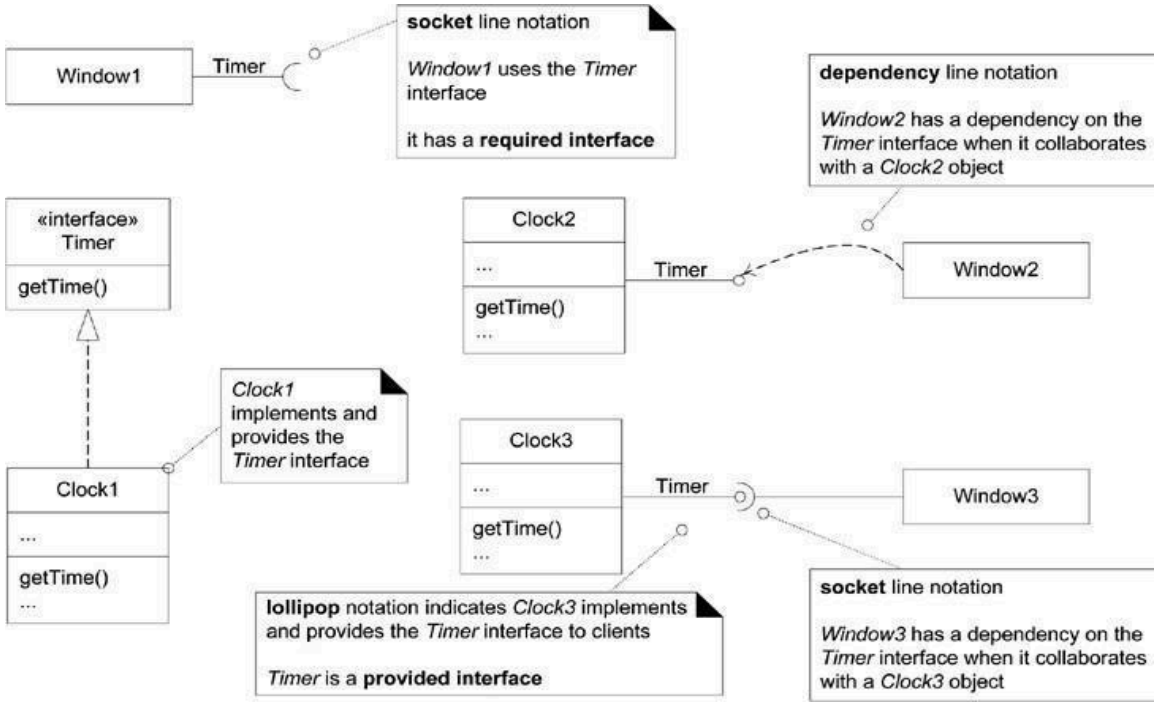
There are many kinds of dependency

- having an attribute of the supplier type
- sending a message to a supplier; the visibility to the supplier could be:
    - an attribute, a parameter variable, a local variable, a global variable, or class visibility (invoking static or class methods)
- receiving a parameter of the supplier type
- the supplier is a superclass or interface



E) Interface realization

The UML provides several ways to show **interface** implementation, providing an interface to clients, and interface dependency (a required interface). In the UML, interface implementation is formally called interface realization

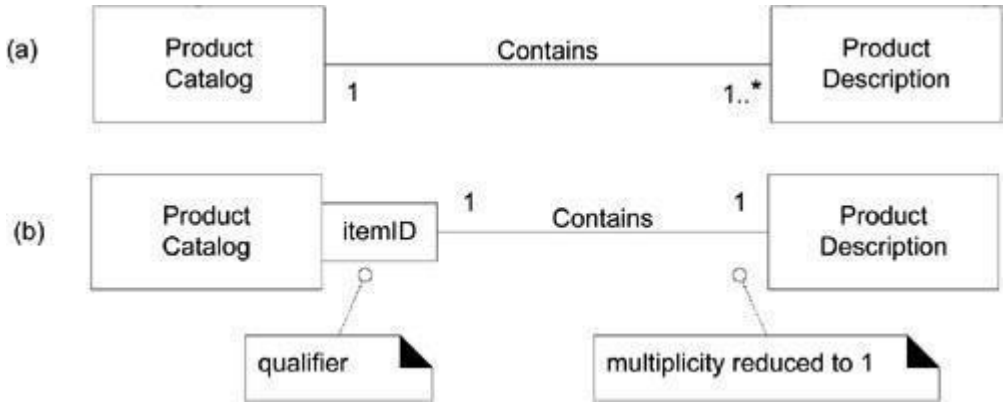


In the above example , Clock is the server program implementing Timer interface giving Timer as the provided interface, window is the client program with Timer

as required interface. The Timer interface contains the services provided by the server object.

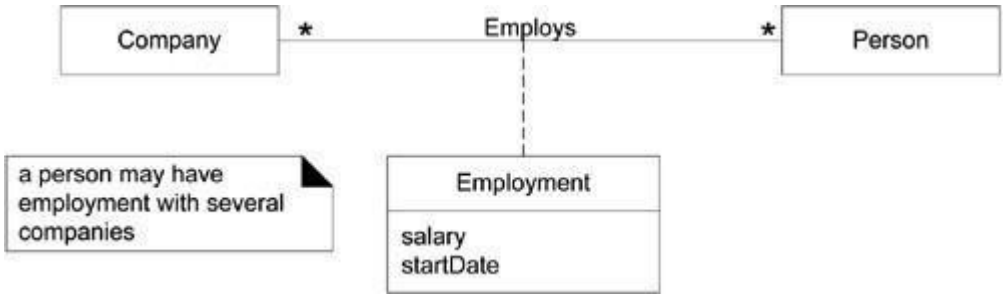
Qualified Association

A **qualified association** has a qualifier that is used to select an object (or objects) from a larger set of related objects, based upon the qualifier key



Association Class

An association class allows you treat an association itself as a class, and model it with attributes, operations, and other features. For example, if a Company employs many Persons, modeled with an Employs association, you can model the association itself as the Employment class, with attributes such as startDate.



ELABORATION

Elaboration is the initial series of iterations during which, on a normal project:

the core, risky software architecture is programmed and tested
the majority of requirements are discovered and stabilized
the major risks are mitigated or retired

Build the core architecture, resolve the high-risk elements, define most requirements, and estimate the overall schedule and resources.

Elaboration is the initial series of iterations during which the team does serious investigation, implements (programs and tests) the core architecture, clarifies most requirements, and tackles the high-risk issues.

Elaboration often consists of two or more iterations; Each iteration is recommended to be between two and six weeks; prefer the shorter versions unless the team size is massive. Each iteration is time boxed, i.e its end date is fixed.

Elaboration is not a design phase or a phase when the models are fully developed in preparation for implementation in the construction step that would be an example of superimposing waterfall ideas on iterative development and the UP.

During this phase, no prototypes are created ; rather, the code and design are production-quality portions of the final system.

Architectural prototype means a production subset of the final system. More commonly it is called the executable architecture or architectural baseline.

Key Ideas and Best Practices will manifest in elaboration:

do short time boxed risk-driven iterations

start programming early

adaptively design, implement, and test the core and risky parts of the architecture

test early, often, realistically

adapt based on feedback from tests, users, developers

write most of the use cases and other requirements in detail, through a series of workshops, once per elaboration iteration

Table -Sample elaboration artifacts, excluding those started in inception.

Artifact	Comment
Domain Model	This is a visualization of the domain concepts; it is similar to a static information model of the domain entities.
Design Model	This is the set of diagrams that describes the logical design. This includes software class diagrams, object interaction diagrams, package diagrams, and so forth.
Software Architecture Document	A learning aid that summarizes the key architectural issues and their resolution in the design. It is a summary of the outstanding design ideas and their motivation in the system.
Data Model	This includes the database schemas, and the mapping strategies between object and non-object representations.
Use-Case Storyboards, UI Prototypes	A description of the user interface, paths of navigation, usability models, and so forth.

Process: Planning the Next Iteration

Organize requirements and iterations by risk, coverage, and criticality.

Risk includes both technical complexity and other factors, such as uncertainty of effort or usability.

Coverage implies that all major parts of the system are at least touched on in early iterations perhaps a "wide and shallow" implementation across many components.

Criticality refers to functions the client considers of high business value.

These criteria are used to rank work across iterations. Use cases or use case scenarios are ranked for implementation early iterations implement high ranking scenarios. The ranking is done before iteration-1, but then again before iteration-2, and so forth, as new requirements and new insights influence the order.

For example:

Rank	Requirement (Use Case or Feature)	Comment
High	Process Sale Logging …	Scores high on all rankings. Pervasive. Hard to add late. …
Medium	Maintain Users …	Affects security sub domain. …
Low	…	…

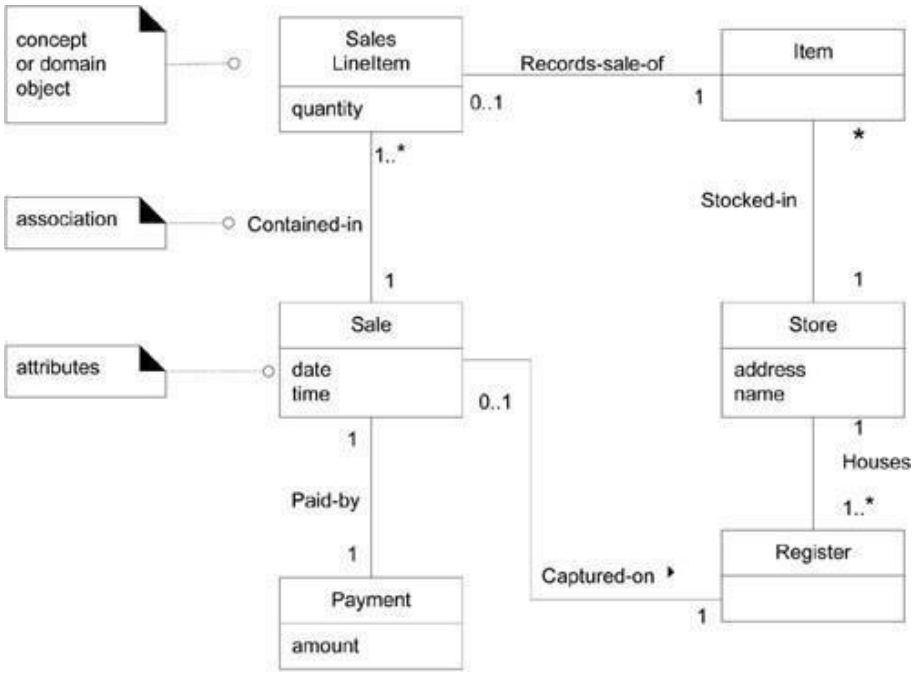
Based on this ranking, we see that some key architecturally significant scenarios of the Process Sale use case should be tackled in early iterations.

DOMAIN MODEL

Domain Models

The figure shows a partial domain model drawn with UML class diagram notation. It illustrates that the conceptual classes 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, information attributes we care about.

Applying the UML class diagram notation for a domain model yields a conceptual perspective model. Identifying a rich set of conceptual classes is at the heart of OO analysis.



What is a Domain Model?

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.

Definition

In the UP, the term "Domain Model" means a representation of real-situation conceptual classes, not of software objects. The term does not mean a set of diagrams describing software classes, the domain layer of a software architecture, or software objects with responsibilities.

