OBJECT ORIENTED ANALYSIS

Analysis emphasizes on **investigation of the problem** rather than how a solution is defined. “What the problem is about and what the system must do?”

Design emphasizes on a **logical solution**, how the system fulfill the requirements.

In Object Oriented analysis, the main emphasis is on **finding and describing the objects** or concepts in the problem domain. e.g. in Library system some concepts may include book, library.

Requirements Process:

Capabilities and conditions to which the system and more broadly, **the project must conform**

A prime challenge of requirements work (fact finding) is to find, communicate, and **record what is really needed**, in a form that clearly speaks to the client and development team members.

Types of requirements:

i. Functional (behavioral)

- calculations, technical details, data manipulation and processing and other **specific functionality**

ii. Non-functional (everything else);

- also known as **quality requirements**, which impose constraints on the design or implementation (such as performance requirements, security, cost and reliability).

Types of Requirements: FURPS+

An acronym representing a model for classifying software quality attributes (functional & non-functional requirements), First Developed at HP

Functionality—features, capabilities, security.

Usability—human factors, help, documentation.

Reliability—frequency of failure, recoverability, predictability

Performance—response times, throughput, accuracy, availability, resource usage

Supportability—adaptability, maintainability, internationalization, configurability.

The "+" in FURPS+ indicates ancillary and sub-factors, such as:

Implementation—resource limitations, languages and tools, hardware, ...

Operations—system management in its operational setting.

Packaging Legal—licensing and so forth.

Fact Finding and Requirements Elicitations

In order to identify all relevant requirements analysts must use a range of techniques:-

- Fact-Finding Overview
  - First, you must identify the information you need
  - Develop a fact-finding plan

Fact finding can be done thru Document review, observation, questionnaire, surveys, sampling, research.

Interviewing

The analyst starts by asking **context-free *questions*;**

- a set of questions that will lead to a basic understanding of the problem, the people who want a solution, the nature of the solution desired, and the effectiveness of the first encounter itself.

Structured Questioning Techniques usually lead by software engineer :

–why are we building this system

–who are the other users

–determine critical functionality

Open-ended questions

–useful when not much is known yet

–examples "describe X " tell me what you do

Closed-ended questions

–when enough about the system is known try to ask specific questions

–example "how often should sales reports be generated?”

Try to proceed from open-ended to closed-ended questions!!

-rephrase answers to confirm the customer’s needs

–make sure you understood the client's answer, check for errors, inconsistencies and ambiguities

Find out who else to interview

–who else uses the system –who will agree / disagree with you

–who interacts with you

The Q&A session should be used for the first encounter only and then be replaced by a meeting format that combines elements of problem solving, negotiation, and specification.

Customers and software engineers often have an unconscious **"us and them"** mindset.

Rather than working as a team to identify and refine requirements, each constituency defines its **own "territory"** and communicates through a series of memos, formal position papers, documents, and question and answer sessions.

Facilitated Application Specification Technique (FAST)

Facilitated Application Specification Techniques (FAST), this approach encourages the creation of a joint team of customers and developers who work together to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of requirements.

The idea is to overcome we/them attitude between developers and users team-oriented approach

Guidelines

–participants must attend entire meeting

–all participants are equal

–preparation is as important as meeting

–all pre-meeting documents are to be viewed as "proposed"

–off-site meeting location is preferred

–set an agenda and maintain it

–don't get mired in technical detail

Approaches to FAST

–Joint Application Design (IBM)

–The Method (Performance Resources)

JAD stands for Joint Application Development.

Sit down with the client and design a paper UI that they can see what the application will look like and behave like.

Give the user a chance to work through common scenarios and see if the application will work for them.

Keep refining until the user feels the application is doing what they want it to do.

As you get functionality implemented, bring the user in and have them work through those scenarios and see if it still works.

If they want a change, have a solid estimate of how long the change will add to the schedule and how much it will cost.

The UML and Development Process

UML only standardized artifacts and notations, but it **does not define a standard development process**

- To increase the likelihood of wide spread acceptance of a **standard modeling notation**.
- There is lot of variation in deciding what constitutes an appropriate process, depending upon the staff skill, research, development ratio, nature of problem, tools and so on.

MACRO STEP LEVELS:

Major steps in S/W development includes

- Plan and Elaborate; planning defining requirements, building proto-types
- Build: Construction of the system.
- Deploy: The implementation of the system into use…..

Macro level steps in development:

Iterative Development:

- It is based on **successive enlargement and refinement** of a system through multiple developments cycles.
- Each cycle tackles a relatively small set of requirements, proceedings through analysis, design, construction and testing.
- The **system grows incrementally** as each cycle is completed.

The identified advantages:

- The complexity is reduced
- Early feedback as implementation occurs rapidly for a small subset of the system.

Iterative development cycle:

- Time- boxing & Development cycle
- The development is bound **within a time box**, a rigidly fixed time, such as four weeks.
- To succeed, it is necessary to choose the requirements carefully.

Use Cases and Iterative Development cycle

- A use case describes **narrative description** of a domain process
- A development cycle is assigned to implement one or more use cases.
- Use cases should be ranked, with high ranking use cases to be tacked first.
- The strategy is to first pick the use case that significantly influences the core architecture.

The Plan and Elaborate phase.

Build Phase – Development cycle.

- A repeated development cycle within which the system is extended
- The final objective is to have a working software system that correctly meets the requirements.

Order of Development cycle artifact creation

Some artifacts may be **developed in parallel** such as

- Conceptual model and glossary
- Interaction diagram and class diagrams

Choosing when to create artifacts:

Draft conceptual model

Expand use cases

These are created during early plain and elaborate phase.

The conceptual model should be created where the emphasis is on finding obvious concepts expressed in requirements while deferring a deep investigation.

The conceptual model should be incrementally refined and extended later within the development cycle.

Defining Models and Artifacts

Models in OOAD are dependent relationship between artifacts (objects)

Modeling Systems

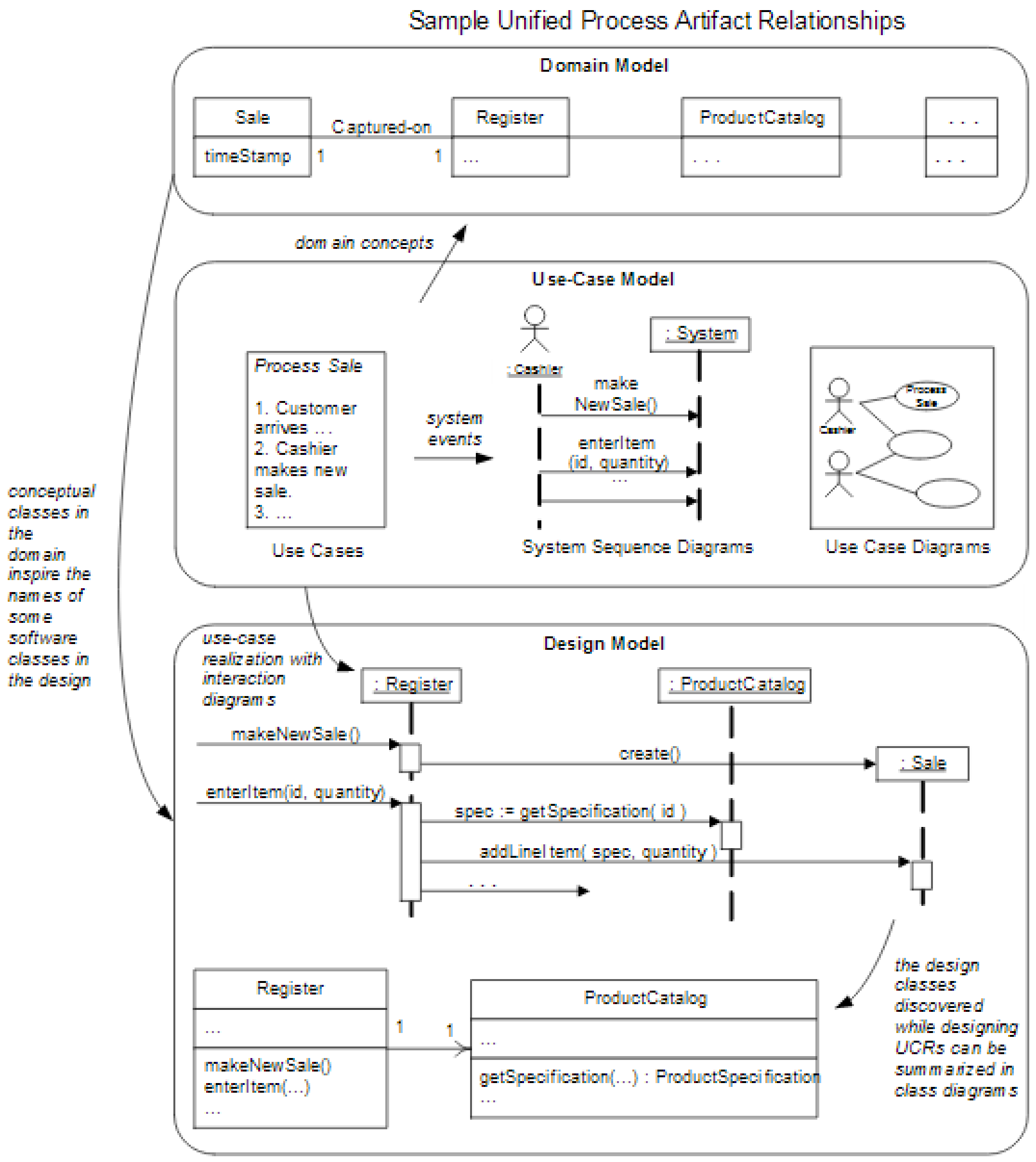
- Systems are inherently complex
- Divide and conquer more appropriate (decompose into understandable chunks)
- Chunks are represented as models

All software systems should be developed by creating models. (which organize and communicate important details of the real world problems it is related to and of the system)

A. Building Conceptual/Domain Models

It is a major activity in the development cycle.

Conceptual/Domain Models



A conceptual/domain model is a representation of **concepts in a problem domain.**

In UML it is a **static structure** diagrams in which **no operations are defined**.

A domain model illustrates meaningful (to the modelers) **conceptual classes in a problem domain**; it is the most important artifact to create during object-oriented analysis.

A domain model is a representation of real-world conceptual classes, not of software components.

It is *not* a set of diagrams describing software classes or software objects with responsibilities.

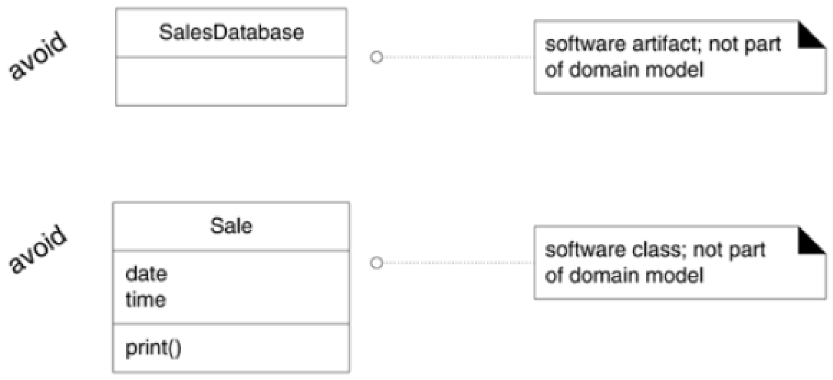
Using UML notation, a domain model is illustrated with a set of **class diagrams** in which **no operations** are defined.

