CHAPTER 1

OBJECT ORIENTED FUNDAMENTALS

CHAPTER 1

OBJECT ORIENTED FUNDAMENTALS

OBJECT ORIENTED SYSTEM (OO System)

The system where overall architecture is perceived as a collection of objects that are assigned specific responsibilities and hence exhibit their behavior based on the type of responsibility assigned to them.

Steps in OO System

i. **Identification of objects** existing in a domain

ii. **Assigning responsibilities** to these objects

iii. **Seek collaboration** between these objects for fulfilling the goal of the system

Decomposition

It is the process of partitioning the problem domain into smaller parts so that the overall complexity of the problem can be comprehended and tackled easily (divide and conquer approach)

- **Algorithmic/Functional decomposition**

  Dividing the problem domain into smaller parts by **focusing on the functionalities that the system provides**. For example functionalities of a Library Management System may include issue_book, return_book, search_book

- **Object Oriented Decomposition**

  Dividing the problem domain into smaller parts by **focusing on the objects with specific responsibilities that form the system**. For example, objects in a Library Management System may include IssueManager, Book, Librarian

Abstraction

Abstractions is defined as the process **hiding the complex details and representing only the necessary features.**

GOAL: C**ommunicating ideas with others** by **representing complexity in a much simplified form**.

Example:

-If you wish to withdraw money from ATM, the only concern is to receive money and not the internal workings of ATM.

Key Differences Between Structured and Object-Oriented Analysis and Design

	Structured	Object-Oriented
Methodology	SDLC	Iterative/Incremental
Focus	Processs	Objects
Risk	High	Low
Reuse	Low	High
Maturity	Mature and widespread	Emerging (1997)
Suitable for	Well-defined projects with stable user requirements	Risky large projects with changing user requirements

OBJECT-ORIENTED ANALYSIS (OOA)

- It is method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain.
- Methodology with an emphasis on finding and describing the objects—or concepts—in the problem domain.
- A thorough investigation of the problem domain is done during this phase by asking the **WHAT** questions (rather than how a solution is achieved)

OBJECT-ORIENTED DESIGN (OOD)

- Methodology with an emphasis on defining software objects and how they collaborate to fulfill the requirements
- A thorough investigation of the problem domain is done during this phase by asking the **HOW** questions (how a solution is achieved)

OOA vs OOD

OOA	OOD
Methodology with an emphasis on finding and describing the objects or concepts in the problem domain.	Methodology with an emphasis on defining software objects and how they collaborate to fulfill the requirements
Elaborate a problem	Provide conceptual solution
WHAT type of questions asked **"What the problem is about and what the system must do?"**	HOW type of questions asked **"How the problem is solved and how are the system goals achieved?"**
Perfomed by : Use Case Domain Model System Sequence Diagram (SSD)	Performed by: Interaction Diagrams Design Class Diagrams (DCD)
Q. What is required in the Library Information System? A. Authentication!!	Q. How is Authentication in the Library Information System achieved? A. Thru Smart Card!! Or Fingerprint!!