A domain model is illustrated with a set of class diagrams in which no operations (method signatures) are defined. It provides a conceptual perspective. It may show:

domain objects or conceptual classes
associations between conceptual classes
attributes of conceptual classes

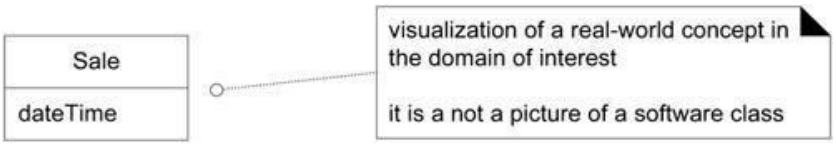
Why Call a Domain Model a "Visual Dictionary"?

Domain Model visualizes and relates words or concepts in the domain. It also shows an abstraction of the conceptual classes, because there are many other things one could communicate about registers, sales, and so forth.

The domain model is a visual dictionary of the noteworthy abstractions, domain vocabulary, and information content of the domain.

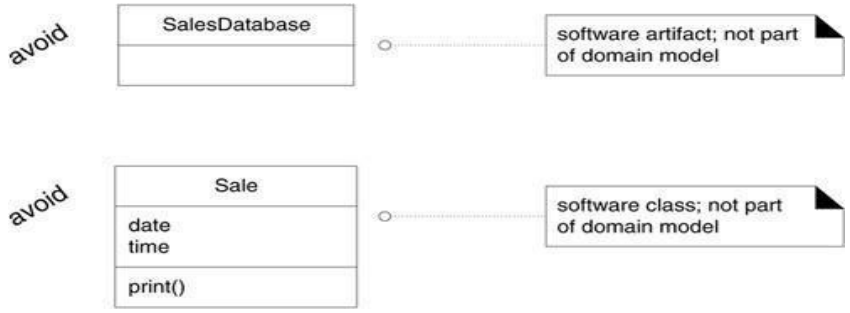
A UP Domain Model is a visualization of things in a real-situation domain of interest, not of software objects such as Java or C# classes, or software objects with responsibilities. Therefore, the following elements are <u>not suitable</u> in a domain model:

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



A domain model shows real –situation conceptual classes, not software classes

.



A domain model does not show software artifacts or classes

<u>Two Traditional Meaning of Domain Model</u>

<u>Meaning 1 :</u>"Domain Model" is a conceptual perspective of objects in a real situation of the world, not a software perspective.

<u>Meaning 2 :</u>"the domain layer of software objects." That is, the layer of software objects below the presentation or UI layer that is composed of domain objects software objects that represent things in the problem domain space with related "business logic" or "domain logic" methods.

CONCEPTUAL CLASSES

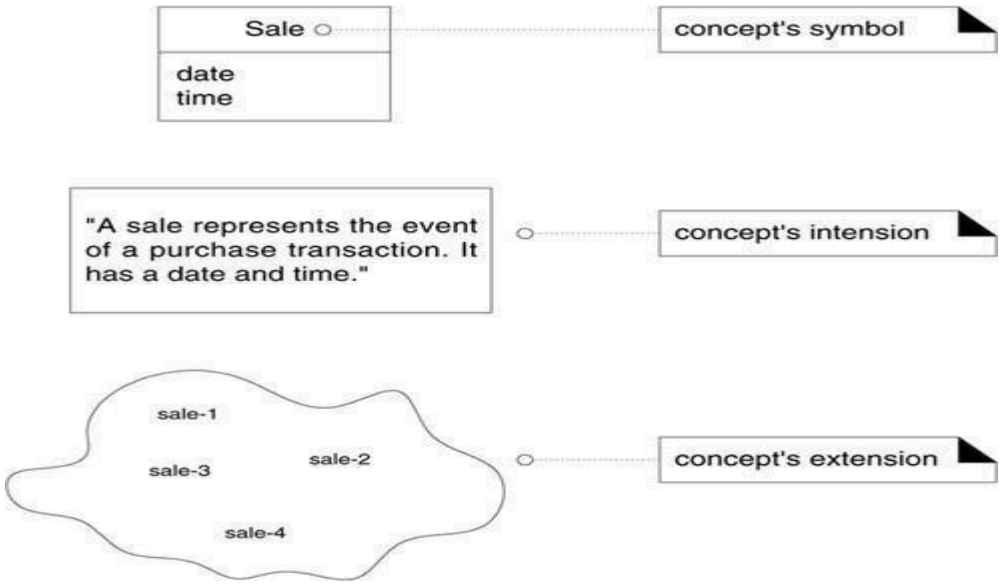
A conceptual class is an idea, thing, or object. It may be considered in terms of its symbol, intension, and extension (see Figure).

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.

For example, consider the conceptual class for the event of a purchase transaction. I may choose to name it by the (English) symbol Sale. The intension of a Sale may state that it "represents the event of a purchase transaction, and has a date and time." The extension of Sale is all the examples of sales; in other words, the set of all sale instances in the universe.

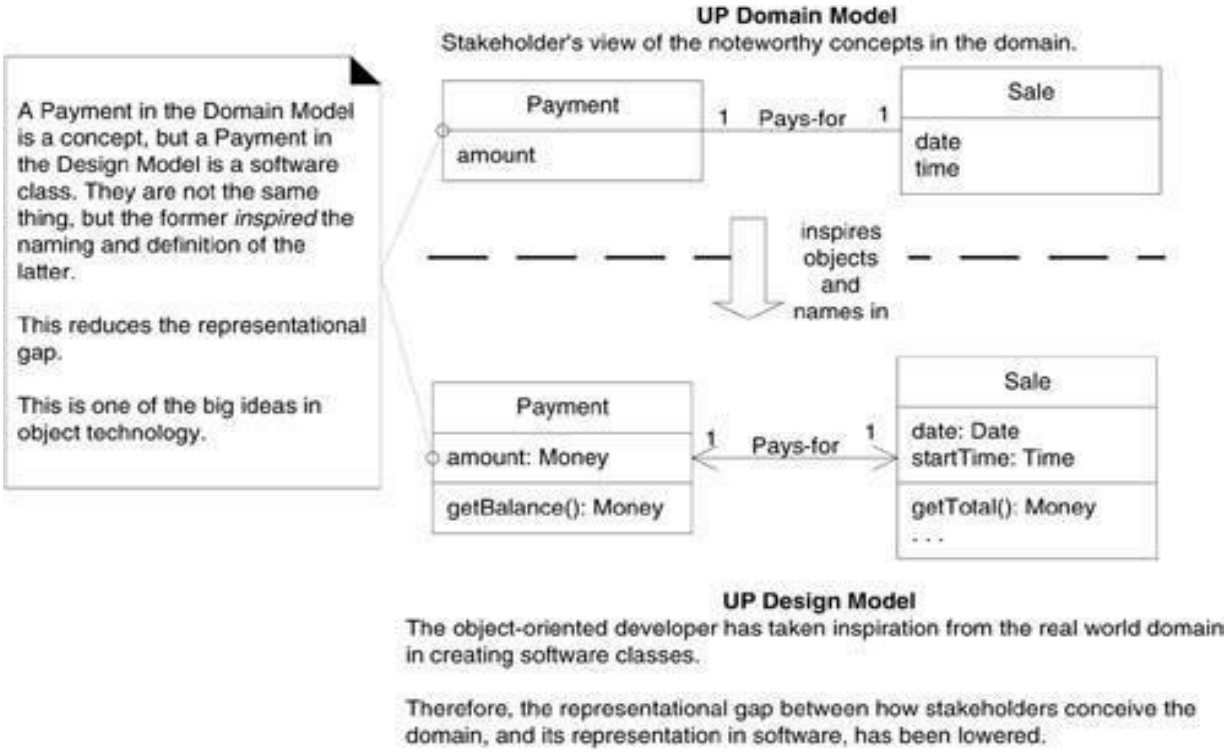


A conceptual class has a symbol, intension and extension Are Domain and Data Models the Same Thing?

A domain model is not a data model (which by definition shows persistent data to be stored somewhere), so do not exclude a class simply because the requirements don't indicate any obvious need to remember information about it or because the conceptual class has no attributes. For example, it's valid to have attribute less conceptual classes, or conceptual classes that have a purely behavioral role in the domain instead of an information role.

Motivation: Why Create a Domain Model ?

Lower Representational Gap with OO Modeling : This is a key idea in OO: Use software class names in the domain layer inspired from names in the domain model, with objects having domain-familiar information and responsibilities. This supports a low representational gap between our mental and software models.



Lower Representational Gap with OO Modeling

Guideline: How to Create a Domain Model?

Bounded by the current iteration requirements under design:

1. Find the conceptual classes (see a following guideline).
2. Draw them as classes in a UML class diagram.
3. Add associations and attributes.

Guideline: To Find Conceptual Classes Three Strategies to Find Conceptual Classes:

1. Reuse or modify existing models. This is the first, best, and usually easiest approach. There are published, well-crafted domain models and data models for many common domains, such as inventory, finance, health, and so forth.
2. Use a category list. (Method 2)
3. Identify noun phrases. (Method 3)

Method 2: Use a Category List

We can create a domain model by making a list of candidate conceptual classes. The guidelines also suggest some priorities in the analysis. Examples are drawn from the 1) POS, 2) Monopoly game 3) airline reservation domains.

Table - Conceptual Class Category List.

Conceptual Class Category	Examples
business transactions Guideline: These are critical (they involve money), so start with transactions.	Sale, Payment Reservation
transaction line items Guideline: Transactions often come with related line items, so consider these next.	SalesLineItem
product or service related to a transaction or transaction line item Guideline: Transactions are for something (a product or service). Consider these next.	Item Flight, Seat, Meal
where is the transaction recorded? Guideline: Important.	Register, Ledger FlightManifest
roles of people or organizations related to the transaction; actors in the use case Guideline: We usually need to know about the parties involved in a transaction.	Cashier, Customer, Store MonopolyPlayer Passenger, Airline
place of transaction; place of service	Store Airport, Plane, Seat
noteworthy events, often with a time or place we need to remember	Sale, Payment Monopoly Game Flight
physical objects Guideline: This is especially relevant when creating device-control software, or simulations.	Item, Register Board, Piece, Die Airplane
descriptions of things	Product Description Flight Description
Guideline: Descriptions are often in a catalog.	Product Catalog Flight Catalog

containers of things (physical or information)	Store, Bin Board Airplane

Table - Conceptual Class Category List.

Conceptual Class Category	Examples
things in a container	Item Square (in a Board) Passenger
other collaborating systems	CreditAuthorizationSystem AirTrafficControl
records of finance, work, contracts, legal matters	Receipt, Ledger Maintenance Log
financial instruments	Cash, Check, LineOfCredit Ticket Credit
schedules, manuals, documents that are regularly referred to in order to perform work	DailyPriceChangeList Repair Schedule

Method 3: Finding Conceptual Classes with Noun Phrase Identification

Another useful technique (because of its simplicity) suggested is linguistic analysis: Identify the nouns and noun phrases in textual descriptions of a domain, and consider them as candidate conceptual classes or attributes.

Guideline

Linguistic analysis has become more sophisticated; it also goes by the name natural language modeling. for example, the current scenario of the Process Sale use case can be used.