It may show:

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

It must not show:

- software artifacts such as window or database
- responsibilities or methods


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

OOA deals with the **Decomposition of a domain** into noteworthy concepts or objects

Domain Modeling — visual representation of domain concepts in simple UML

- Other names: **conceptual models**, domain object model, analysis object model
- Bounded by a specific domain as described in use cases
- Concepts are not software classes (these are in domain layer)

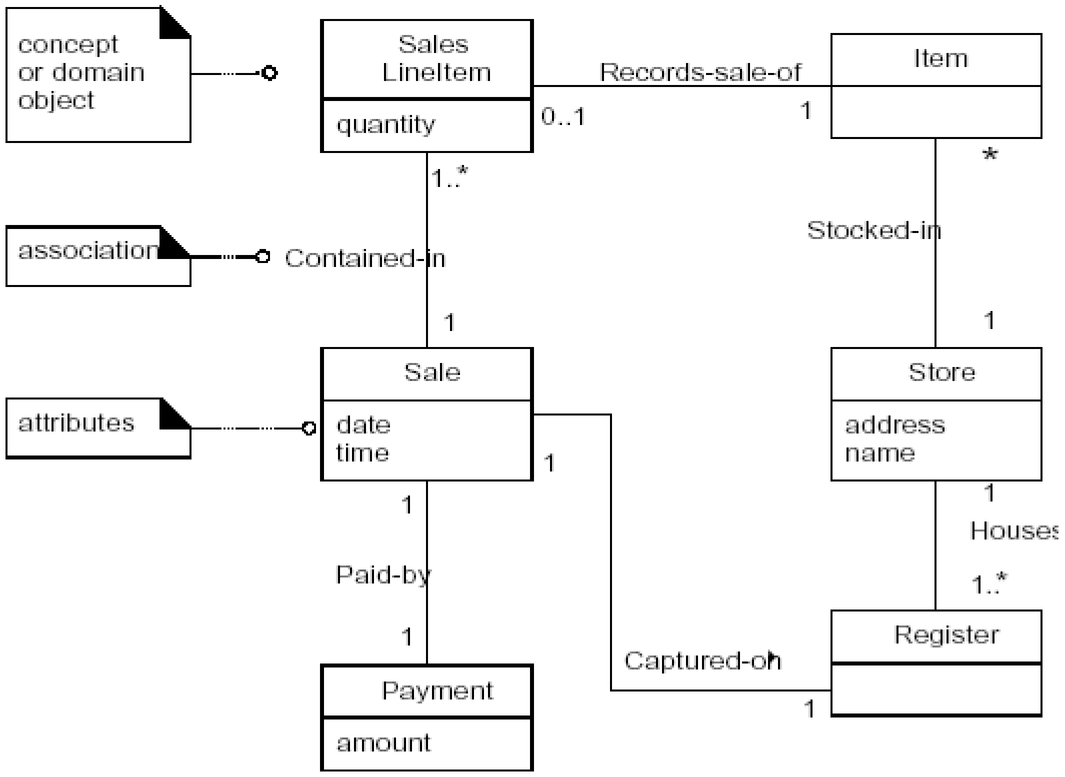
In object Oriented design the emphasis is on **defining logical software objects** that will ultimately be implemented in an object oriented programming language.

Three steps for Creating a Domain Model

i. Find the **domain concepts**

ii. Draw them in a UML class diagram

iii. Add **associations** and **attributes**



• Domain concepts — noteworthy abstractions, domain vocabulary idea, thing or object

e.g Payment, Sale

• Associations — relationships between concepts e.g. Payment Pays-For Sales

• Attributes – information content e.g. Sale records date and time

It can be viewed as a model that communicates what the important terms are, and how they are related.

i. Domain Model : Identifying Conceptual Classes

The domain model illustrates conceptual classes or vocabulary in the domain.

Informally, a conceptual class is **an idea, thing, or object.** 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.

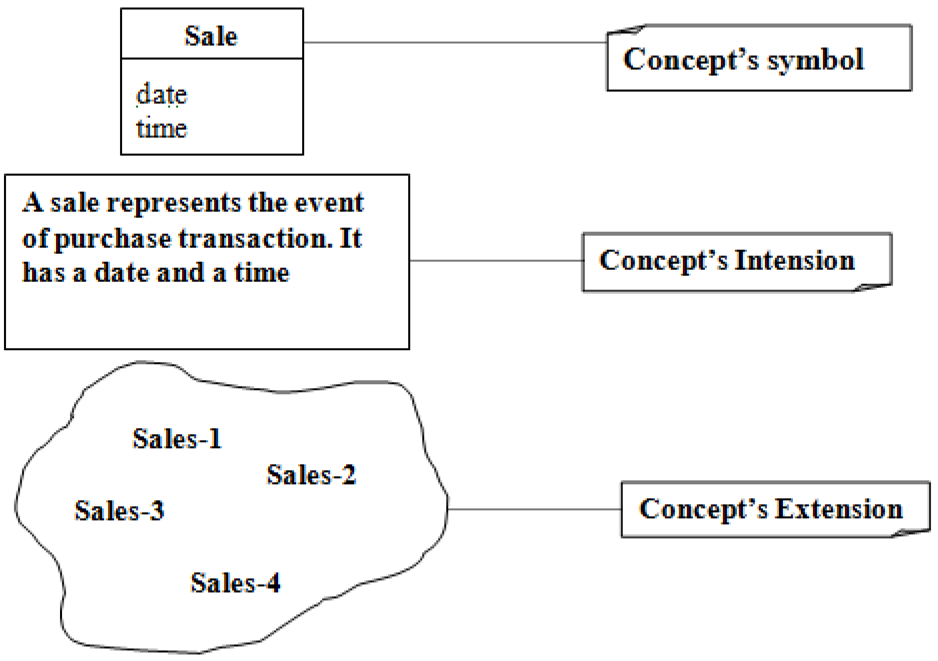
• Extension—the set of examples to which the conceptual class applies.

Conceptual class for the event of a purchase transaction:

Name : Sale.

Intension: represents the event of a purchase transaction, and has a date and time.

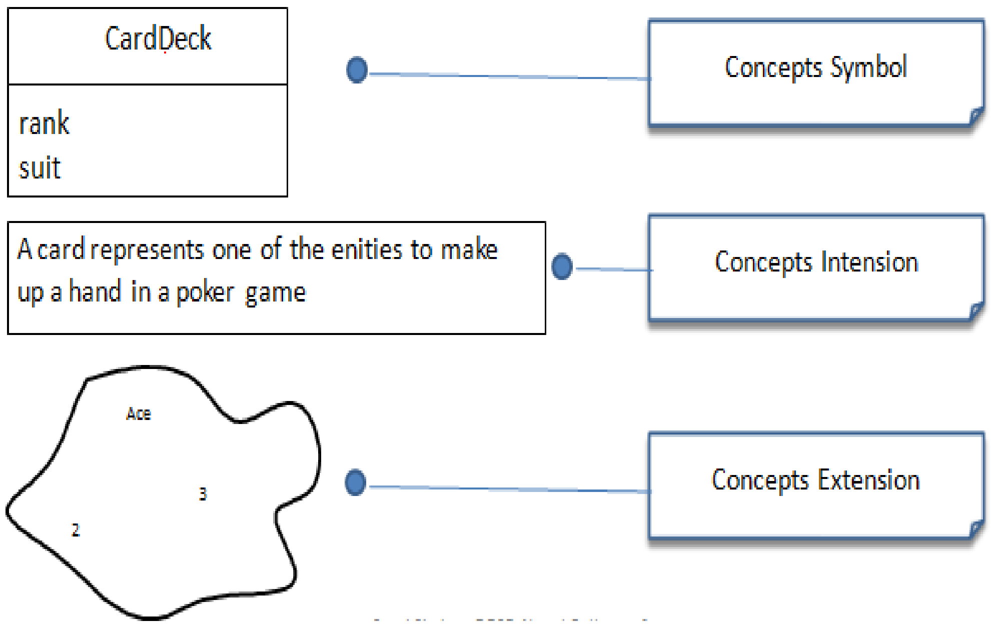
Extension: All the examples of sales; in other words, the set of all sales.



Name : Card

Intension: represents one card in a poker deck that has a rank (2..14) and a suit

Extension: the set of all cards



Domain Models and Decomposition: Software problems can be complex; decomposition—divide-and-conquer—is a common strategy to deal with this complexity by division of the problem space into comprehensible units.

In **structured analysis**, the dimension of decomposition is by processes or **functions.**

In **object-oriented analysis**, the dimension of decomposition is fundamentally by **things or entities in the domain.**

Strategy to Identify Concepts

Two techniques are presented in the following sections:

i. Use a conceptual class category list.

ii. Identify noun phrases.

i. Finding Concepts with the Concept Category List: Start the creation of a domain model by making a list of candidate conceptual classes.

Conceptual Class Category	**Examples**
Business Transactions Guidelines: Critical concepts (involves money)	Sale, Payment Reservation
Transaction line items Guideline: transactions often come with related line items,	SaleLineItems
Product ore service related to a transaction or transaction line item related Guideline: Transactions are for something (a product or service)	Flight, item, seat, meal
Where is the transaction recorded?	Register,ledger, flightManifest
Roles of people or organizations related to the transaction: actors in the use case Guideline: It is desirable to know about	Cashier, customer, airline, passenger, Pilot

the parties involved in a transaction	
Place of transaction: place of service	Store, airport, plane , seat
Noteworthy events, often with a time or place we need to remember	Sale, payment, flight
Physical or Tangible objects Guidelines: especially relevant when creating device control software or simulations	POST, die, Airplane, board, item
Specification, design or description of things	ProductSpecification, FlightDescription
Catalogues	ProductCatalogue, PartCatalogue
Containers of other things	Store, Bin Airplane
Things in a container	Item, square, passenger
Other collaborating items	CreditAuthorizationSystem, AirTrafficControl
Record of finance, work, contracts, legal matters	Receipt, ledger logs
Places	Store, Airport
Financial instruments	Cash, check, lineofCredit
Schedules, manuals, documents that are regularly referred to in order to perform work	dailyPriceChangeList, repairSchedule

ii. Finding Concepts with the Noun Phrase Identification: Identify the Noun and the Noun Phrase in the textual description in the Problem Domain and consider them as candidate concepts or attributes

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.

Cashier repeats steps 2-3 until indicates done.

5. System presents total with **taxes** calculated.

6. Cashier tells Customer the total, and asks for **payment.**

7. Customer pays and System handles payment.

8. System logs the completed **sale** and sends sale and payment information to the external **Accounting** (for accounting and **commissions)** and **Inventory** systems (to update inventory).

9. System presents **receipt.**

10.Customer leaves with receipt and goods (if any).

Extensions (or Alternative Flows):

7a. Paying by cash:

1. Cashier enters the cash **amount tendered.**

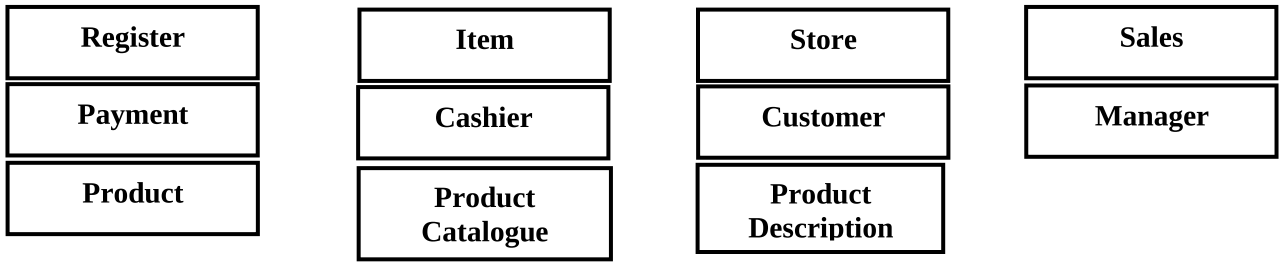
Candidate Conceptual Classes for the Sales Domain

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

The list is constrained to the requirements and simplifications currently under consideration—the simplified scenario *of Process Sale.*

Register	*ProductDescription*
Item	*SalesLineItem*
Store	*Cashier*
Sale	*Customer*
Payment	*Manager*
ProductCatalog	

The Process of Sale Conceptual Model (Concepts Only)



Domain Modeling Guidelines

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

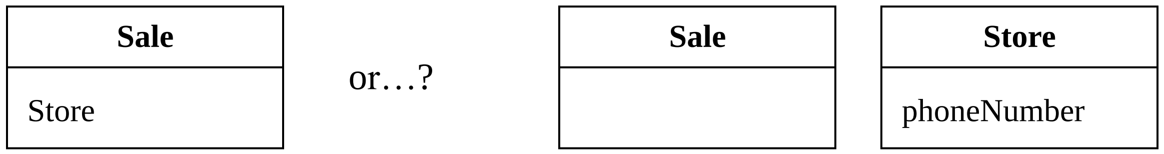
A Common Mistake in Identifying Conceptual Classes

Perhaps the most common mistake when creating a domain model is to **represent something as an attribute when it should have been a concept.**

The common rule :

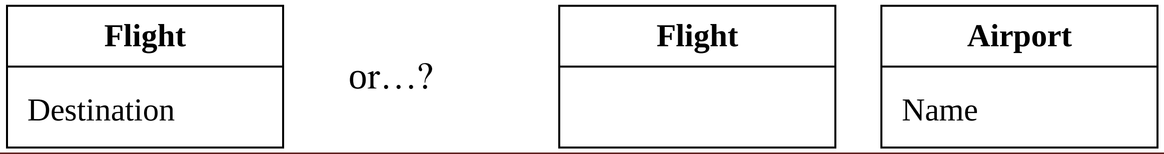
If we do not think of some conceptual class X as a number or text in the real world, X is probably a conceptual class, not an attribute.

As an example, should store be an attribute of Sale, or a separate conceptual class Store?



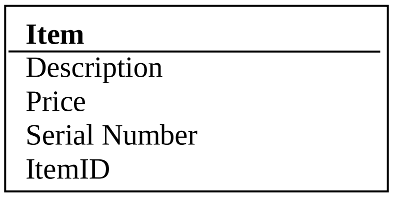
In the real world, a store is not considered a number or text—the term suggests a legal entity, an organization, and something occupies space. Therefore, Store should be a concept.