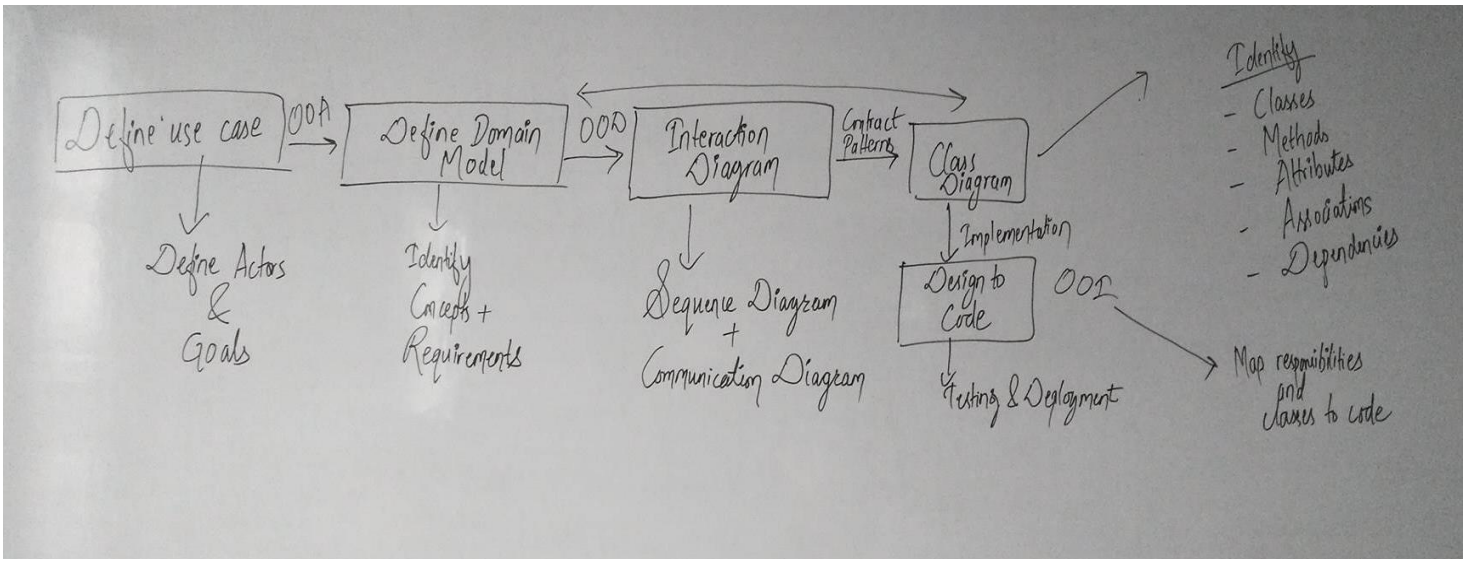


Fig: MODELS IN OODM

(Now you know why OODM is important, right? ☺)

REQUIREMENTS

Capabilities and conditions to which the system/ **project must conform**

Types of requirements:

FUNCTIONAL (BEHAVIORAL)

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

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 requirement)

Functionality—features, capabilities, security.

Usability—human factors, help, documentation.

Reliability—frequency of failure, recoverability, predictability

Performance/ **Productivity**—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.

REQUIREMENT PROCESS

1) 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 through Document review, observation, questionnaire, surveys, sampling, research.

2) 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

3) Open-ended questions

–useful when not much is known yet , broad-oriented, free of context.

4) 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!!

Find out who else to interview

–who else uses the system

–who interacts with you

–who will agree / disagree with you

5) 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

–set an agenda and maintain it

–don’t get mired in technical detail

USE CASES: DESCRIBING PROCESSES

Definition: - It is a **narrative document** that describes the sequence of events of an actor (an external agent) using a system to complete a process.

-Writing use cases, stories of using a system, is an excellent technique to understand and describe requirements. Informally, they are stories of using a system to meet goals.

-The essence is discovering and recording functional requirements by writing stories of using a system to help fulfill various stakeholder goals

-"Use cases are narrative description of domain processes"

Actor

It is an **entity that is external** to the system, who in some way participates in the story of Use Case.

An Actor typically stimulates the system with input events, or receives something form it. Actors are represented by the role they play in the Use Case.

Actors can include

- Roles that people play
- Computer System
- Electrical or Mechanical Devices

Types of Actors

PRIMARY ACTORS have user goals fulfilled thru using services of system under discussion. e.g. cashier . These are identified to find user goals that drive use cases

SUPPORTING ACTORS provide service or information to the system. e.g. automated payment authorization. These are identified to clarify external interfaces and protocols

OFFSTAGE ACTORS has interest in the behavior of the use case . e.g. government tax agency. These are identified to ensure all interests are identified and satisfied

Scenario

A scenario is a specific sequence of actions and interactions between actors and the system under discussion; it is also called a use case instance.

A use case is a collection of related success and failure scenarios that describes an actor using a system to support a goal

Use cases are functional or behavioral requirements that indicate what the system will do

TYPES: ESSENTIAL VS CONCRETE/REAL USE CASES

Essential Use Cases:

- All expanded use cases, remain relatively **free of technology and implementation details**
- Design decisions are deferred and abstracted. That is, only essential activities and motivations
- High level use cases are always essential in nature due to their brevity and abstraction.

Actor Action	System Response
1. The customer **identifies** themselves	2. Presents options
3. and so on	4. and so on

Concrete/Real Use Cases:

- Concretely describes the process in terms of its real current design, **committed to specific I/O technology** etc.
- This helps the designer to identify what task the interaction diagram may fulfill, in addition to what is in contract

Actor Action System	**Response**
1. The Customer **inserts their card**	2. Prompts for **PIN**
3. Enters PIN on key pad	4. Display options menu.
5.and so on.	6. and so on.