Main Success Scenario (or Basic Flow):

1. **Customer** arrives at a **POS checkout** with goods and/or **services** to purchase.
2. **Cashier** starts a new **sale**.
3. **Cashier** enters item **identifier.**
4. System records **sale line item** and presents **item description, price**, and running **total**. Price calculated from a set of price rules.

Underlined words are nouns. The next level of scrutiny derives class names.

Example: Find and Draw Conceptual Classes

Case Study: POS Domain

From the category list and noun phrase analysis, a list is generated of candidate conceptual classes for the domain. There is no such thing as a "correct" list. It is a somewhat arbitrary collection of abstractions and domain vocabulary

Initial POS domain model.

Guidelines

1. **Agile Modeling Sketching a Class Diagram:** The sketching style in the UML class diagram is to keep the bottom and right sides of the class boxes open. This makes it easier to grow the classes as we discover new elements.

2. **Agile Modeling Maintain the Model in a Tool?** The purpose of creating a domain model is to quickly understand and communicate a rough approximation of the key concepts.

3. **Report Objects - Include 'Receipt' in the Model?** Receipt is a term in the POS domain. But it's only a report of a sale and payment, and thus duplicate information.

4. Use Domain Terms:
Make a domain model in the spirit of how a cartographer or mapmaker works:

- Use the existing names in the territory. For example, if developing a model for a library, name the customer a "Borrower" or "Patron" the terms used by the library staff.
- Exclude irrelevant or out-of-scope features. For example, in the Monopoly domain model for iteration-1
- Do not add things that are not there.

5. **How to Model the Unreal World**? Some software systems are for domains

that find very little analogy in natural or business domains; software for

telecommunications is an example. For example, here are candidate conceptual classes related to the domain of a telecommunication switch: Message, Connection, Port, Dialog, Route, and Protocol.

6. **A Common Mistake with Attributes vs. Classes** If we do not think of some conceptual class X as a number or text in the real world, X is probably a conceptual class, not an attribute. As an example, should store be an attribute of Sale, or a separate conceptual class Store?

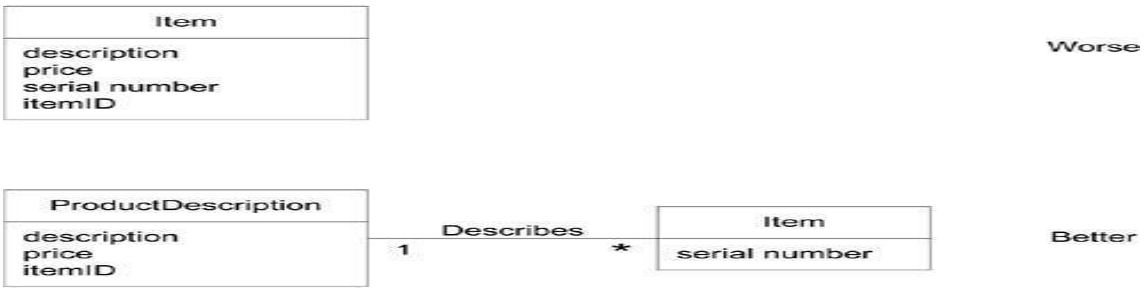
In the real world, a store is not considered a number or text the term suggests a legal entity, an organization, and something that occupies space. Therefore, Store should be a conceptual class.

As another example, consider the domain of airline reservations. Should destination be an attribute of Flight, or a separate conceptual class Airport?

In the real world, a destination airport is not considered a number or text-it is a massive thing that occupies space. Therefore, Airport should be a concept.

7 When to Model with 'Description' Classes? A description class contains information that describes something else. For example, a Product Description that records the price, picture, and text description of an Item.

Motivation: Why Use 'Description' Classes? The need for description classes is common in many domain models. The need for description classes is common in sales, product, and service domains. It is also common in manufacturing, which requires a description of a manufactured thing that is distinct from the thing itself Figure. Descriptions about other things. The * means a multiplicity of "many." It indicates that one Product Description may describe many (*) Items**.**



When Are Description Classes Useful?

Add a description class (for example, Product Description) when:

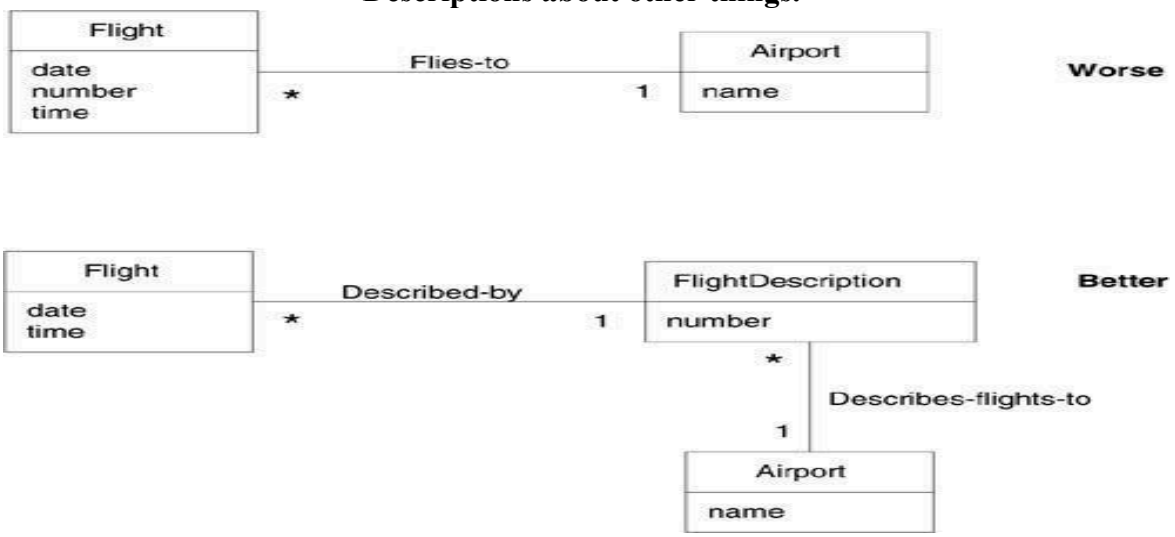
- There needs to be a description about an item or service, independent of the current existence of any examples of those items or services.
- Deleting instances of things, they describe (for example, Item) results in a loss of information that needs to be maintained, but was incorrectly associated with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

Example: Descriptions in the Airline Domain

As another example, consider an airline company that suffers a fatal crash of one of its planes. Assume that all the flights are cancelled for six months pending completion of an investigation. Also assume that when flights are cancelled, their corresponding Flight software objects are deleted from computer memory. Therefore, after the crash, all Flight software objects are deleted.

If the only record of what airport a flight goes to is in the Flight software instances, which represent specific flights for a particular date and time, then there is no longer a record of what flight routes the airline has The problem can be solved, both from a purely conceptual perspective in a domain model and from a software perspective in the software designs, with a Flight Description that describes a flight and its route, even when a particular flight is not scheduled in following figure

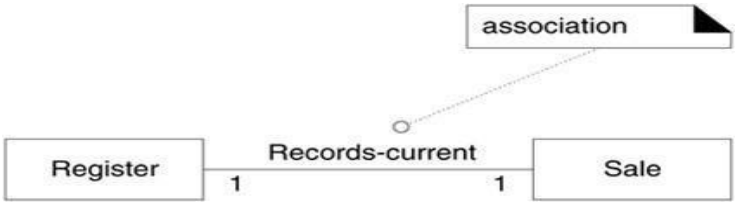
Descriptions about other things.



Note that the prior example is about a service (a flight) rather than a good. Descriptions of services or service plans are commonly needed.

Associations

An **association** is a relationship between classes (more precisely, instances of those classes) that indicates some meaningful and interesting connection (see Figure)



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

Include the following associations in a domain model:

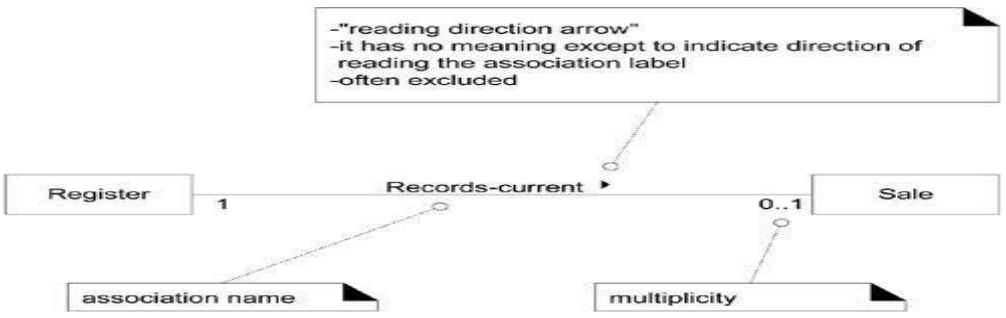
- Associations for which knowledge of the relationship needs to be preserved for some duration ("need-to-remember" associations).
- Associations derived from the Common Associations List.

Guideline 1. Avoid Adding Many Associations

- We need to avoid adding too many associations to a domain model. In a graph with n nodes, there can be $(n \cdot (n-1))/2$ associations to other nodes-a potentially very large number. A domain model with 20 classes could have 190 associations lines!
- During domain modeling, an association is not a statement about data flows, database foreign key relationships, instance variables, or object connections in a software solution; it is a statement that a relationship is meaningful in a purely conceptual perspective-in the real domain.

Applying UML: Association Notation

An association is represented as a line between classes with a capitalized association name. See Figure



The UML notation for associations.

The ends of an association may contain a multiplicity expression indicating the numerical relationship between instances of the classes.

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.

An optional "reading direction arrow" indicates the direction to read the association name; it does not indicate direction of visibility or navigation. If the arrow is not present, the convention is to read the association from left to right or top to bottom.

Guideline 2: To Name an Association in UML

Name an association based on a ClassName-VerbPhrase - ClassName format where the verb phrase creates a sequence that is readable and meaningful. Simple association names such as "Has" or "Uses" are usually poor, as they seldom enhance our understanding of the domain.

For example,

- Sale Paid-by CashPayment
    - bad example (doesn't enhance meaning): Sale Uses CashPayment
- Player Is-on Square
    - bad example (doesn't enhance meaning): Player Has Square

Association names should start with a capital letter, since an association represents a classifier of links between instances; in the UML, classifiers should start with a capital letter.

Applying UML: Roles

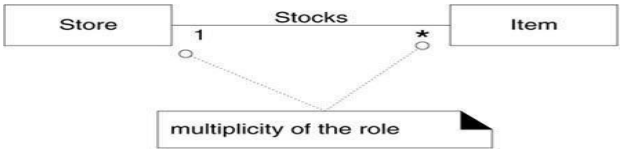
Each end of an association is called a **role**. Roles may optionally have:

- multiplicity expression
- name
- navigability

Applying UML: Multiplicity

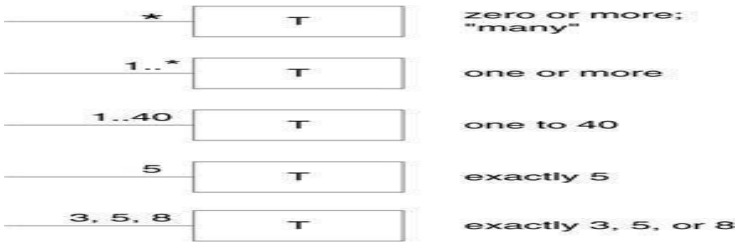
Multiplicity defines how many instances of a class A can be associated with one instance of a class B

Multiplicity on an association.