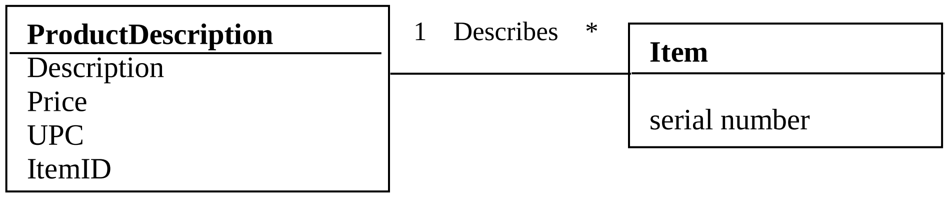
In Airline Domain, Will Destination be a concept of or an attribute Of Flight?



Destination Airport is a massive thing that occupies space so it is a concept not an attribute

Specification or Description Concepts





Description or specification objects are strongly related to the things they describe. In a domain model, it is common to state that an XDescription Describes an X

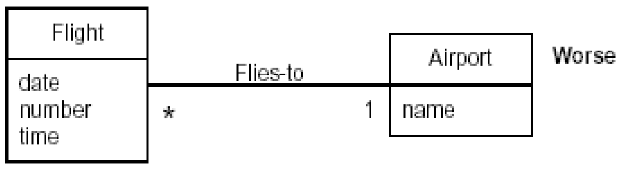
When Are Description Conceptual Classes Required?

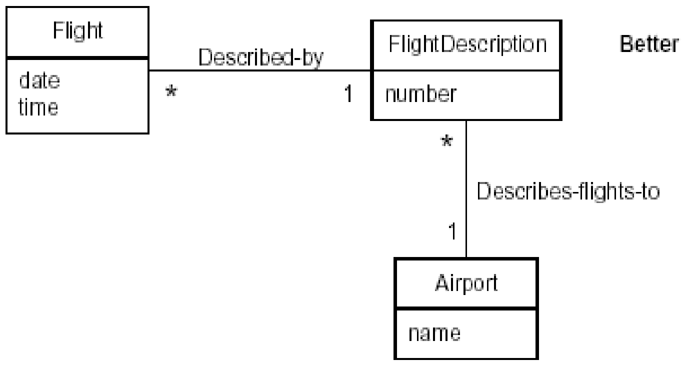
Add a specification or description conceptual class (*ProductDescription)* when:

- 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.

Another example:





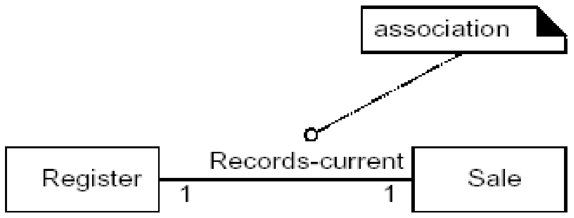
Why create a Domain Model?

Domain models are created **to get a better understanding of the key concepts and vocabulary used in a particular domain**. If it is required to develop a software for an aircraft manufacturing company, one should do proper domain modeling to understands the key concepts and jargons or technology specific words used there in order to be familiar with that domain. It helps in reducing the gap between the software representation and the mental model of the domain as perceived by the analyst.

ii. Domain Modeling : Adding Associations

Associations

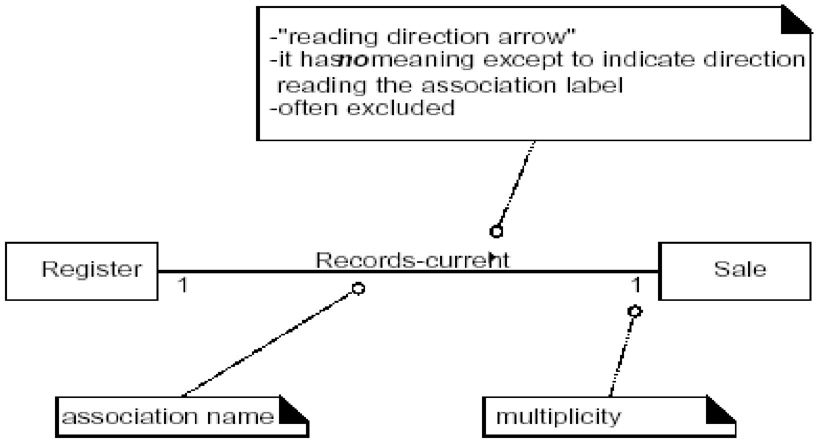
It is **a relationship between concepts** that indicates some meaningful interesting connection.



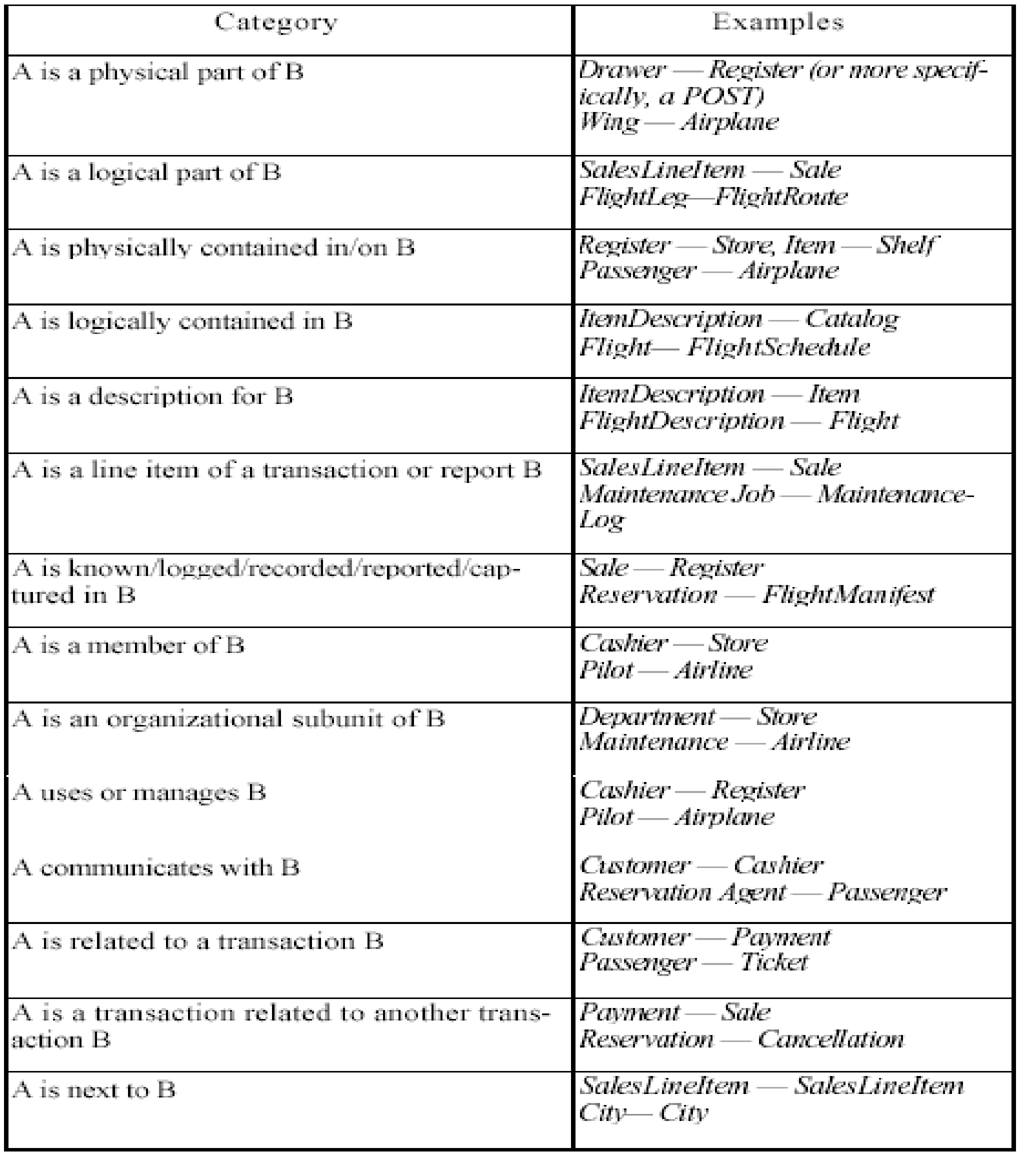
The UML Notation for association

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.


-"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

Finding Associations- Common Association List:



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

High Priority Associations

Some high-priority association categories that are invariably useful to include in a domain model:

A is a physical or logical part of B A is recorded in B

A is a physically or logically contained in B

Association Guidelines

- Focus on those associations in which knowledge of the relationship needs to be preserved "need to know"
- It is more important to identify concepts then 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

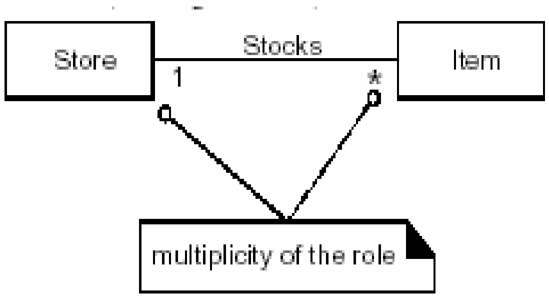
Each end of an association is called a **role.** Roles may optionally have:

- name
- navigability
- multiplicity expression

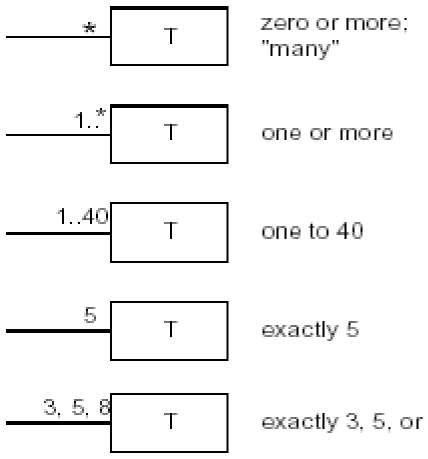
i. Multiplicity

- It defines how many instances of A can be associated with one instance of a type B, at a particular moment in time.

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



Some Examples of Multiplicity



In UML the multiplicity value is context dependent

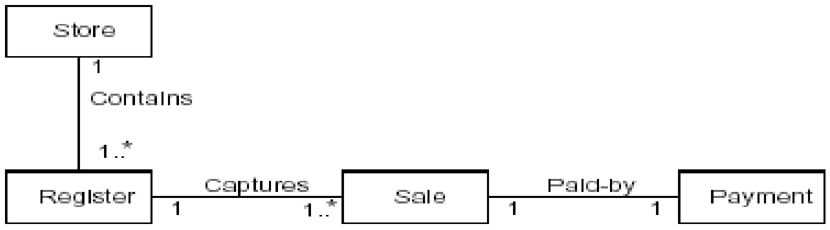
Naming Association

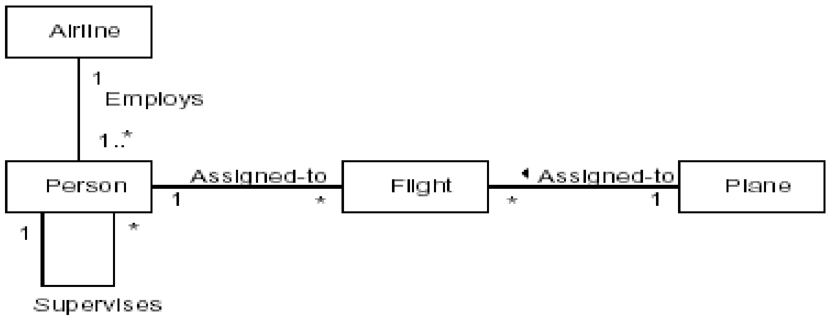
- Name an association based on a **TypeName-VerbPhrase-TypeName** format where the verb phrase creates a sequence that is readable and meaningful in the model context.
- Association names should **start with a capital letter**, since an association represents a classifier of links between instances; in the UML, classifiers should start with a capital letter.