Essential Use Case

Actor Action System	**Response**
1. The Cashier **records the identifier**.	
If there is more than one of the same item,	
the Cashier can enter the quantity as well.	2. Determines the item price from each item and adds the item information to the running sales transaction.
3. and so on	4. and so on

Concrete/Real Use Case

Actor Action System	**Response**
1. For each item, the Cashier types in the	
Universal Product Code (UPC) in the UPC	
input field of Window1. They then press the	
" Enter Item" button with the mouse OR by	
pressing the key	2. Displays the item price and adds the item information to the running sales transaction. The description and price of the current item are **displayed in Textbox2 of Window1**.

3. and so on

4. and so on

THREE USE CASE FORMATS

- Brief (High Level)
- Casual
- Fully Dressed

1. **Brief**—terse one-paragraph summary, usually of the main success scenario.

It describes a process **very briefly**, usually in two or three sentences.

It is useful to create this type of use case **during the initial requirements and project scoping** in order quickly understand the degree of complexity and functionally in a system.

They are very **terse and vague** on design decisions.

Case Study: Buy Goods at Supermarket

: A customer arrives at a checkout with items to purchase. The cashier uses the POS system to record each purchased item. The system presents a running total and line-item details. The customer enters payment information, which the system validates and records. The system updates inventory. The customer receives a receipt from the system and then leaves with the items

It is useful to start with high level use cases to quickly obtain some understanding of overall major processes.

2. **Casual**—informal paragraph format. Multiple paragraphs that cover various scenarios.

Main Success Scenario: A customer arrives at a checkout with items to return. The cashier uses the POS system to record each returned item ...

Alternate Scenarios:

If the credit authorization is reject, inform the customer and ask for an alternate payment method.

If the item identifier is not found in the system, notify the Cashier and suggest manual entry of the identifier code (perhaps it is corrupted).

If the system detects failure to communicate with the external tax calculator system, ...

3. **Fully dressed**—the most elaborate. All steps and variations are written in detail, and there are supporting sections, such as preconditions and success guarantees.

These detailed use cases are written after many uses cases have been identified and written in a brief format

Template:

Use Case Section	Comment
Use Case Name	Start with a verb.
Scope	The system under design.
Level	"user-goal" or "subfunction"
Primary Actor	Calls on the system to deliver its services.
Stakeholders and Interests	Who cares about this use case, and what do they want?
Preconditions	What must be true on start, *and* worth telling the reader?
Success Guarantee	What must be true on successful completion, *and* worth telling the reader.
Main Success Scenario	A typical, unconditional happy path scenario of success.
Extensions	Alternate scenarios of success or failure.
Special Requirements	Related non-functional requirements.
Technology and Data Variations List	Varying I/O methods and data formats.
Frequency of Occurrence	Influences investigation, testing, and timing of implementation.
Miscellaneous	Such as open issues.

Use Case UC1: Buy Goods
Scope: NextGen POS Application
Level: User goal
Primary Actor: Customer
Stakeholders and Interests:
Cashier: Wants accurate, fast entry, and no payment errors, as cash drawer short ages are deducted from his/her salary.
- Salesperson: Wants sales commissions updated.
Customer: Wants purchase and fast service with minimal effort. Wants proof of purchase to support returns.

Preconditions: Cashier is identified and authenticated.
Success Guarantee (Postconditions): Sale is saved. Tax is correctly calculated.
Accounting and Inventory are updated. Commissions recorded. Receipt is generated. Payment authorization approvals are recorded.
Main Success Scenario (or Basic Flow):
1. Customer arrives at POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.

Extensions (or Alternative Flows):

*a. At any time, System fails:

To support recovery and correct accounting, ensure all transaction sensitive state and events can be recovered from any step of the scenario.

1. Cashier restarts System, logs in, and requests recovery of prior state.
2. System reconstructs prior state.

2a. System detects anomalies preventing recovery:

1. System signals error to the Cashier, records the error, and enters a clean state.

Frequency of Occurrence: Could be nearly continuous.

Miscellaneous:

- What are the tax law variations?
- Explore the remote service recovery issue.
- What customization is needed for different businesses?
- Must a cashier take their cash drawer when they log out?
- Can the customer directly use the card reader, or does the cashier have to do it?