For example, a single instance of a Store can be associated with "many" (zero or more, indicated by the *) Item instances.

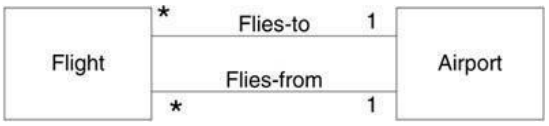
Multiplicity values.



Applying UML: Multiple Associations Between Two Classes

The domain of the airline is the relationships between a Flight and an Airport the flying-to and flying-from associations are distinctly different relationships, which should be shown separately.

Multiple associations.



Guideline 3 : To Find Associations with a Common Associations List

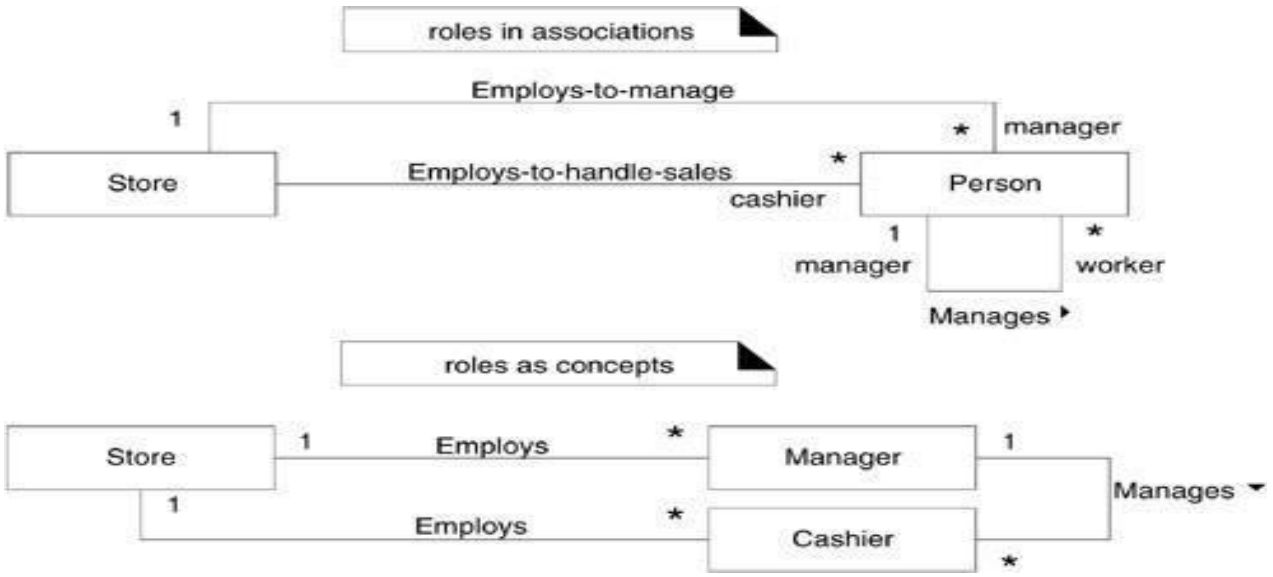
Start the addition of associations by using the list in Table . It contains common categories that are worth considering, especially for business information systems. Examples are drawn from the 1) POS, 2) Monopoly, and 3) airline reservation domains.

Table - Common Associations List.

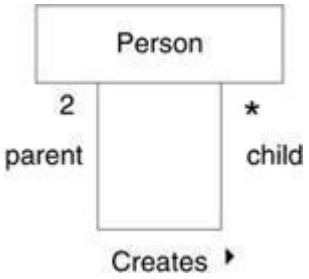
Category	Examples
A is a transaction related to another transaction B	CashPaymentSale CancellationReservation
A is a line item of a transaction B	SalesLineItemSale

Category	Examples
A is a product or service for a transaction (or line item) B	ItemSalesLineItem(or Sale)FlightReservation
A is a role related to a transaction B	CustomerPayment PassengerTicket
A is a physical or logical part of B	DrawerRegister SquareBoard SeatAirplane
A is physically or logically contained in/on B	RegisterStore, ItemShelf SquareBoard PassengerAirplane
A is a description for B	ProductDescriptionItem FlightDescriptionFlight
A is known / logged / recorded / reported / captured in B	SaleRegister PieceSquare ReservationFlightManifest
A is a member of B	CashierStore PlayerMonopolyGame PilotAirline
A is an organizational subunit of B	DepartmentStore MaintenanceAirline
A uses or manages or owns B	CashierRegister PlayerPiece PilotAirplane
A is next to B	SalesLineItemSalesLineItem SquareSquare CityCity

Roles as Concepts versus Roles in Associations: In a domain model, a real-world role especially a human role may be modeled in a number of ways, such as a discrete concept, or expressed as a role in an association. For example, the role of cashier and manager may be expressed in at least the two ways illustrated in Fig



Reflexive Associations: A concept may have an association to itself; this is known as a reflexive association

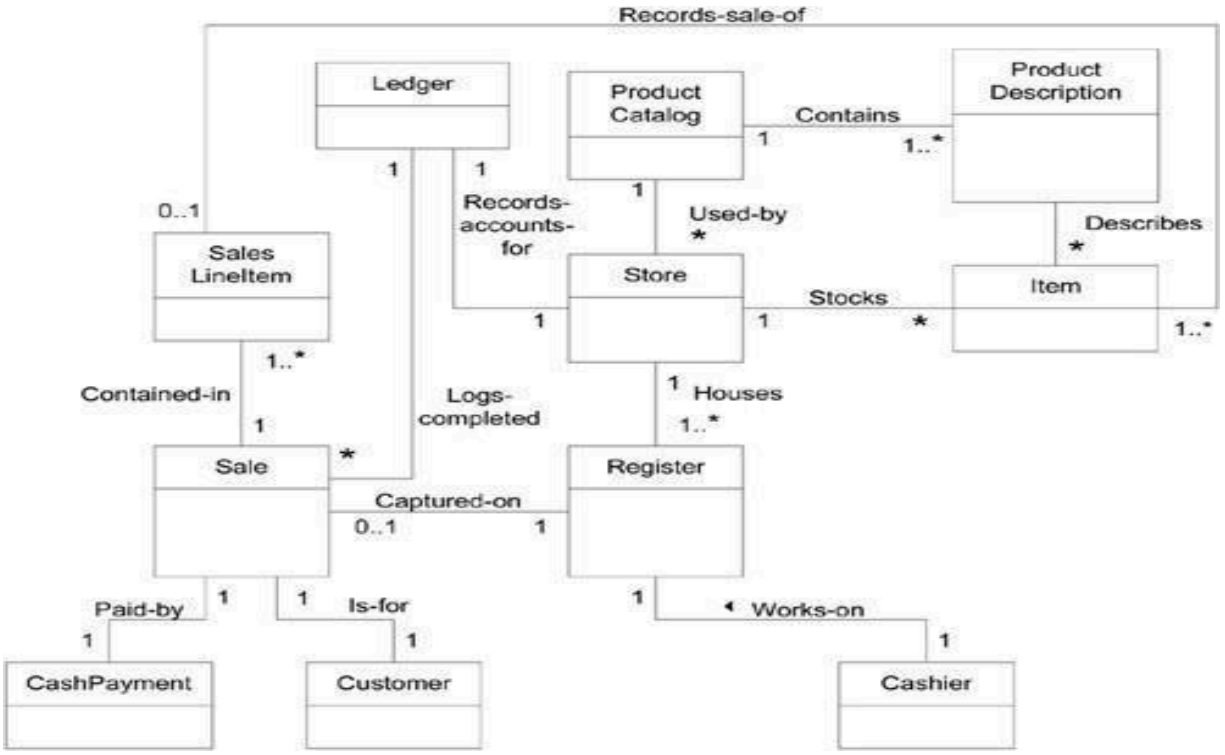


Example: Associations in the Domain Models

Case Study: NextGen POS: The domain model in Figure shows a set of conceptual classes and associations that are candidates for our POS domain model. The associations are primarily derived from the "need-to-remember" criteria of these iteration requirements, and the Common Association List. For example:

- Transactions related to another transaction Sale Paid-by Cash Payment.
- Line items of a transaction Sale Contains SalesLineItem.
- Product for a transaction (or line item) SalesLineItem Records-sale-of Item.

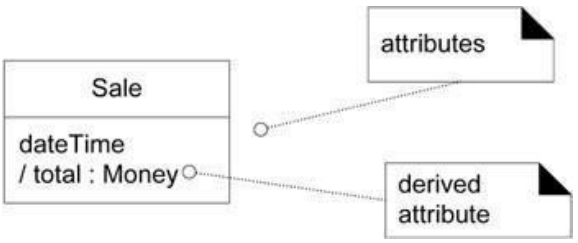
NextGen POS partial domain model.



Attributes: An **attribute** is a logical data value of an object. Include attributes that the requirements (for example, use cases) suggest or imply a need to remember information. For example, a receipt (which reports the information of a sale) in the Process Sale use case normally includes Therefore,

- Sale needs a date Time attribute.
- Store needs a name and address.
- Cashier needs an ID.

Applying UML- Attribute Notation: Attributes are shown in the second compartment of the class box. Their type and other information may optionally be shown.



Class and attributes.

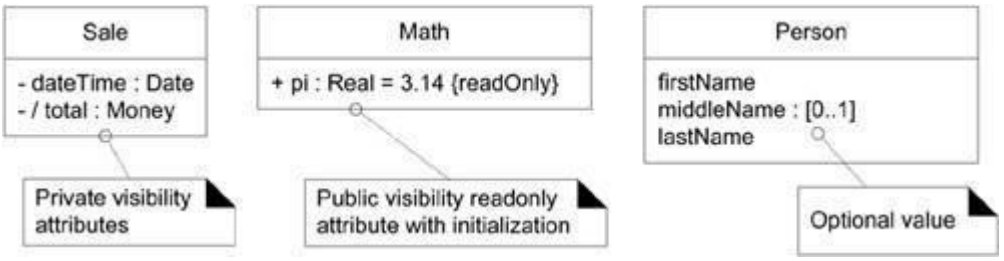
More Notation

The full syntax for an attribute in the UML is:

visibility name: type multiplicity = default {property-string}

Some common examples are shown in Fig

Attribute notation in UML.



{readOnly} is probably the most common property string for attributes.

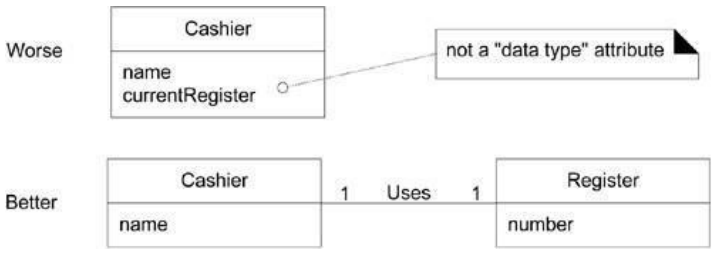
Multiplicity can be used to indicate the optional presence of a value, or the number of objects that can fill a (collection) attribute.

Derived Attributes: When we want to communicate that 1) this is a noteworthy attribute, but 2) it is derivable, we use the UML convention: a / symbol before the attribute name.