Good: Sale PaidBy CashPayment	Bad: Sale uses CashPayment
Good: Player IsOn Square	Bad: Player has Square

Two common and equally legal formats for a compound association name are:

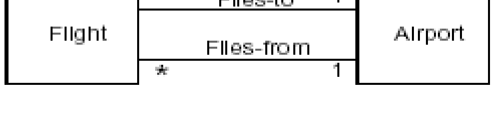
- *Paid-by* - *PaidBy*





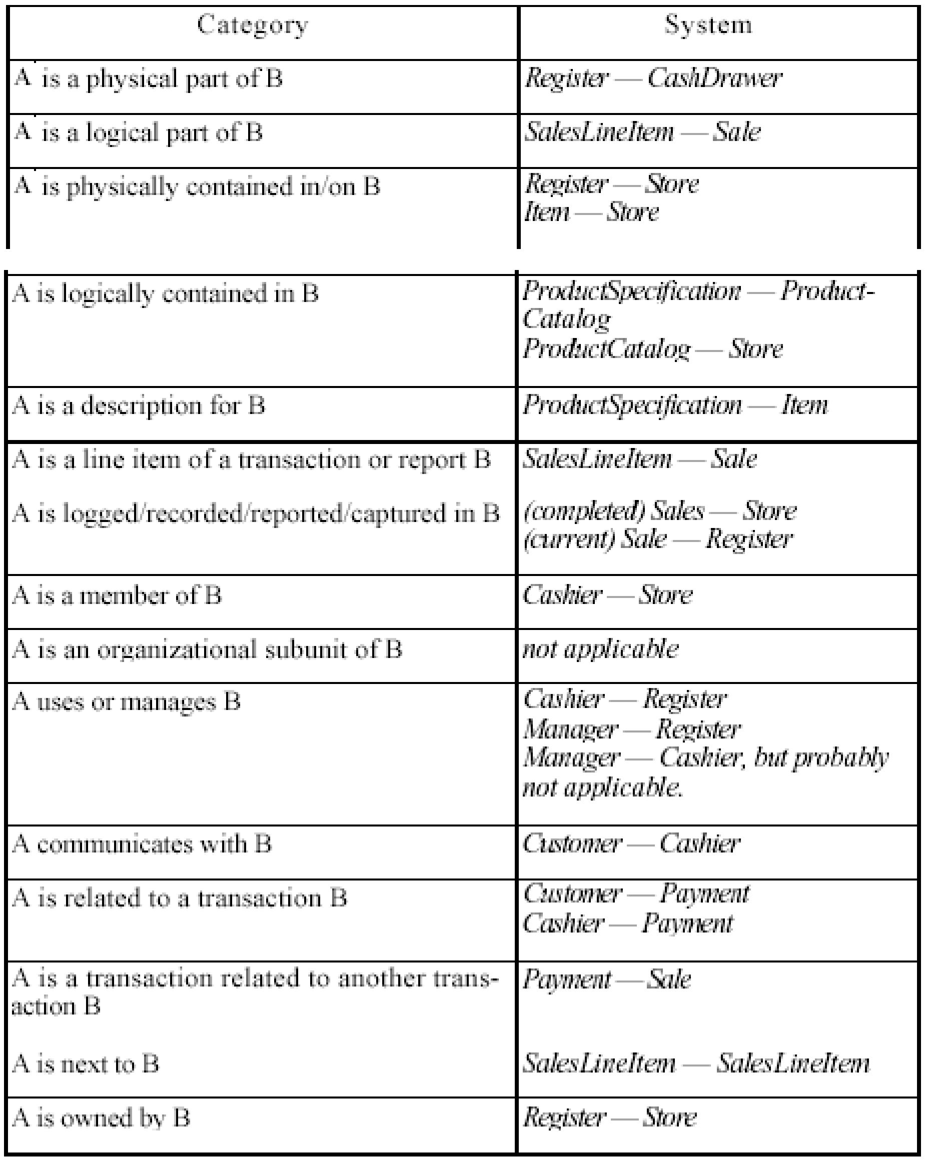
Multiple Associations between Two Types

Two types may have multiple associations between them; this is not uncommon.



Applying the Category of Associations Checklist

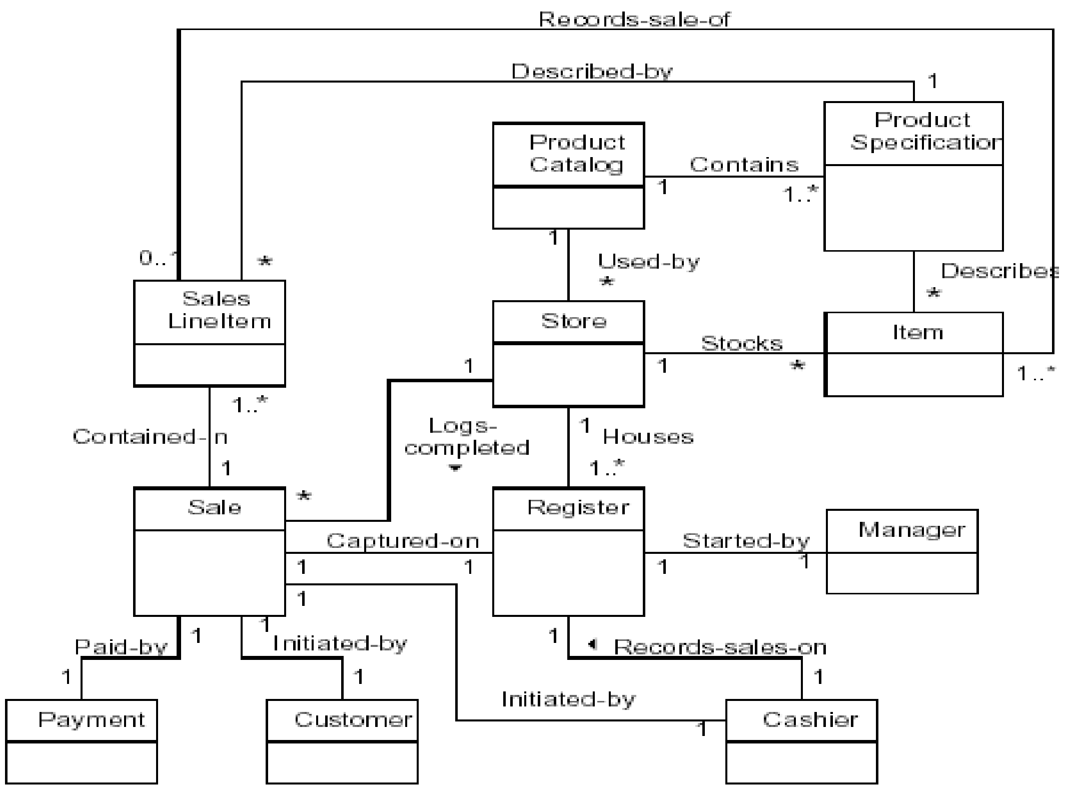
Based on previously identified types and considering the current use case requirements.



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

Point of Sale Domain Model

The figure below shows candidate concepts and associations for the POST system



iii. Domain Model: Identifying Attributes

Attribute: It is a **logical data value** of an Object

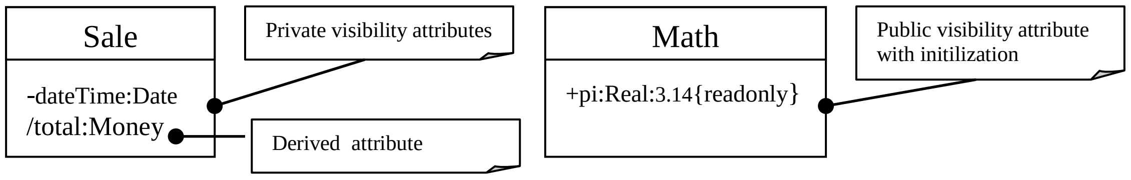
UML Attribute Notation

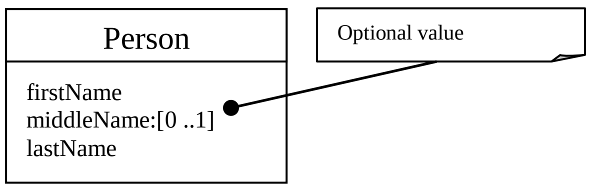
Attributes are shown in the second compartment of the class box.

Their type may optionally be shown

The syntax for an attribute in the UML:

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


Sale
-dateTime:Date
/total:Money
Private visibility attributes
Derived attribute
Math
+pi:Real:3.14{readonly}
Public visibility attribute with initilization



Valid Attribute Types

Keep Attributes Simple

Intuitively, most simple attribute types are what are often thought of as **primitive data types, such as numbers.**

The type of an attribute should not normally be a complex domain concept, such as a *Sale or Airport.*

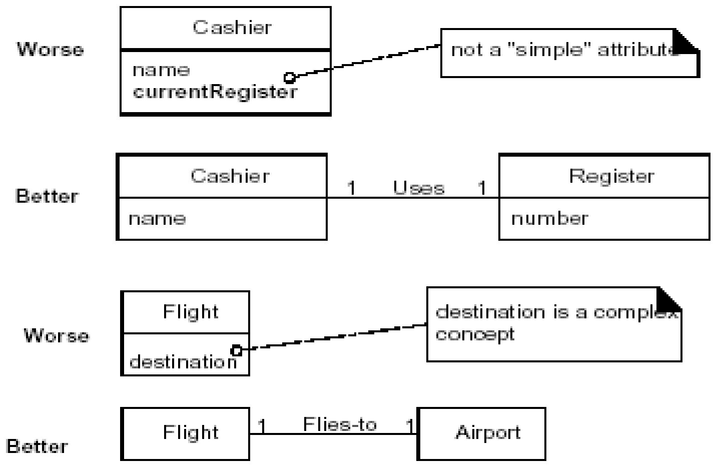
For example, the *currentRegister* attribute in the *Cashier* class is undesirable because its type is meant to be a *Register,* which is not a simple attribute type (such as *Number* or *String).*

The most useful way to express that a *Cashier* uses a *Register* is with an association, not with an attribute.

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, Universal Product Code (UPC), SKU,ZIP or postal codes, enumerated types**



Data Types

Attributes should generally be **data types.** This is a UML term that implies a set of values for which unique identity is not meaningful.

For example, it is not meaningful to distinguish between:

• Separate instances of the *Number* 5.

• Separate instances of the *String* 'cat'.

• Separate instances of *PhoneNumber* that contain the same number.

• Separate instances *of Address* that contain the same address.

Non-Primitive Attribute Types

Represent what may initially be considered a primitive data type (such as a number or string) as a non-primitive class if:

- It is composed of separate sections.
  - phone number, name of person
- There are operations usually 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) or European Article Number (EAN)

Attributes in the PoST Domain Model

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

Payment

ProductSpecification

Sale

SalesLineItem

Store

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

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.

date, *time*—A receipt is a paper report of a sale. It

normally shows date and time of sale.

quantity—To record the quantity entered, when there is more than one item in a line item sale (for example, *five* packages of tofu).

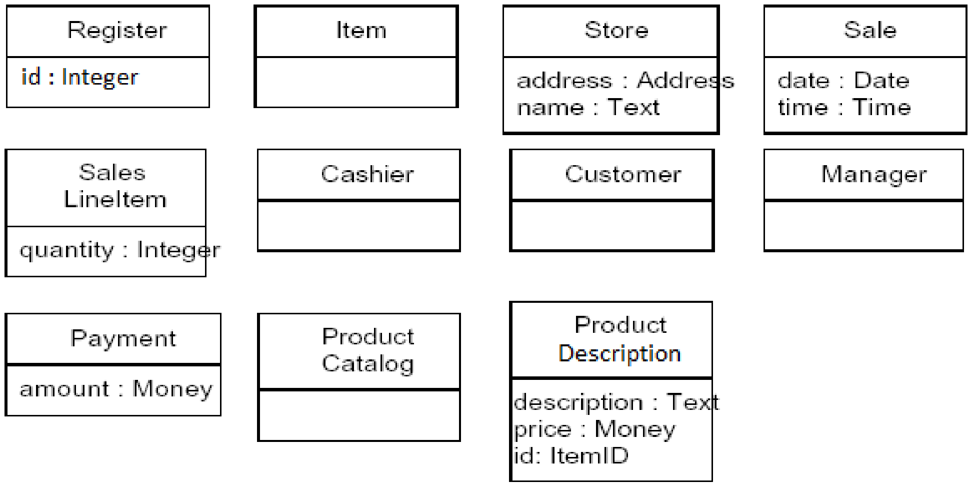
address, *name*—The receipt requires the name and address of the store.

The list of attributes should primarily be constrained to the requirements and simplifications currently under consideration.

Some attributes are clearly called for by reading Requirements specification

Current use cases under considerations Simplifications, clarifications, and assumption documents

e.g. It is necessary to record the date and time of a sale, therefore the sale concept requires a date and a time attribute.