GUIDELINES FOR DEFINING USE CASES

(Finding Primary Actors, Goals, and Use Cases)

1. Choose the **system boundary**.

   Is it just a software application, the hardware and application as a unit, that plus a person using it, or an entire organization?

2. Identify the **primary actors**.

   Those that have user goals fulfilled through using services of the system.

3. For each, identify their **user goals**.

   Raise them to the highest user goal level that satisfies the EBP guideline.

4. Define **use cases** that satisfy user goals; **name them** according to their goal.

   Usually, user goal-level use cases will be one-to-one with user goals.

System and their boundaries:

A use case describes interaction with the system. Typical boundaries includes

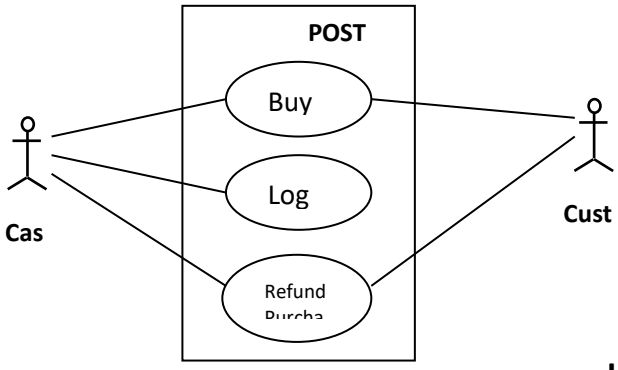
- Hardware/software boundary of a device or computer system.
- Department of an organization.
- Entire organization.

The system boundary is important as it identifies:

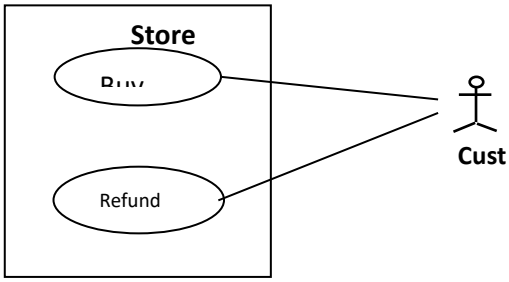
- what is external versus internal.
- the responsibilities of the system

Scenario one:

If we choose the entire store or business as the system then the only customer is an actor not the cashier because the cashier acts as a resource within the business system.



Use case and actors when the POST



Use case and actors when the Store is the

Scenario two:

If we choose the point of sale hardware and software as the system both cashier as well as customer may be treated as actors.

METHODS OF IDENTIFYING THE USE CASES

Method 1 (Actor based)

Identifying the actor related to a system or organization and then for each actor, identifying the processes they initiate or participate in.

Using actor goal list

A recommended procedure:

1. Find the user goals.
2. Define a use case for each.

Actor	Goal
Cashier	process sales, process rentals, handle return, cash in , cash out
Manager	start up , shut down
System Admin	add users, modify users, delete users, manage security, manage system tables
Sales Activity Sys.	analyze sales and performance data
Customer	Buy Items, Refund Items

Method 2 (External events based)

Identifying the external events that a system must respond to and then relating the events to actor and use cases.

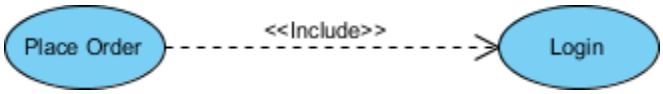
External Event	Actor	Goal/use case
Enter sale line item	Cashier	process a sale
Enter payment	Cashier or customer	process a sale

USE CASE TERMINOLOGIES

(RELATIONSHIPS IN USE CASE / REUSING USE CASE)