Guideline 1: Suitable Attribute Types - Focus on Data Type Attributes in the Domain Model

Most attribute types should be what are often thought of as "primitive" data types, such as numbers and Booleans. For example, the current Register attribute in the Cashier class in Figure is undesirable because its type is meant to be a Register, which is not a simple data type (such as Number or String).

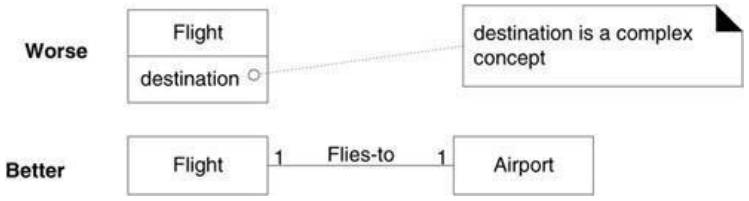
Relate with associations, not attributes



Guideline: The attributes in a domain model should preferably be data types. Very common data types include: Boolean, Date (or Date Time), Number, Character, String (Text), Time. Other common types include: Address, Color, Geometrics (Point, Rectangle), Phone Number, Social Security Number, Universal Product Code (UPC), SKU, ZIP or postal codes, enumerated types

A common confusion is modeling a complex domain concept as an attribute. To illustrate, a destination airport is not really a string; it is a complex thing that occupies many square kilometers of space. Therefore, Flight should be related to Airport via an association, not with an attribute, as shown in Fig.

Don't show complex concepts as attributes; use associations.



Guideline : Relate conceptual classes with an association, not with an attribute.

Data Types

Attributes in the domain model should generally be data types; informally these are "primitive" types such as number, boolean, character, string, and enumerations (such as Size = {small, large}).

For example, it is not (usually) meaningful to distinguish between:

- Separate instances of the Integer 5.
- Separate instances of the String 'cat'.
- Separate instance of the Date "Nov. 13, 1990".

Guideline 1: When to define New Data type Classes?

Guidelines for modeling data types

Represent what may initially be considered a number or string as a new data type class in the domain model if:

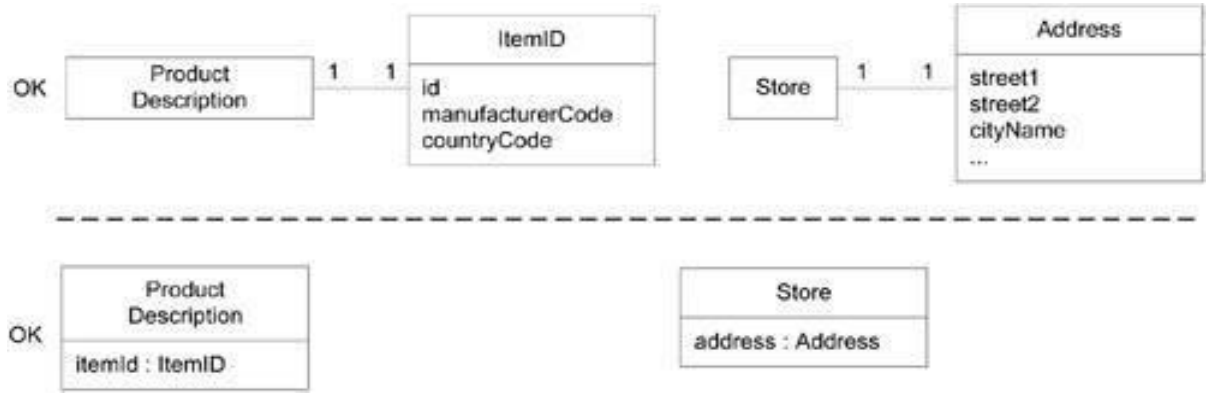
- It is composed of separate sections. –Ex phone number, name of person
- There are operations associated with it, such as parsing or validation. - social security number
- It has other attributes. - promotional price could have a start (effective) date and end date
- It is a quantity with a unit. - payment amount has a unit of currency
- It is an abstraction of one or more types with some of these qualities.

Item identifier in the sales domain is a generalization of types such as Universal Product Code (UPC) and European Article Number (EAN)

Applying these guidelines to the POS domain model attributes yields the following analysis:

- The item identifier is an abstraction of various common coding schemes, including UPC-A, UPC-E, and the family of EAN schemes. These numeric coding schemes have subparts identifying the manufacturer, product, country (for EAN), and a check-sum digit for validation. Therefore, there should be a data type ItemID class, because it satisfies many of the guidelines above.
- The price and amount attributes should be a data type Money class because they are quantities in a unit of currency.
- The address attribute should be a data type Address class because it has separate sections.

Applying UML: Where to Illustrate These Data Type Classes?



Two ways to indicate a data type property of an object.

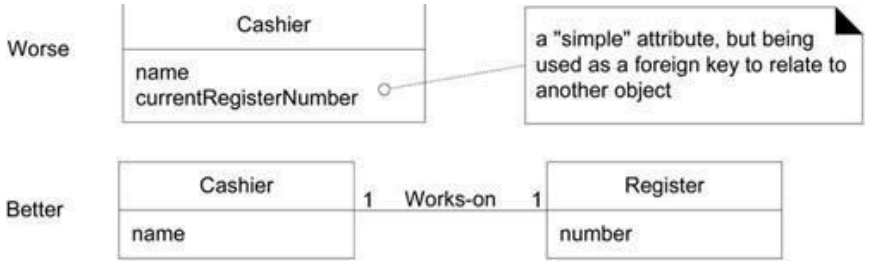
Should the ItemID class be shown as a separate class in a domain model? Since ItemID is a data type (unique identity of instances is not used for equality testing), it may be shown only in the attribute compartment of the class box, as shown in above Figure. On the other hand, if ItemID is a new type with its own attributes and associations, showing it as a conceptual class in its own box may be informative.

Guideline 2: No Attributes Representing Foreign Keys

In Following Fig, the currentRegisterNumber attribute in the Cashier class is undesirable because its purpose is to relate the Cashier to a Register object. The

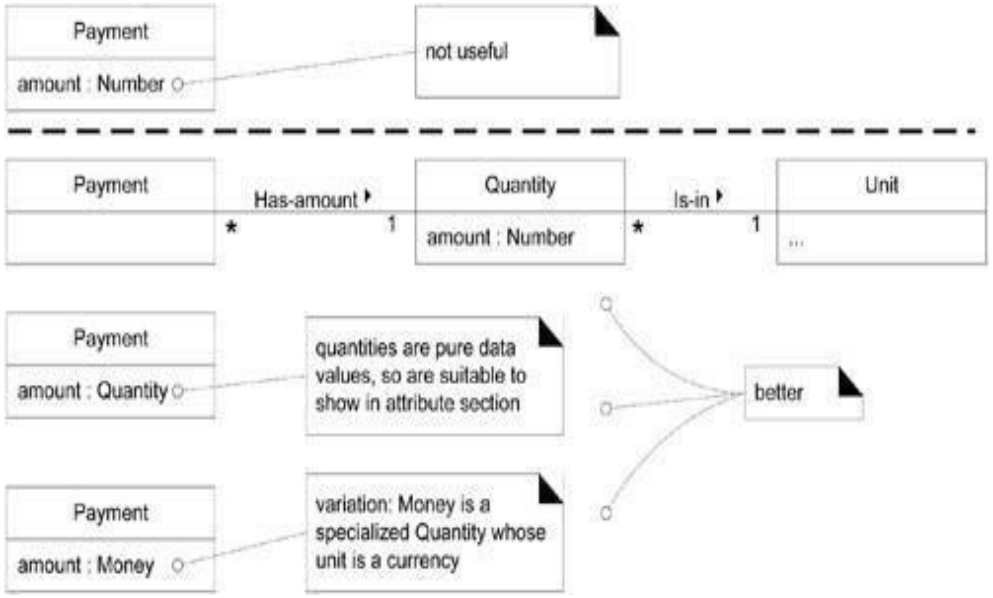
better way to express that a Cashier uses a Register is with an association, not with a foreign key attribute.

Do not use attributes as foreign keys.



Guideline 3 : Modeling Quantities and Units

Most numeric quantities should not be represented as plain numbers. Consider price or weight. These are quantities with associated units, and it is common to require knowledge of the unit to support conversions.



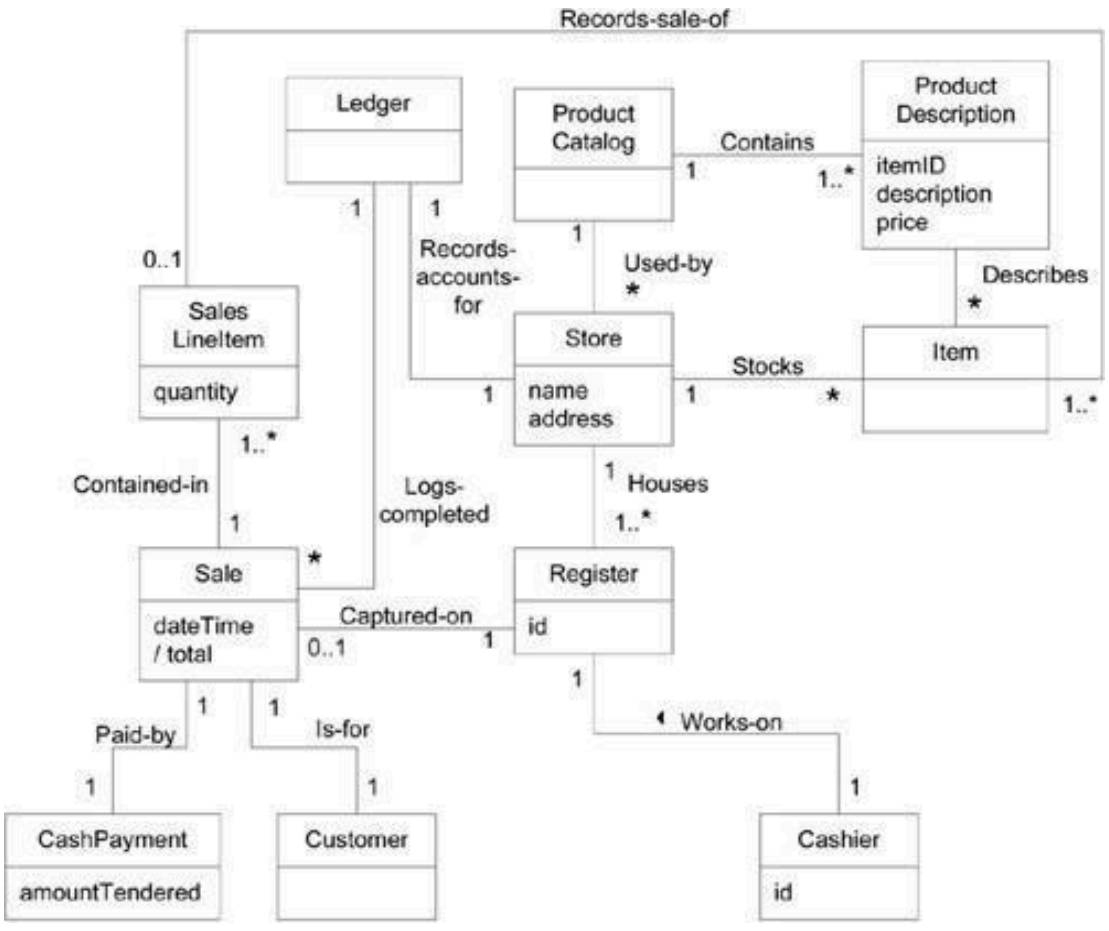
Modeling quantities.