Discussion for POST Attributes

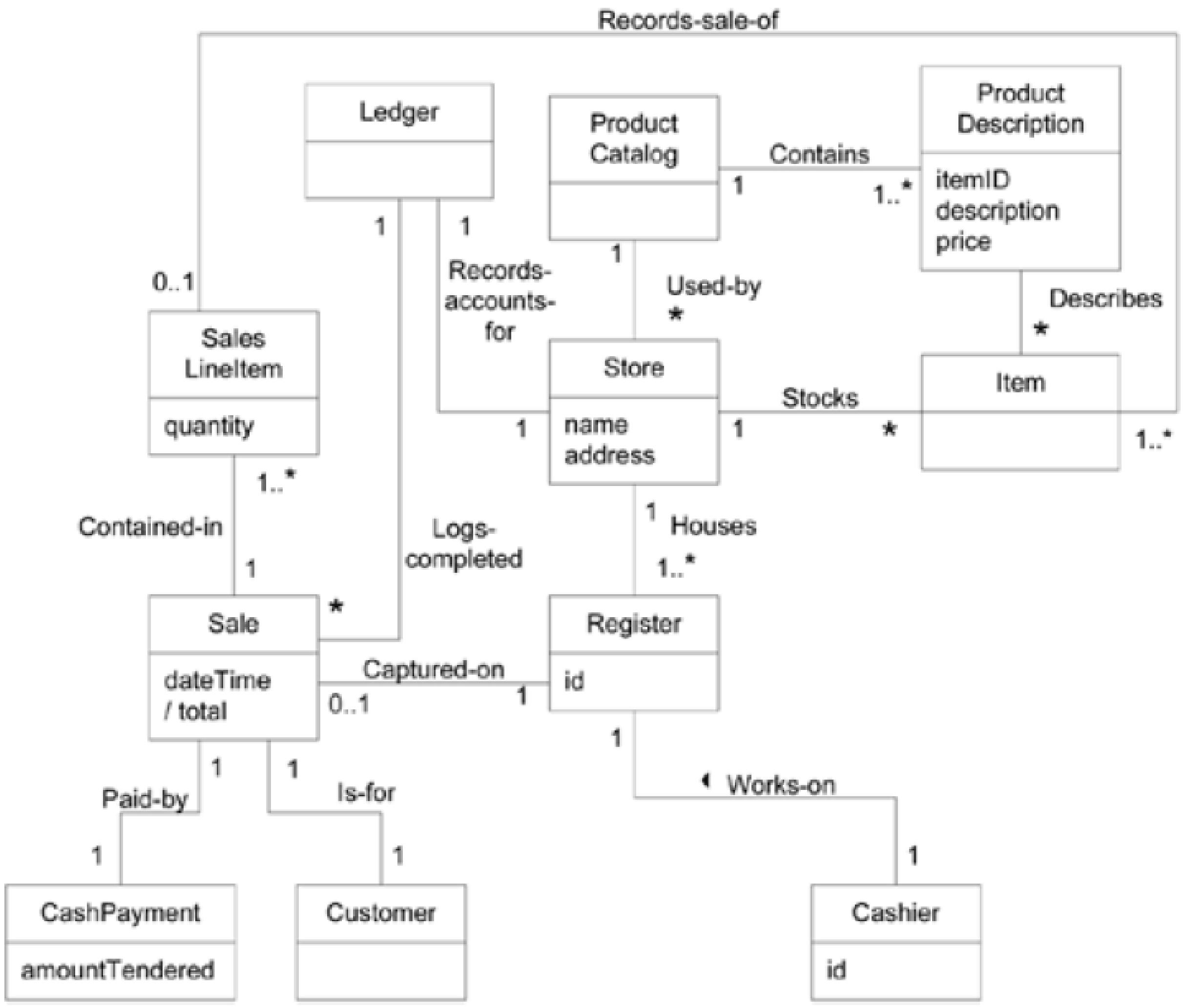
Payment: amount- In order to determine that sufficient payment was provided, and to calculate change, an amount must be entered.

ProductDescription: description- In order to show the description on a display or a receipt

UPC: upc- Inorder to look up ItemSpecification, given an entered UPC, it is necessary to relate them to a UPC

Price- In order to calculate the sales total

A partial domain model.

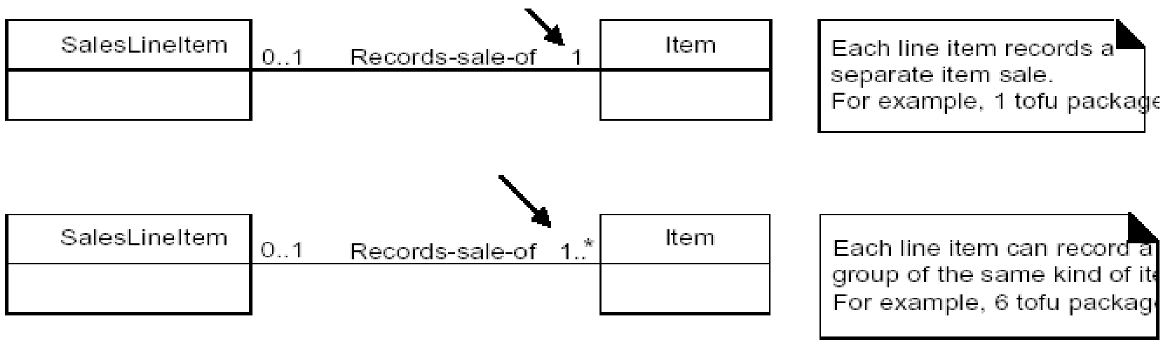


Multiplicity from SalesLineItem to Item

It is possible for a cashier to receive a group of like items (for example, six tofu packages), enter the *itemID* once, and then enter a quantity (for example, six).

Consequently, individual *SalesLineItem* can be associated with more than one instance of item.

The quantity that is entered by the cashier may be recorded as an attribute of the *SalesLineItem* .



SalesLineItem
0..1
Records-sale-of
1
Item
Each line item records a separate item sale.
For example, 1 tofu package
SalesLineItem
0..1
Records-sale-of
1..*
Item
Each line item can record a group of the same kind of it
For example, 6 tofu packag

However, the quantity can be calculated from the actual multiplicity value of the relationship, so it may be characterized as a **derived attribute**—one that may be derived from other information. In the UML, a derived attribute is indicated with a "/" symbol.

It is used when we want to communicate that

- This is a noteworthy attribute but
- It is derivable

e.g. a cashier can receive a group of like items (a package of pencils) and enter itemID once and then enter a quantity. A sale line item can be associated with more than one instance of an item

B. Building System Sequence Diagrams - System Behavior

A system sequence diagram is a fast and easily created artifact that **illustrates input and output events** related to the systems under construction.

The UML contains notation in the form of sequence diagrams to **illustrate events from external actors to a system.**

System Behavior

It is a description of **what system does**, without explaining how system does it.

Before proceeding to a logical design of how a software application will work, it is useful to investigate and define its behavior as a **"black box."**

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.

Other parts include the use cases, and system contracts

System Sequence Diagrams

Use cases describe **how external actors interact** with the software system

During this interaction an actor generates events to a system, usually requesting some operation in response. For example, when a cashier enters an item's ID, the cashier is requesting the POS system to record that item's sale.

That request **event initiates an operation** upon the system.

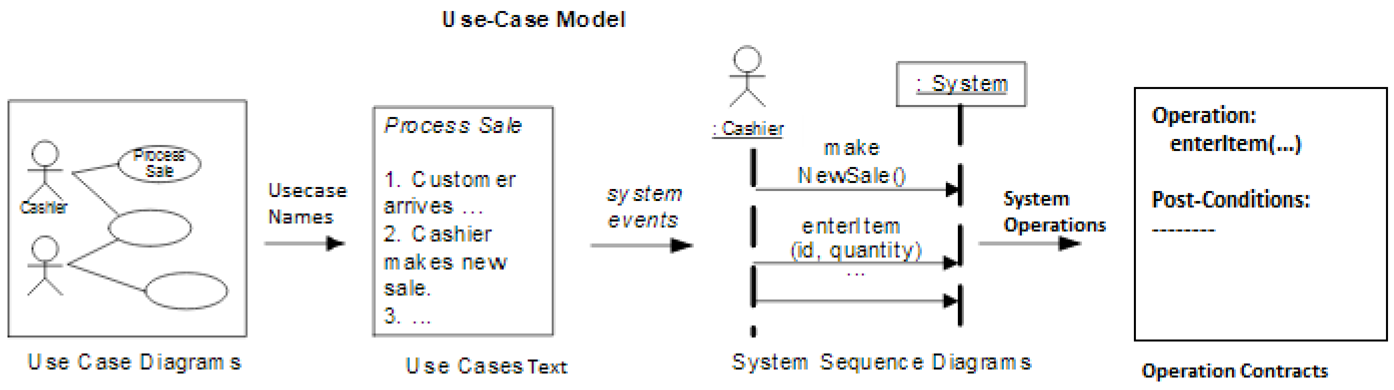
It is desirable to **isolate and illustrate the operations that an external actor requests** of a system, because they are an important part of understanding system behavior.

The UML includes sequence diagrams as a notation that can illustrate actor interactions and the operations initiated by them.

A system sequence diagram (SSD) is a picture that shows, for a 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**; the emphasis of the diagram is events that cross the system boundary **from actors to systems.**

An SSD should be done for the **main success scenario** of the use case, and frequent or complex alternative scenarios.



Example of a sequence diagram

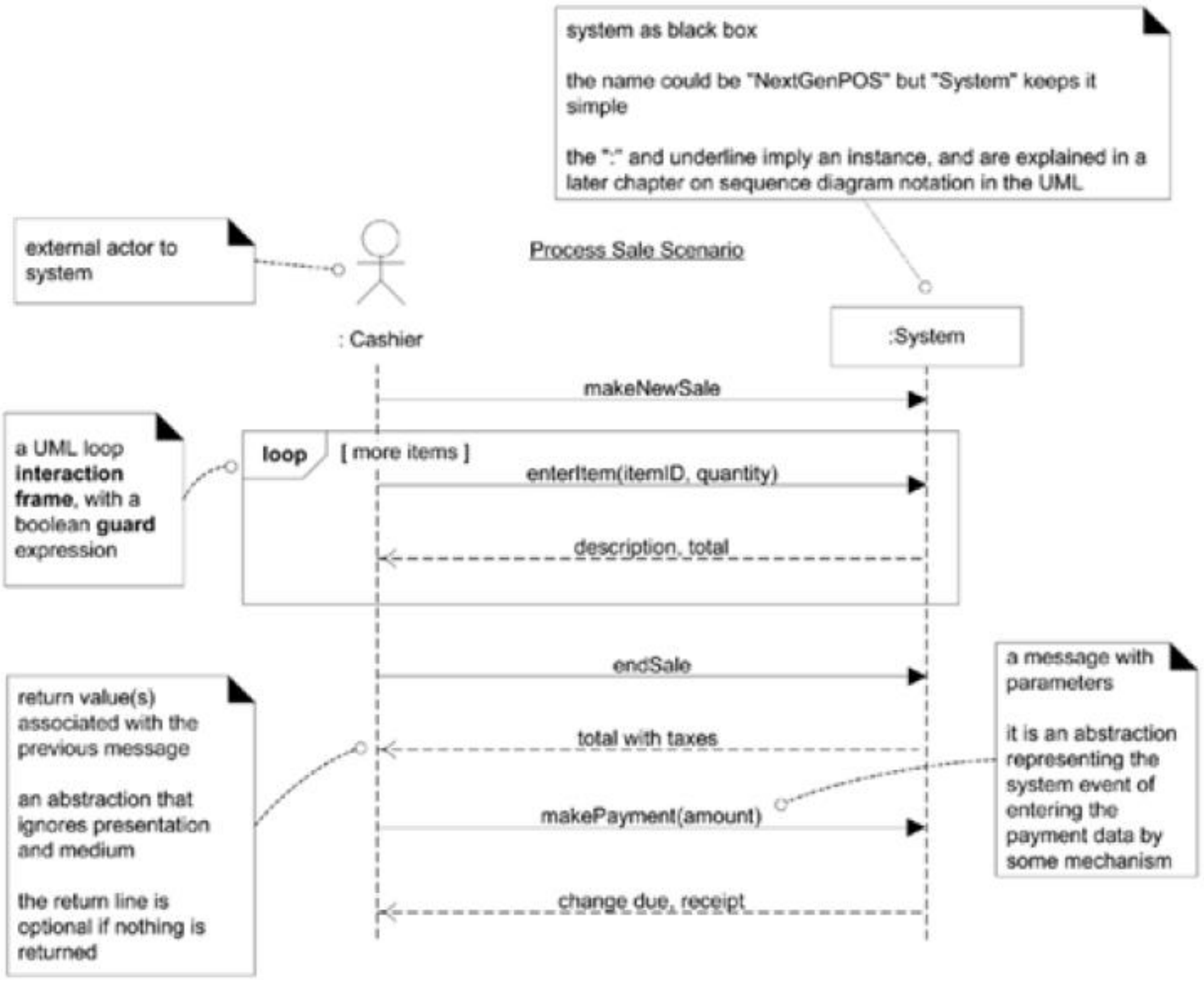
An SSD shows, for a particular course of events within a use case, the external actors that interact directly with the system, the system (as a black box), and the system events that the actors generate .