Example: Attributes in the Domain Models -Case Study: NextGen POS

See following Fig. The attributes chosen reflect the information requirements for this iteration the Process Sale cash-only scenarios of this iteration. For example:

CashPayment	amountTendered to determine if sufficient payment was provided, and to calculate change, an amount (also known as "amount tendered") must be captured.

Cash Payment	amount Tendered to determine if sufficient payment was provided, and to calculate change, an amount (also known as "amount tendered") must be captured.
Product-Description	description to show the description on a display or receipt. itemId to look up a Product Description. price to calculate the sales total, and show the line-item price.
Sale	date Time A receipt normally shows date and time of sale, and this is useful for sales analysis.
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.



NextGen POS partial domain model.

DOMAIN MODEL REFINEMENT

OBJECTIVES

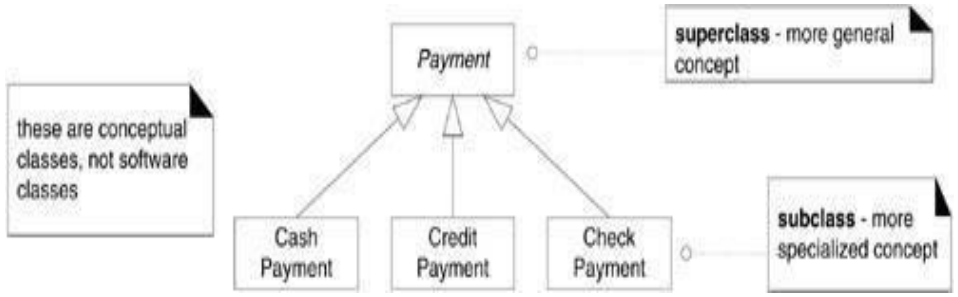
- Refine the domain model with generalizations, specializations, association classes, time intervals, composition, and packages.
- Generalization and specialization are fundamental concepts in domain modeling that support an economy of expression;
- Association classes capture information about an association itself.
- Time intervals capture the important concept that some business objects are valid for a limited time.
- Packages are a way to organize large domain models into smaller units.

Concepts Category List : This Table shows some concepts being considered in this iteration.

Category	Examples
physical or tangible objects	CreditCard, Check
Transactions	CashPayment, CreditPayment, CheckPayment
other computer or electro-mechanical systems external to our system	CreditAuthorizationService, CheckAuthorizationService
abstract noun concepts	
Organizations	CreditAuthorizationService, CheckAuthorizationService
records of finance, work, contracts, legal matters	AccountsReceivable

Generalization

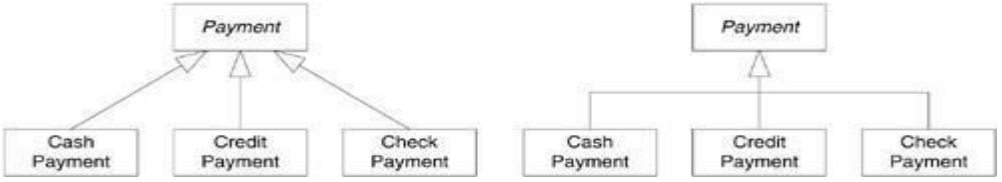
The concepts CashPayment, CreditPayment, and CheckPayment are all very similar. In this situation, it is possible (and useful) to organize them (as in following Figure) into a generalization-specialization class hierarchy (or simply **class hierarchy**) in which the **super class** Payment represents a more general concept, and the **subclasses** more specialized ones.



Generalization-specialization hierarchy.

Generalization is the activity of identifying commonality among concepts and defining superclass (general concept) and subclass (specialized concept) relationships. 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.

Guideline : Identify domain superclasses and subclasses relevant to the current iteration, and illustrate them in the Domain Model.



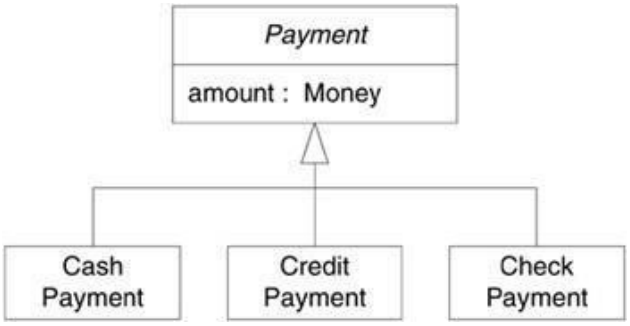
Class hierarchy with separate and shared arrow notations.

Defining Conceptual Super classes and Subclasses :

Definition *:* A conceptual super class definition is more general or encompassing than a subclass definition.

For example, consider the superclass Payment and its subclasses (Cash Payment, 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 to another, and that all payments have an amount of money transferred. The model corresponding to this is shown in following Figure.

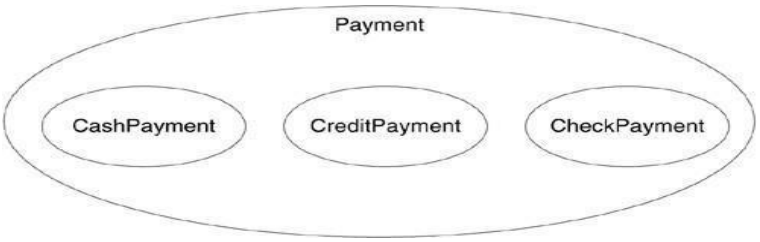
Payment class hierarchy.



A Credit Payment is a transfer of money via a credit institution which needs to be authorized. My definition of Payment encompasses and is more general than my definition of Credit Payment.

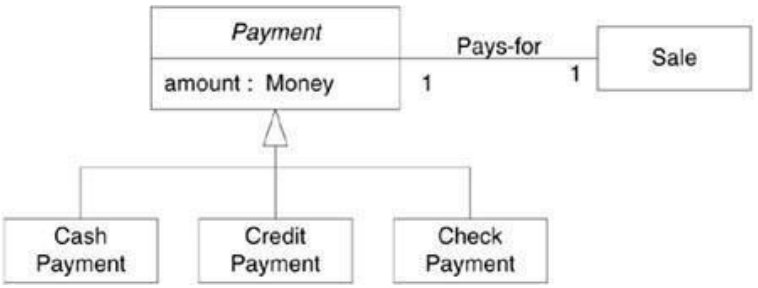
Definition: All members of a conceptual subclass set are members of their superclass set. For example, in terms of set membership, all instances of the set Credit Payment are also members of the set Payment. In a Venn diagram, this is shown as in following Fig

Venn diagram of set relationships.



Conceptual Subclass Definition Conformance: When a class hierarchy is created, statements about super classes that apply to subclasses are made. For example, the following Figure states that all Payments have an amount and are associated with a Sale.

Subclass conformance.



Guideline: <u>100% Rule</u>

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

- attributes
- associations

Conceptual Subclass Set Conformance: A conceptual subclass should be a member of the set of the superclass. Thus, Credit Payment should be a member of the set of Payments.

Guideline: <u>Is-a Rule</u>

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.

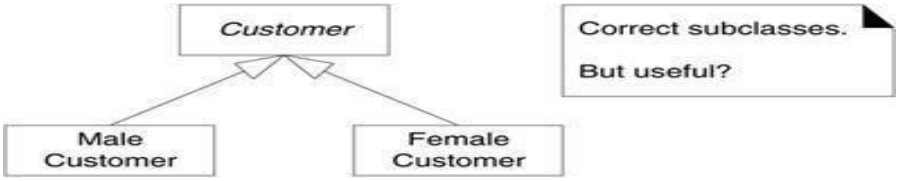
Guideline: Correct Conceptual Subclass

A potential subclass should conform to the:

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

When to Define a Conceptual Subclass?

Definition: A conceptual class partition is a division of a conceptual class into disjoint subclasses. For example, in the POS domain, Customer may be correctly partitioned (or subclassed) into Male Customer and Female Customer. But is it relevant or useful to show this in our model (see following figure)? This partition is not useful for our domain; the next section explains why



Legal conceptual class partition, but is it useful in our domain

Motivations to Partition a Conceptual Class into Subclasses

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.
4. The subclass concept represents an animate thing (for example, animal, robot) that behaves differently than the superclass or other subclasses, in ways that are of interest.

Based on the above criteria, it is not compelling to partition Customer into the subclasses Male Customer and Female Customer because they have no additional attributes or associations, are not operated on (treated) differently, and do not behave differently in ways that are of interest . This table shows some examples of class partitions from the domain of payments and other areas, using these criteria

Example subclass partitions

Conceptual Subclass Motivation	Examples
The subclass has additional attributes of interest.	Payments not applicable.Library Book, subclass of LoanableResource, has an ISBN attribute.

Conceptual Subclass Motivation	Examples
The subclass has additional associations of interest.	Payments CreditPayment, subclass of Payment, is associated with a CreditCard. Library Video, subclass of LoanableResource, is associated with Director.
The subclass concept is operated upon, handled, reacted to, or manipulated differently than the superclass or other subclasses, in ways that are of interest.	Payments CreditPayment, subclass of Payment, is handled differently than other kinds of payments in how it is authorized. Library Software, subclass of LoanableResource, requires a deposit before it may be loaned.
The subclass concept represents an animate thing (for example, animal, robot) that behaves differently than the superclass or other subclasses, in ways that are of interest.	Payments not applicable. Library not applicable. Market Research MaleHuman, subclass of Human, behaves differently than FemaleHuman with respect to shopping habits.

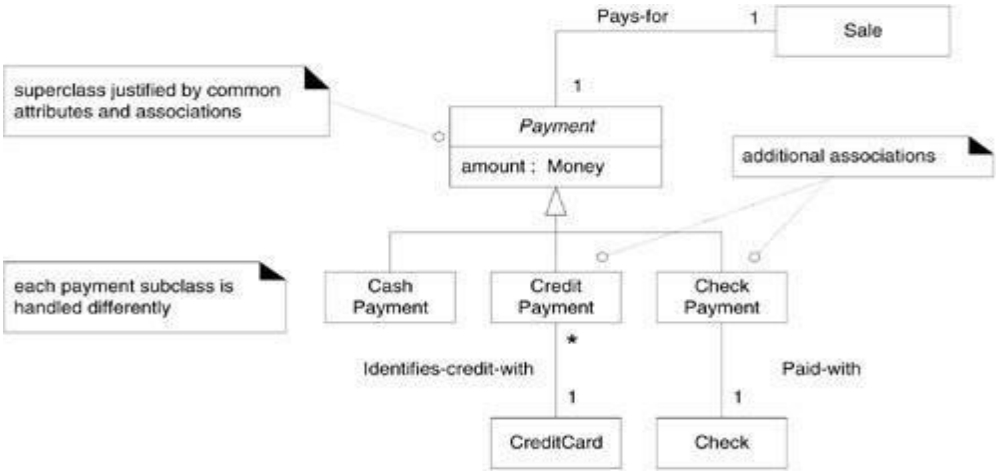
When to Define a Conceptual Superclass?

Motivations to generalize and define a superclass: Guideline

Create a 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 that can be factored out and expressed in the superclass.
- All subclasses have the same association that can be factored out and related to the superclass.

NextGen POS Conceptual Class Hierarchies

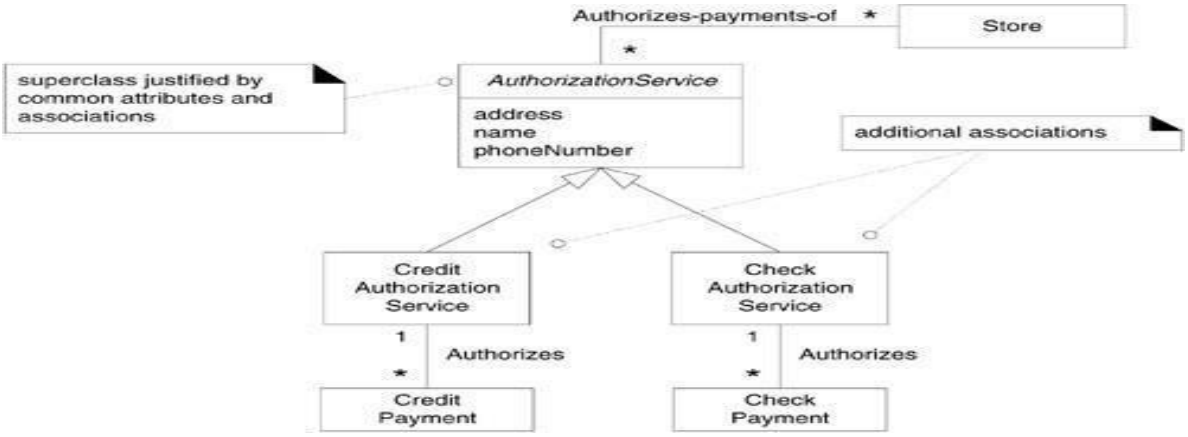


Justifying Payment subclasses.

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.

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 following Figure.

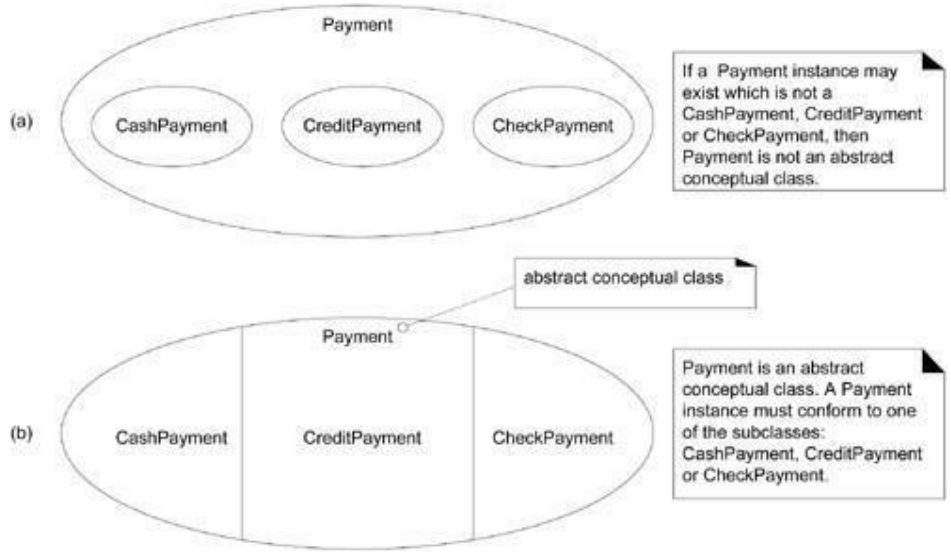
Justifying the Authorization Service hierarchy



Abstract Conceptual Classes

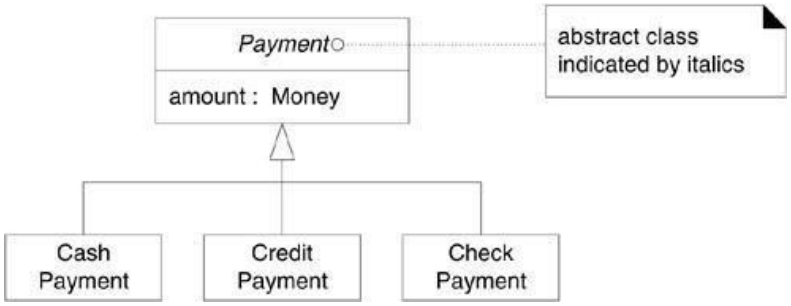
Definition: If every member of a class C must also be a member of a subclass, then class C is called an abstract conceptual class. For example, assume that every Payment instance must more specifically be an instance of the subclass Credit Payment, Cash Payment, or Check Payment. This is illustrated in the Venn diagram of Figure (b). Since every Payment member is also a member of a subclass, Payment is an abstract conceptual class by definition.

Abstract conceptual classes.



Abstract Class Notation in the UML: To review, the UML provides a notation to indicate abstract classes the class name is italicized

Abstract class notation.



Guideline : Identify abstract classes and illustrate them with an italicized name in the Domain Model, or use the {abstract} keyword.

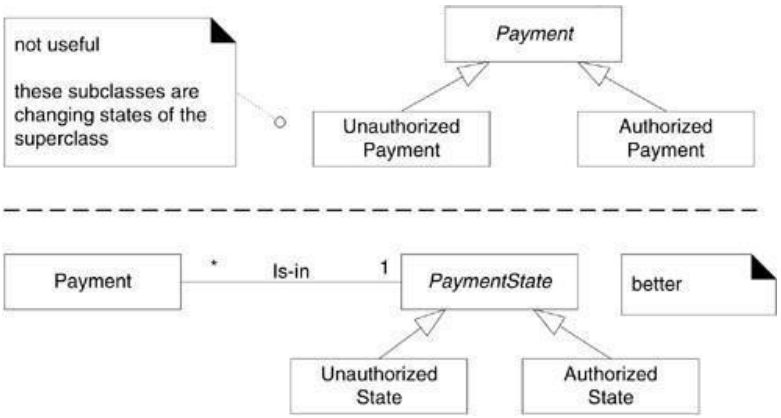
Modeling Changing States

Assume that a payment can either be in an unauthorized or authorized state, and it is meaningful to show this in the domain model. As shown in Figure, one modeling approach is to define subclasses of Payment: Unauthorized Payment and Authorized Payment.

Guideline: Do not model the states of a concept X as subclasses of X. Rather, either:

- Define a state hierarchy and associate the states with X, or
- Ignore showing the states of a concept in the domain model; show the states in state -diagrams instead.

Modeling changing states.



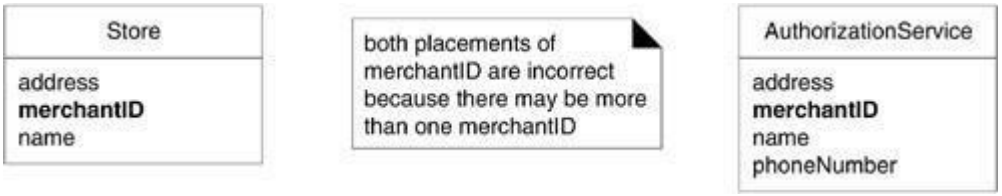
Association Classes

The following domain requirements set the stage for association classes:

- Authorization services assign a merchant ID to each store for identification during communications.
- A payment authorization request from the store to an authorization service needs the merchant ID that identifies the store to the service.
- Furthermore, a store has a different merchant ID for each service.

Placing merchantID in Store is incorrect because a Store can have more than one value for merchantID. The same is true with placing it in Authorization Service (see Figure).

Inappropriate use of an attribute.



Guideline: In a domain model, if a class C can simultaneously have many values for the same kind of attribute A, do not place attribute A in C. Place attribute A in another class that is associated with C.

For example:

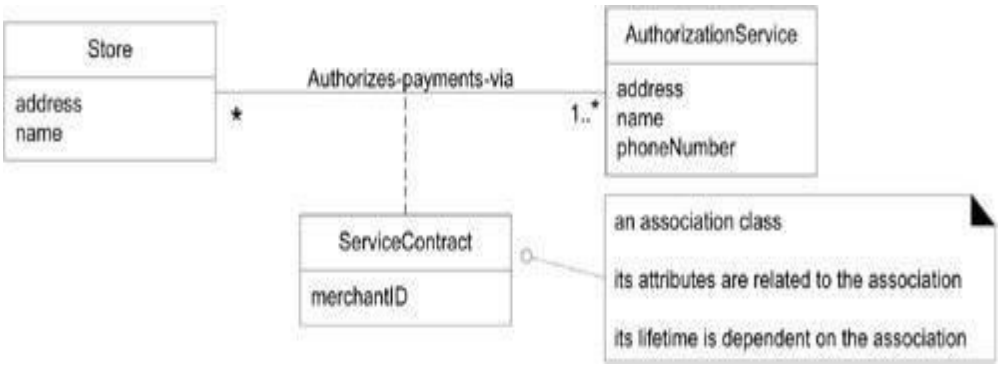
- A Person may have many phone numbers. Place phone number in another class, such as Phone Number or Contact Information, and associate many of these to Person.

First attempt at modeling the merchantID problem.

The fact that both Store and Authorization Service are related to Service Contract is a clue that it is dependent on the relationship between the two. The merchantID may be thought of as an attribute related to the association between Store and Authorization Service.

This leads to the notion of an association class, in which we can add features to the association itself. Service Contract may be modeled as an association class related to the association between Store and Authorization Service.

.



An association class

Guideline: Clues that an association class might be useful in a domain model:

- An attribute is related to an association.
- Instances of the association class have a lifetime dependency on the association.
- There is a many-to-many association between two concepts and information associated with the association itself.

AGGREGATION AND COMPOSITION

How to Identify Composition: Guideline

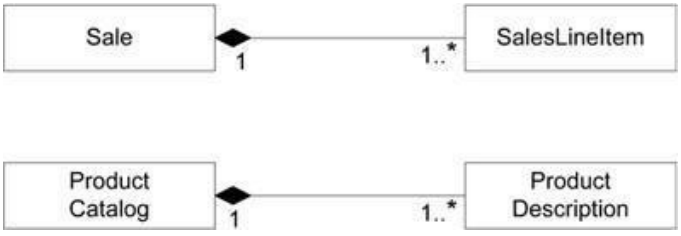
Consider showing composition when:

- 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.

Composition in the NextGen Domain Model

In the POS domain, the SalesLineItems may be considered a part of a composite Sale;

Aggregation in the point-of-sale application.