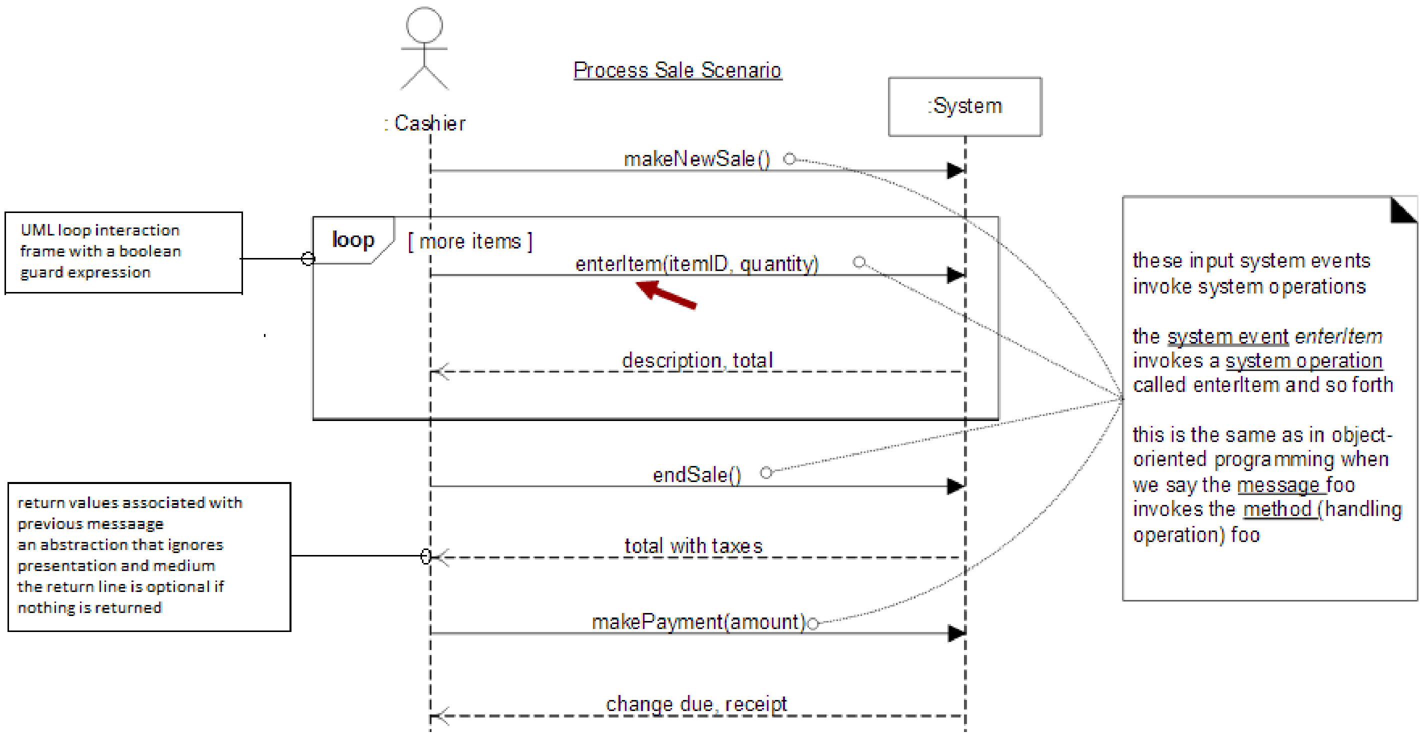
Time proceeds downward, and the ordering of events should follow their order in the use case.

System events may include parameters.

This example is for the main success scenario of the Process Sale use case.

It indicates that the cashier generates **makeNewSale, enteritem, endSale, and makePayment system events.**


system as black box
the name could be "NextGenPOS" but "System" keeps it simple
the ":" and underline imply an instance, and are explained in a later chapter on sequence diagram notation in the UML
external actor to system
Process Sale Scenario
: Cashier
:System
makeNewSale
a UML loop interaction frame, with a boolean guard expression
loop
[more items]
enterItem(itemID, quantity)
description, total
endSale
total with taxes
makePayment(amount)
change due, receipt
a message with parameters
it is an abstraction representing the system event of entering the payment data by some mechanism
return value(s) associated with the previous message
an abstraction that ignores presentation and medium
the return line is optional if nothing is returned


Process Sale Scenario
: Cashier
:System
makeNewSale()
UML loop interaction frame with a boolean guard expression
loop
[more items]
enterItem(itemID, quantity)
description, total
endSale()
total with taxes
makePayment(amount)
change due, receipt
these input system events invoke system operations
the system event enterItem invokes a system operation called enterItem and so forth
this is the same as in object-oriented programming when we say the message foo invokes the method (handling operation) foo
return values associated with previous messaage
an abstraction that ignores presentation and medium
the return line is optional if nothing is returned

Significance of Drawing System Sequence Diagrams

It is required to design software to handle events from mouse, keyboard etc coming in to the system and execute a response

Software system reacts to three things

- External events from actors (human or computers)
- Timer events
- Faults or exceptions

So it becomes necessary to know the external/system events to analyze the system behavior

System sequence diagrams are drawn to investigate and define the behavior of a software application as a black box before going into detailed design of how it works

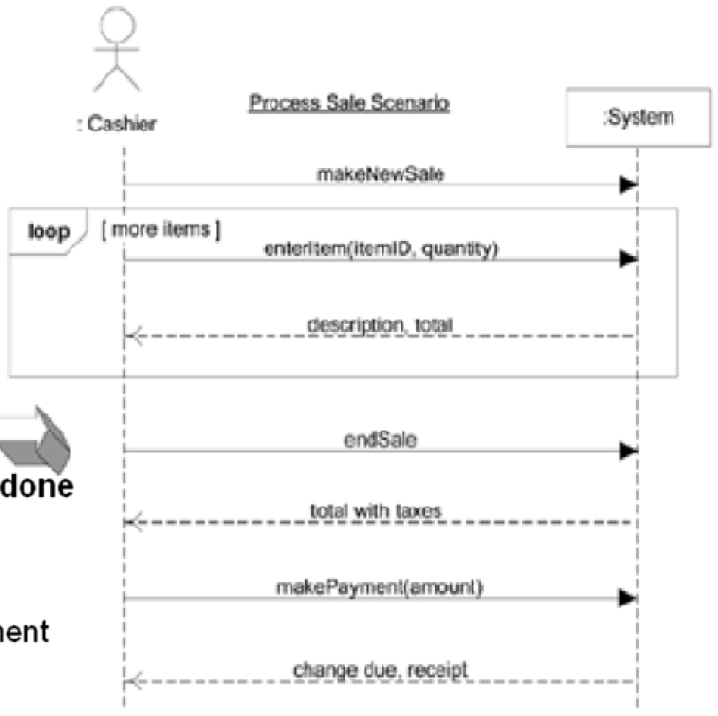
System behavior is a description of what a system does, without explaining how it does it and System sequence diagram is a part of that description

SSDs and Use Cases

An SSD shows system events for a scenario of a use case, therefore it is generated from inspection of a use case. So there can be multiple scenarios in case of a system and those scenarios can be further elaborated by using individual system sequence diagrams to depict the system events.

Simple Cash only Process Sale Scenario

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**
   Cashier repeats steps 3-4 times until indicates done
5. System presents total with taxes calculated
6. Cashier tells customer the total and asks for payment
7. Customer pays and system handles payment

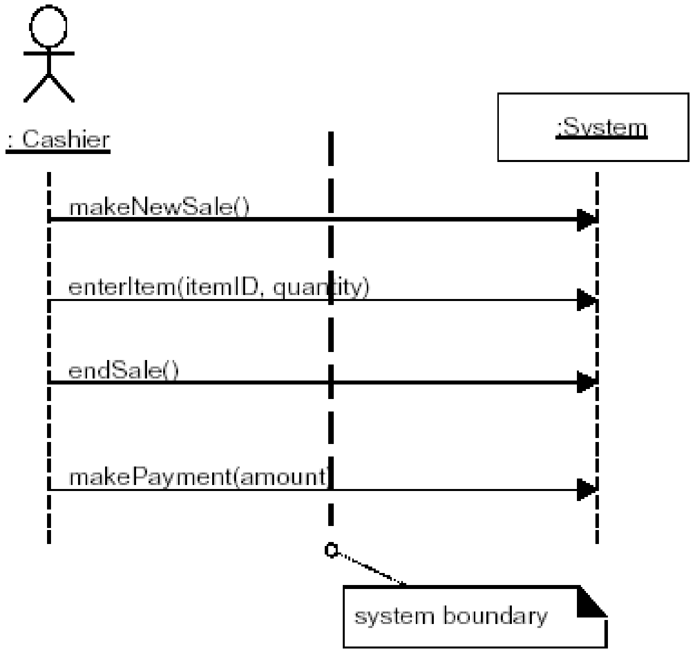


Like in case of an ATM system the use case scenarios `make_withdrawal`, `make_balance_enquiry`, `maintain_ATM` etc can be elaborated further by individual system sequence diagrams to identifying the system events pertaining to the individual scenario.

System Events and the System Boundary

To identify system events, it is necessary to be clear on the choice of system boundary.

For the purposes of software development, the **system boundary is usually chosen to be the software (and possibly hardware) system itself**; in this context, a system event is an external event that directly stimulates the software

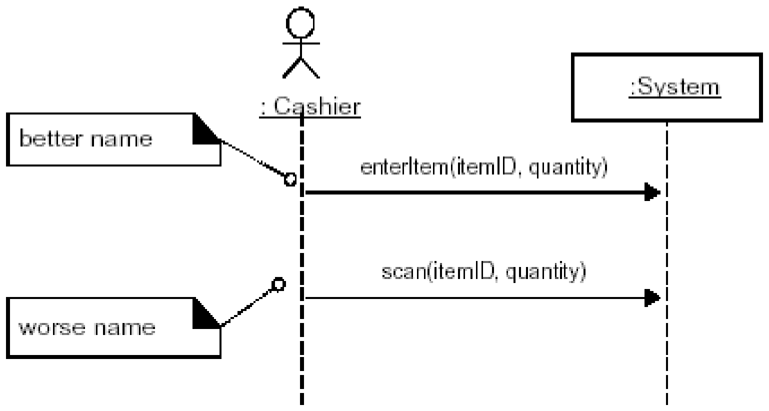


Consider the *Process Sale* use case to identify system events. First, we must determine the actors that directly interact with the software system. The customer interacts with the cashier, but for this simple cash-only scenario, does not directly interact with the POS system—only the cashier does. Therefore, **the customer is not a generator of system events**; only the cashier is.

Giving Names to System Events and System Operations

System events (and their associated system operations) should be **expressed at the level of intent** rather than in terms of the physical input medium or interface widget level.

It also improves clarity to **start the name of a system event with a verb** (add...,enter..., end..., make...)



A **system event is an external input event generated by an actor to a system**.

An Event initiates a responding operation.

A **system operation is an operation of the system that executes in response to a system event**s

When the Cashier generates the enterItem system event it causes the execution of the enterItem system operation (in this case the event is the named stimulus and the operation is the response)

So it is **better to name a system event as enterItem** rather than scan as scan is specific form of entering items into system

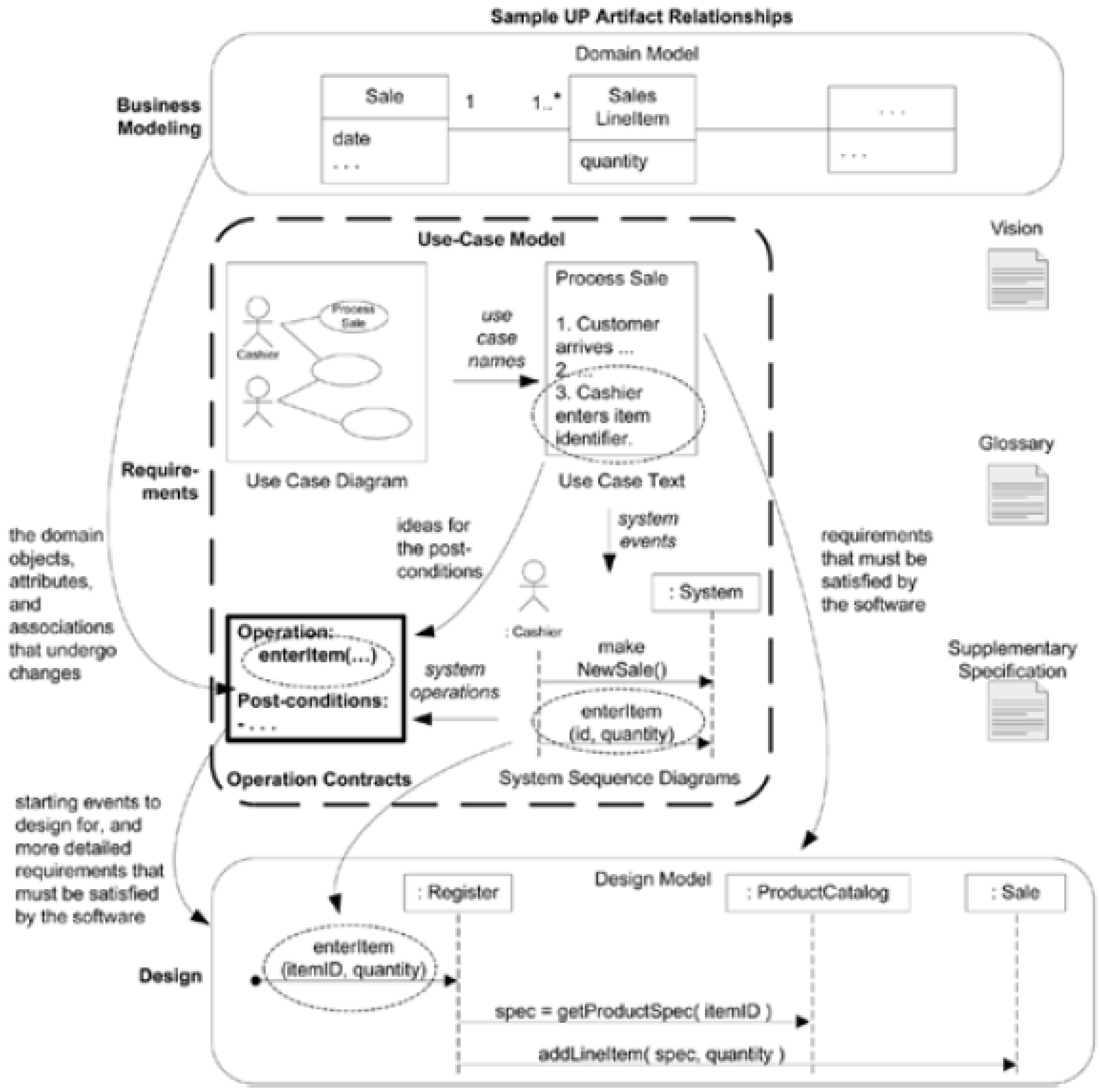
Recording System Operations

The set of all required systems operations is determined by identifying the system events, using parameters e.g. `enterItem(UPC, quantity), endSale(), makePayment(amount)`

How to make a Sequence Diagram

- **Draw a line** representing the system as a Black Box
- **Identify each actor** that directly operates on the system. Draw a line for each such actor
- From the Use Case typical course of events text, **identify the system (external) events** that each actor generates. Illustrate them on the diagram.
- Optionally, include the use case text to the left of the diagram.

System Behavior – Contracts



Contracts for operations can **help define system behavior**; they describe the **outcome of executing system operation** in terms of state changes to domain objects.

The UML contains support for defining contracts by allowing the definition of **pre and post-conditions** of operations.

Their creation is dependent on

- Development of Conceptual Model
- System Sequence Diagrams
- The Identification of System Operations

Contracts:

Use cases are the primary mechanism in the UP to describe system behavior, and are usually sufficient. However, sometimes a more detailed description of system behavior has value.

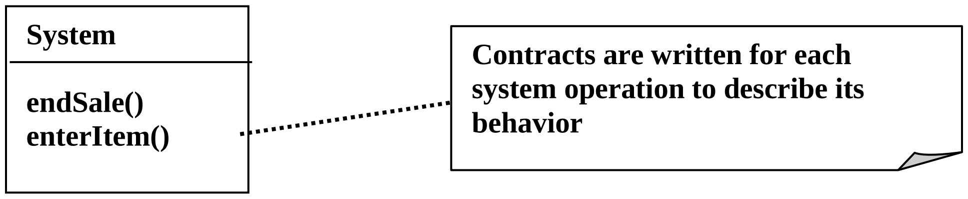
Contracts **describe detailed system behavior in terms of state changes** to objects in the Domain Model, after a system operation has executed.

It is a document that **describes what an operation commits** to achieve.

It is usually declarative in style, **emphasizing what will happen** rather than how it will be achieved.

It is common for Contracts to be **expressed in Terms of Pre and the Post Conditions** state changes

A contract can be written for an individual method of a software class or for a system operation


System
endSale()
enterItem()
Contracts are written for each system operation to describe its behavior

SSD, System Events, System Operations and Operation Contracts

Contracts are defined for system operations, the operations that the system offers in its public interface to handle incoming system events.

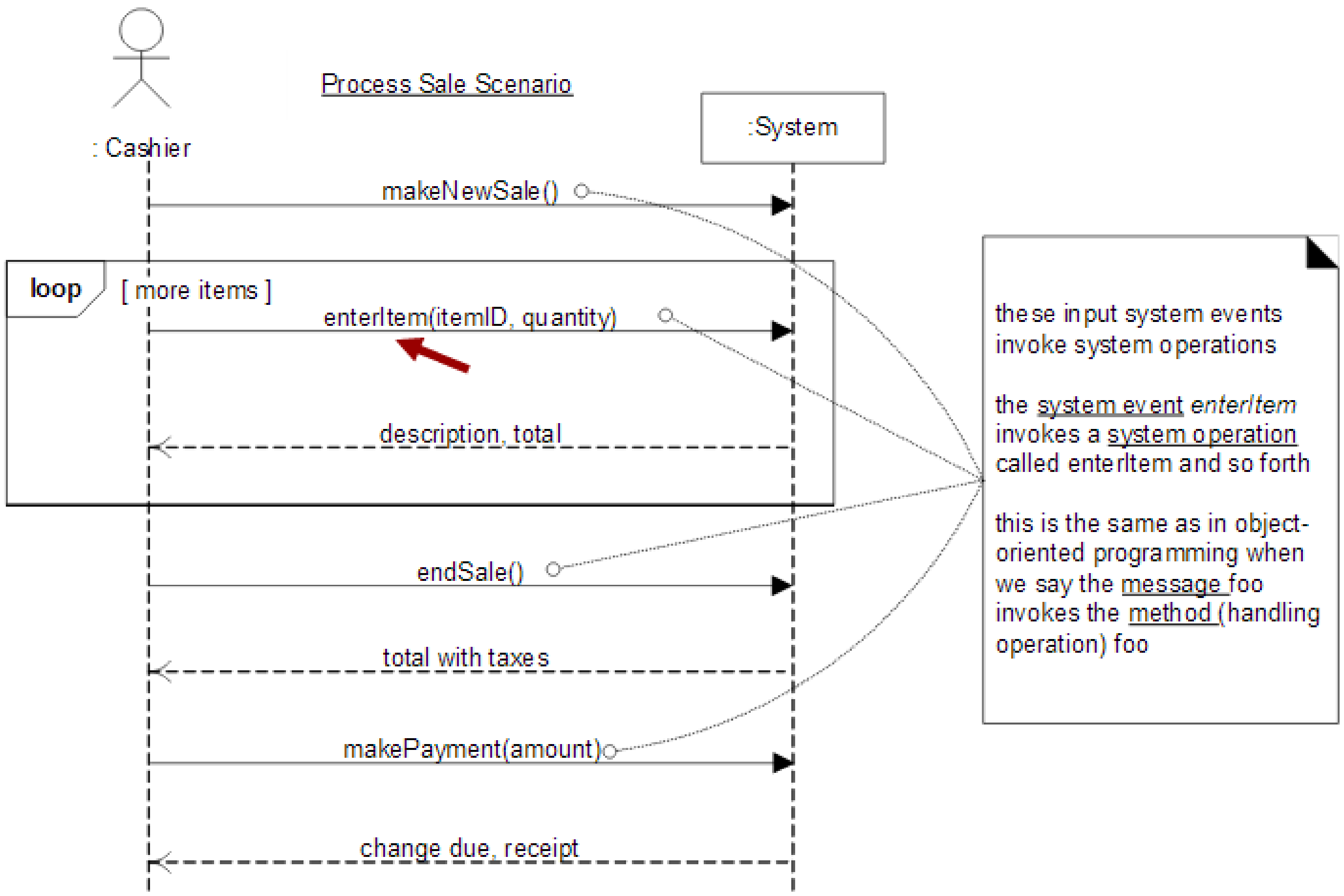
System operations can be identified by discovering these system events

SSD shows system events or I/O messages relevant to the system

Input system events imply that the system has system operations to handle the events just like an OO message (a kind of even or signal) is handled by an OO method (a kind of operation)

The entire set of system operations defines the public system interface, viewing the system as a single class or component.

In the UML the system as a whole can be represented as one object of a class named System



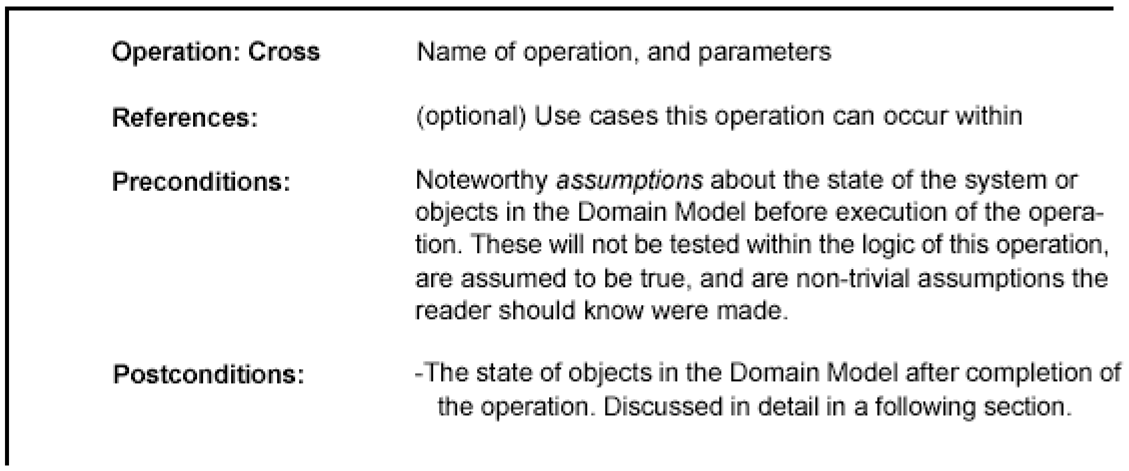
these input system events invoke system operations

the system event *enterItem* invokes a system operation called enterItem and so forth

this is the same as in object-oriented programming when we say the message foo invokes the method (handling operation) foo

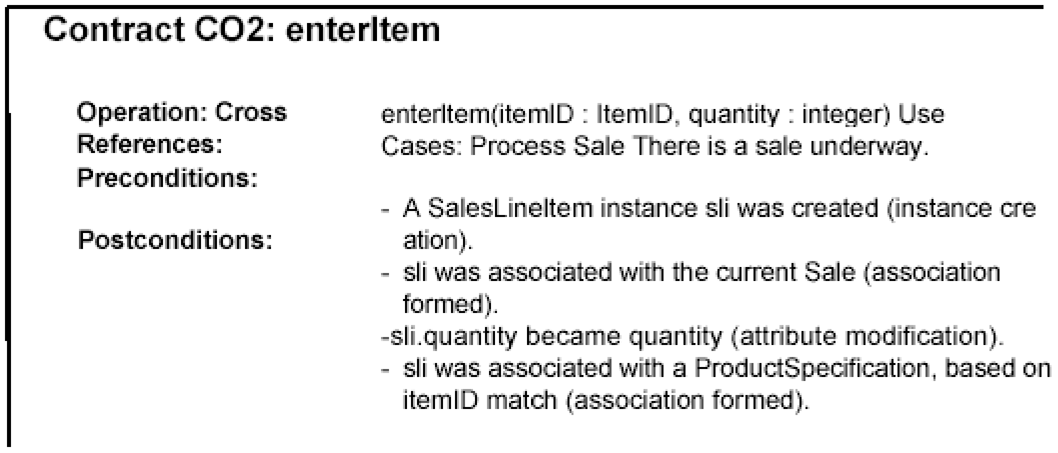
System operations handling input system events

Contract Sections



Operation: Cross	Name of operation, and parameters
References:	(optional) Use cases this operation can occur within
Preconditions:	Noteworthy *assumptions* about the state of the system or objects in the Domain Model before execution of the operation. These will not be tested within the logic of this operation, are assumed to be true, and are non-trivial assumptions the reader should know were made.
Postconditions:	-The state of objects in the Domain Model after completion of the operation. Discussed in detail in a following section.

Example Contract: enterItem


Contract CO2: enterItem

Operation: Cross	enterItem(itemID : ItemID, quantity : integer) Use
References:	Cases: Process Sale There is a sale underway.
Preconditions:	
	- A SalesLineItem instance sli was created (instance cre ation).
Postconditions:	- sli was associated with the current Sale (association formed).
	-sli.quantity became quantity (attribute modification).
	- sli was associated with a ProductSpecification, based on itemID match (association formed).

Post Conditions

The postconditions **describe changes in the state of objects** in the Domain Model and provide a detailed view of what the outcome of the operation must be

Domain Model state changes include **instances created, associations formed or broken, and attributes changed.**

(worse) Create a SalesLineItem

Postconditions are not actions to be performed, during the operation; rather, they are declarations about the Domain Model objects that are true when the operation has finished. ***after the smoke has cleared.***

Express postconditions in the **past tense**, as they are declarations about a state change in past.

(better) A SalesLineItem was created.

(worse) Create a SalesLineItem.

In other domains, when a loan is paid off or someone cancels their membership in something, associations are broken.

Think about postconditions using the following image: The system and its objects are presented on a theatre stage.

1. Before the operation, take a picture of the stage.
2. Close the curtains on the stage, and apply the system operation *(background noise of clanging, screams, and screeches...).*
3. Open the curtains and take a second picture.
4. Compare the before and after pictures, and express as postconditions the changes in the state

of the stage (A *SalesLineltem was created...).*

Guidelines: Contracts

Apply the following advice to create contracts:

To make contracts:

i. **Identify system operations** from the SSDs.

ii. For system operations that are complex and perhaps subtle in their results, or which are not clear in the use case, construct a contract.

iii. To **describe the postconditions**, use the following categories:

- **instance** creation and deletion
- **attribute** modification
- **associations** formed and broken

Advice on Writing Contracts

State the postconditions in a declarative, passive past tense form *(was* ...) to emphasize the declaration of a state change rather than a design of how it is going to be achieved. For example:

(better) A *SalesLineltem* was created. .