SYSTEM SEQUENCE DIAGRAMS

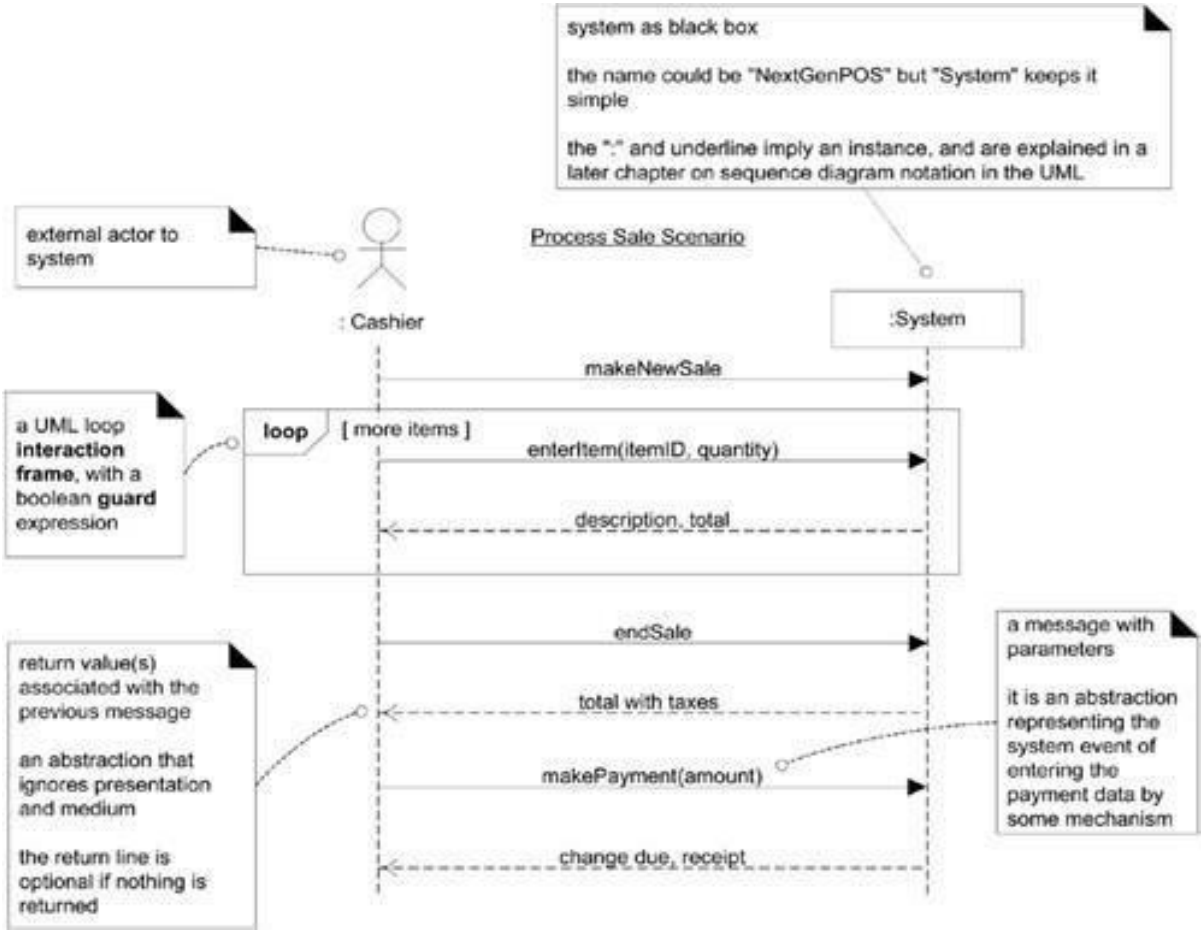
Use cases describe how external actors interact with the software system we are interested in creating. During this interaction an actor generates system events to a system, usually requesting some system operation to handle the event.

For example, when a cashier enters an item's ID, the cashier is requesting the POS system to record that item's sale (the enter Item event). That event initiates an operation upon the system. The use case text implies the enter Item event, and the SSD makes it concrete and explicit.

A system sequence diagram is a picture that shows, for one particular scenario of a use case, the events that external actors generate their order, and inter-system events. All systems are treated as a black box.

Guideline: Draw an SSD for a main success scenario of each use case, and frequent or complex alternative scenarios.

SSD for a Process Sale scenario.



Why Draw an SSD?

A software system reacts to three things:

1) external events from actors (humans or computers),
2) timer events,
3) faults or exceptions (which are often from external sources).

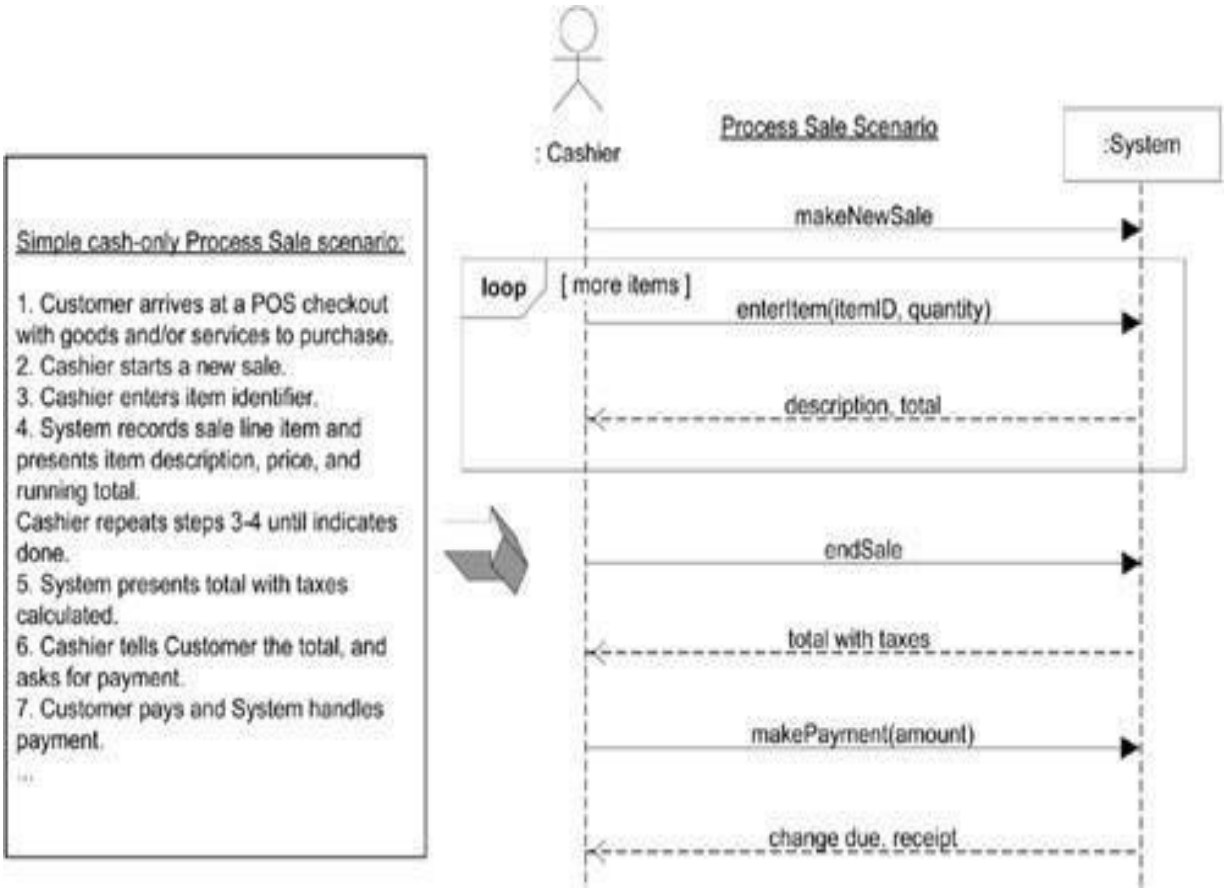
Therefore, it is useful to know what, precisely, are the external input events the system events. They are an important part of analyzing system behavior.

System behavior is a description of what a system does, without explaining how it does it. One part of that description is a system sequence diagram.

RELATIONSHIP BETWEEN SSDS AND USE CASES

An SSD shows system events for one scenario of a use case; therefore, it is generated from inspection of a use case (see Figure below).

SSDs are derived from use cases; they show one scenario.



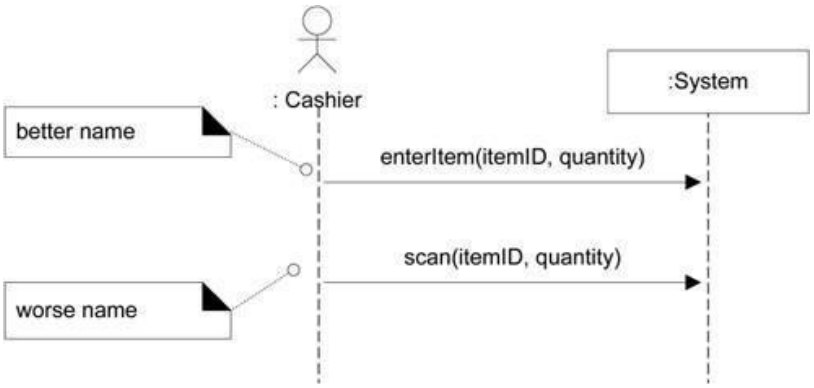
How to Name System Events and Operations?

Which is better, scan(itemID) or enterItem(itemID)?

System events should be expressed at the abstract level of intention rather than in terms of the physical input device.

Thus "enterItem" is better than "scan" (that is, laser scan) because it captures the intent of the operation while remaining abstract and noncommittal with respect to design choices about what interface is used to capture the system event.

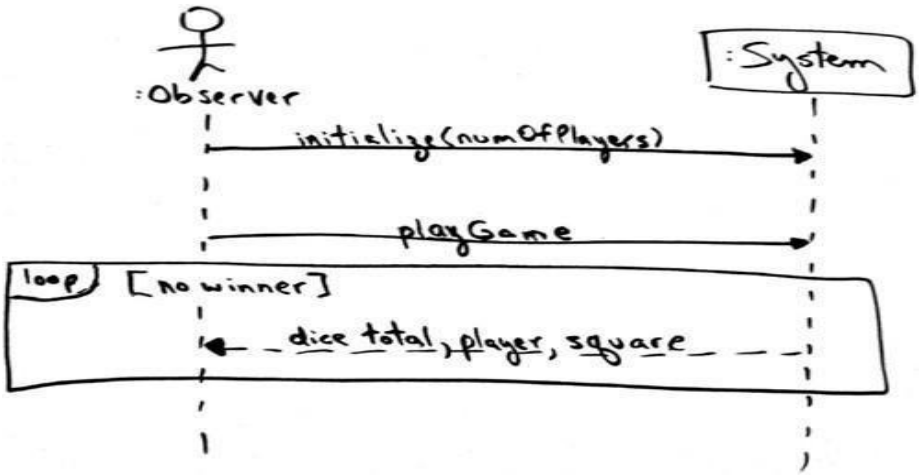
Choose event and operation names at an abstract level.



Example: Monopoly SSD

The Play Monopoly Game use case is simple, as is the main scenario. The observing person initializes with the number of players, and then requests the simulation of play, watching a trace of the output until there is a winner.

SSD for a Play Monopoly Game scenario.



Process:

Draw SSDs only for the scenarios chosen for the next iteration. Don't create SSDs for all scenarios, unless you are using an estimation technique that requires identification of all system operations.

WHEN TO USE CLASS DIAGRAMS

Class diagram is a static diagram and it is used to model the static view of a system. The static view describes the vocabulary of the system.

Class diagram is also considered as the foundation for component and deployment diagrams. Class diagrams are not only used to visualize the static view of the system

but they are also used to construct the executable code for forward and reverse engineering of any system.

Class diagram clearly shows the mapping with object-oriented languages such as Java, C++, etc. From practical experience, class diagram is generally used for construction purpose.

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.
- Forward and reverse engineering.

Ex: Order Processing System