(worse) Create a *SalesLineltem.*

Remember to establish a memory between existing objects or those newly created by defining the forming of an association.

For example, it is not enough that a new *SalesLineltem* instance is created when the *enterltem* operation occurs. After the operation is complete, it should also be true that the newly created instance was associated with *Sale;*

thus:

The *SalesLineltem* was associated with the *Sale* (association formed).

Operations

An **operation is an abstraction**, not an implementation. By contrast, **a method (in the UML) is an implementation of an operation.**

A UML operation has a **signature** (name and parameters), and also an operation specification, which describes the effects produced by executing the operation; the postconditions.

A UML operation specification may not show an algorithm or solution, but **only the state changes or effects of the operation**.

Operations contracts expressed with the OCL

Associated with the UML is a formal language: Object constraint Language (OCL)

Which can be used to **express constraints in models**

The OCL **defines an official format for specifying pre and post conditions** for operations as :

System: makeNewSale()

Pre: <statements in OCL>

Post:

Contract and other artifacts

Pre-conditions

These define the **assumptions about the state of the system** at the beginning of the operation.

Some worth pre-conditions are

Things that are important to test in software at some point during execution of the operation.

Things that will not be tested, but upon which the success of the operation hinges

Post conditions

To summarize, the post conditions fall into these categories:

i. Instance creation and deletion.

ii. Attribute modification.

iii. Associations (to be precise, UML *links)* formed and broken.

Point of Sale Terminal-Case Study

- Customer arrives at the point of sale terminal with the items
- The cashier enters the items into the `Register` by noting the item id and its quantity
- The system searches the price of each `SalesLineItem` and computes its subtotal
- The system finally computes the Grand Total of the entire `Sale`
- The cashier enters the P`ayment` (amount tendered) made by the customer
- The system prints the receipt and displays the change amount.

i. *Instance Creation and Deletion*

After the *itemID* and *quantity* of an item have been entered, what new **object should have been created**? A *SalesLineltem.* Thus:

. A *SalesLineltem* instance *sli* was created (instance creation).

ii. *Attribute Modification*

After the itemID and quantity of an item have been entered by the cashier, what **attributes of new or existing objects should have been modified**? The *quantity* of the *SalesLineltem* should have become equal to the *quantity* parameter. Thus:

. *sliquantity* became *quantity* (attribute modification).

iii. *Associations Formed and Broken*

After the *itemID* and *quantity* of an item have been entered by the cashier, what **associations between new or existing objects should have been formed or broken**?
The new *SalesLineltem* should have been related to its *Sale,* and related to its *ProductDescription*. Thus:

. *sli* was associated with the current *Sale* (association formed).

. *sli* was associated with a *ProductDescription,* based on *ItemID* match (association formed).

A post-condition that breaks an association:

Consider an operation to allow the **deletion of line items**. The post-condition could read "The selected *SalesLineltem's* association with the *Sale* was broken." In other domains, **when a loan is paid off** or **someone cancels their membership** in something, associations are broken.

Contract CO1: makeNewSale

Operation:	makeNewSale()
Cross References:	Use Cases: Process Sale
Preconditions:	none
Postconditions:	- A Sale instance s was created (instance creation).
	- s was associated with a Register (association formed).
	- Attributes of s were initialized.

Contract CO2: enterItem

Operation:	enterItem(itemID: ItemID, quantity: integer)
Cross References:	Use Cases: Process Sale
Preconditions:	There is a sale underway.
Postconditions:	- A SalesLineItem instance sli was created (instance creation).
	- sli was associated with the current Sale (association formed).
	- sli.quantity became quantity (attribute modification).
	- sli was associated with a ProductDescription, based on itemID match (association formed).

Contract CO3: endSale

Operation:	endSale()
Cross References:	Use Cases: Process Sale
Preconditions:	There is a sale underway.
Postconditions:	- Sale.isComplete became true (attribute modification).

Contract CO4: makePayment

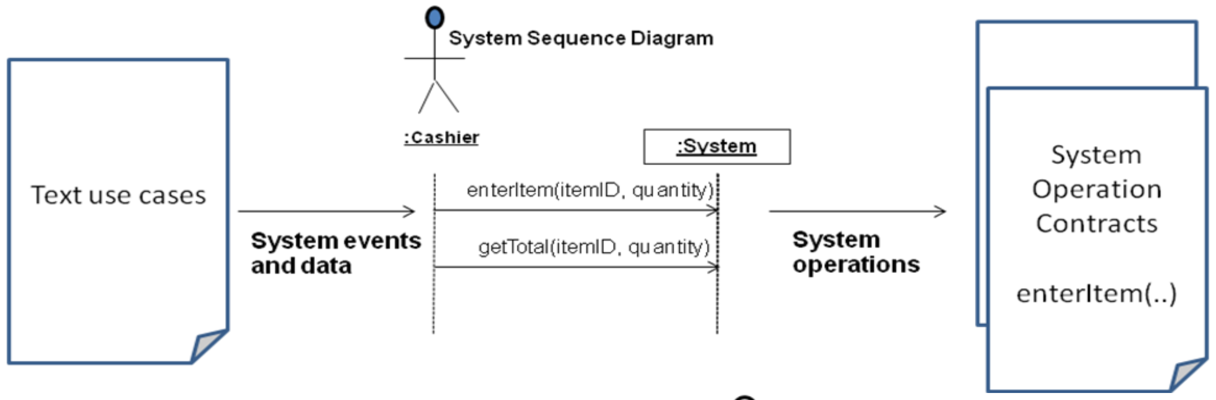
Operation:	makePayment(amount: Money)
Cross References:	Use Cases: Process Sale
Preconditions:	There is a sale underway.
Postconditions:	

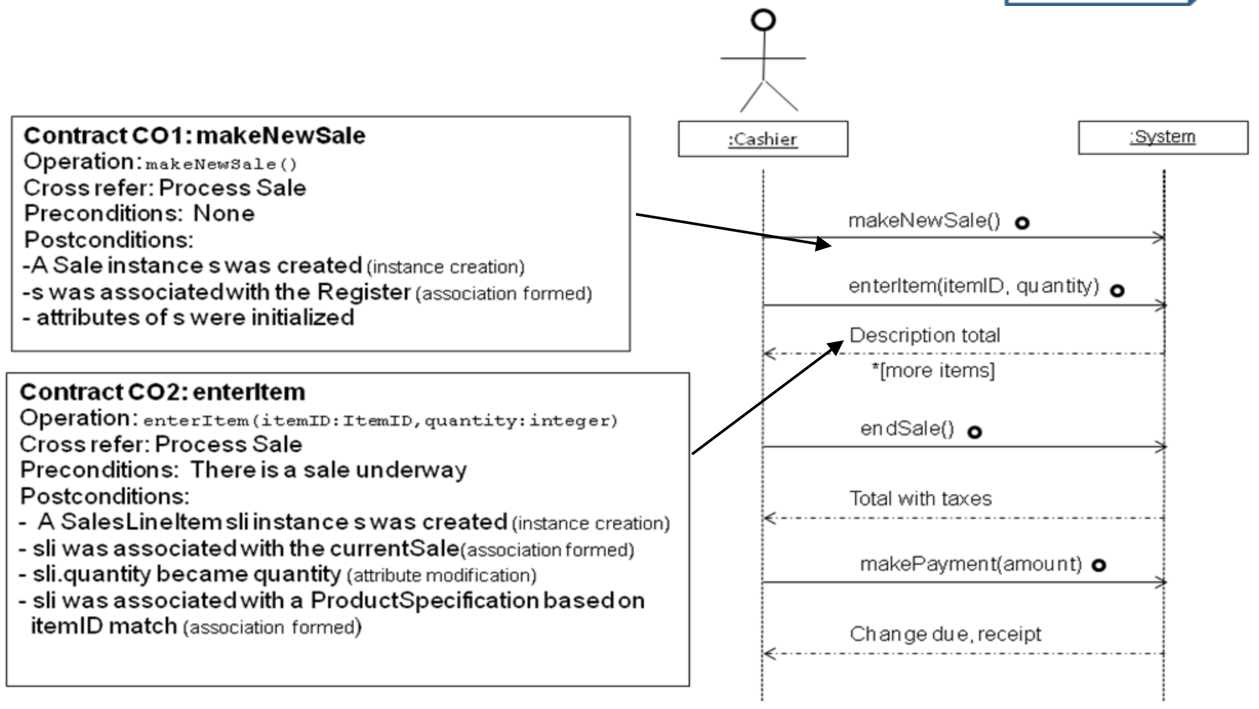
- A Payment instance p was created (instance creation).

- p.amountTendered became amount (attribute modification).

- p was associated with the current Sale (association formed).

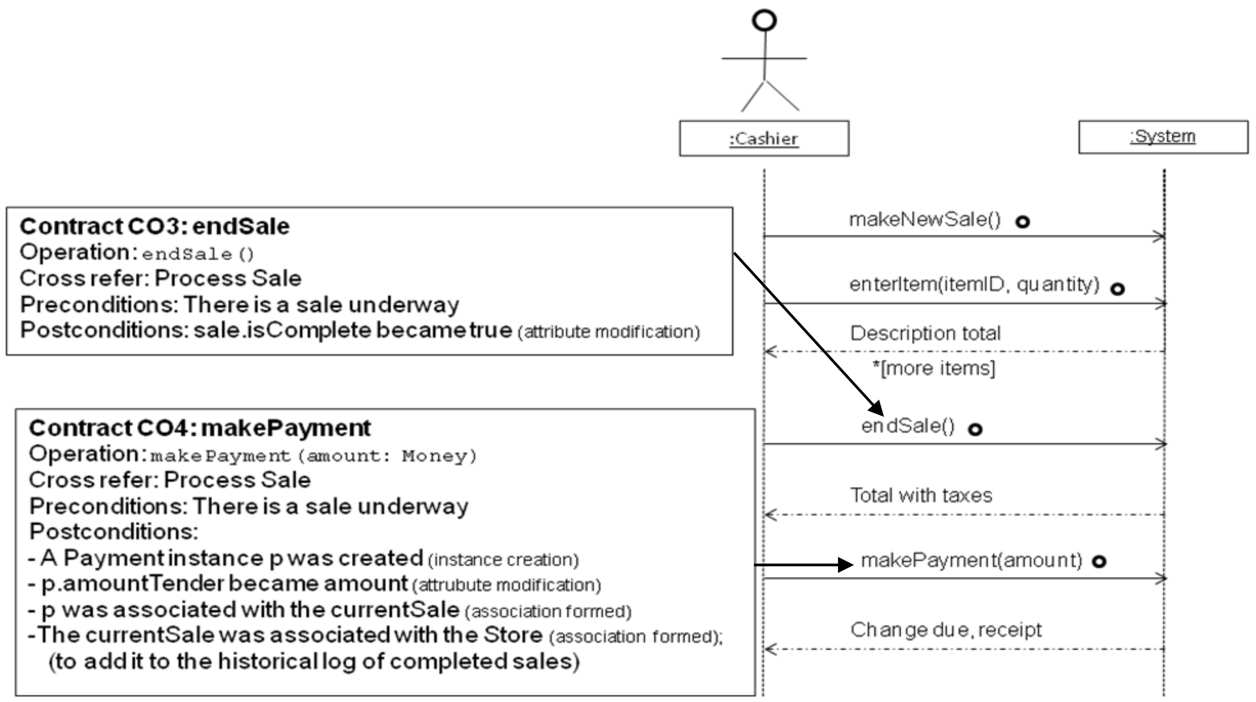
- The current Sale was associated with the Store (association formed); (to add it to the historical log of completed sales)





Contract CO1: makeNewSale
Operation: `makeNewSale()`
Cross refer: Process Sale
Preconditions: None
Postconditions:
-A Sale instance s was created (instance creation)
-s was associated with the Register (association formed)
- attributes of s were initialized

Contract CO2: enterItem
Operation: `enterItem(itemID:ItemID, quantity:integer)`
Cross refer: Process Sale
Preconditions: There is a sale underway
Postconditions:
- A SalesLineItem sli instance s was created (instance creation)
- sli was associated with the currentSale (association formed)
- sli.quantity became quantity (attribute modification)
- sli was associated with a ProductSpecification based on itemID match (association formed)



Contract CO3: endSale
Operation: `endSale()`
Cross refer: Process Sale
Preconditions: There is a sale underway
Postconditions: sale.isComplete became true (attribute modification)

Contract CO4: makePayment
Operation: `makePayment(amount: Money)`
Cross refer: Process Sale
Preconditions: There is a sale underway
Postconditions:
- A Payment instance p was created (instance creation)
- p.amountTender became amount (attrubute modification)
- p was associated with the currentSale (association formed)
-The currentSale was associated with the Store (association formed);
(to add it to the historical log of completed sales)