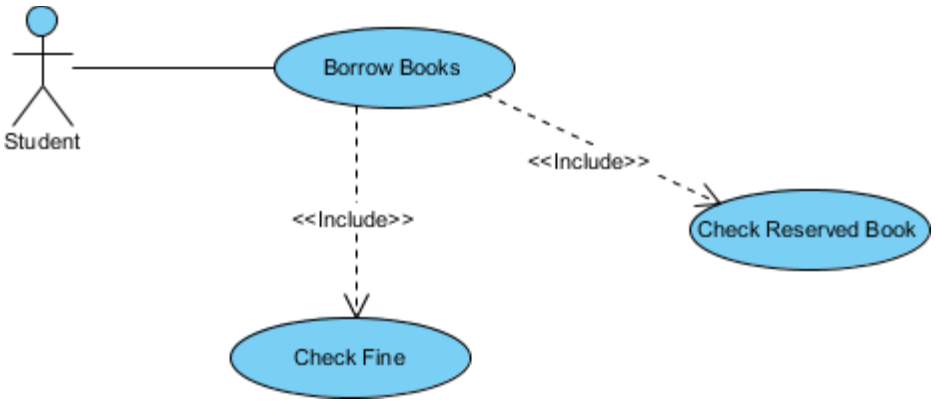
Include

- When a use case is depicted as using functionality of another functionality of another use case, this relationship between the use cases is named as an “include” or “uses” relationship.
- A use case includes the functionality described in another use case as a part of its business process flow.
- The include relationship adds additional functionality not specified in the base use case. The <<Include>> relationship is used to include common behavior from an included use case into a base use case in order to support re-use of common behavior.
- An include relationship is depicted with a directed arrow having a dotted line. The tip of arrowhead points to the child use case and the parent use case connected at base of the arrow.
- The stereotype "<<include>>" identifies the relationship as an include relationship.



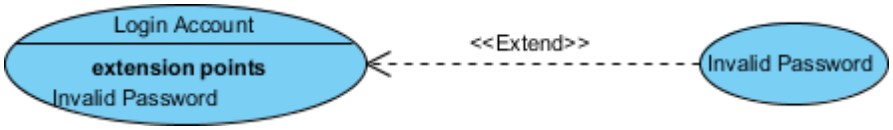
(<<includes>> Think it as a Precondition just to understand, ok ☺)

Use Case Example - Include Relationship



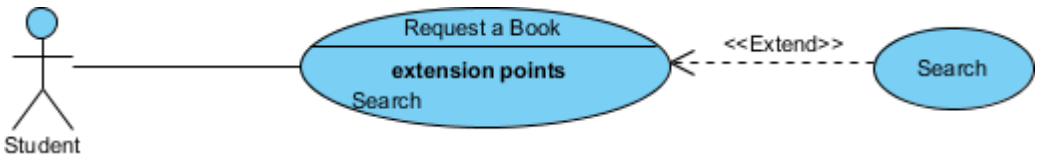
Extends

- The extend relationships are important because they show optional functionality or system behavior. The <<extend>> relationship is used to include optional behavior from an extending use case in an extended use case.
- Indicates that an **"Invalid Password"** use case may include (subject to specified in the extension) the behavior specified by base use case **"Login Account"**.
- Depict with a directed arrow having a dotted line. The tip of arrowhead points to the base use case and the child use case is connected at the base of the arrow.
- The stereotype "<<extends>>" identifies as an extend relationship



Use Case Example - Extend Relationship

Take a look at the use case diagram example below. It shows an extend connector and an extension point "Search".



(<<extends>> Think it as a Postcondition)

(EXAMPLES: REFER TO THE EXAMPLES DONE IN THE CLASS TODAY ☺)

The UML

The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems.

The UML has emerged as the standard diagramming notation for object-oriented modeling.

The UML is a language for

-visualizing **-constructing**

-specifying **-documenting**

1. UML is a language for visualizing

It is very significant to model the system and Modeling requires certain degree of visualization of the system.

If the developer directly codes the requirement of a system, he can have the following problems.

Communicating these conceptual models to others. As all speak different language, there are chances of error and understanding the system.

Something in a software cannot be understood unless they build modules that transcend the textual programming language.

If a developer never wrote down the models that are in his head such information is lost forever from the implementation once the developer moved on.

In a system development some things are best modeled textually, while some are best modeled graphically.

Thus UML is such a language that is more than just a bunch of graphical symbols. Each symbol in contrast has a well-defined semantics which is best for visualization

2. It is a language for specifying.

Specifying means building models that are **precise, unambiguous and complete**.

In particular UML addresses the specification of all the important analysis, design and implementation decisions that must be:

-Scientific -Distributed web-based services.

3. UML is a language for constructing.

The models of UML can directly be **connected to a variety of programming languages** i.e . It is possible to map a UML model to a programming language such as Java, C++, etc.

4. UML is a language for documenting.

A healthy software organization produces **all sorts of artifacts** in addition to executable code.

Their artifacts include:

Requirements	Source code	Prototypes
Architecture	Project plans	Release
Design	Tests	

The UML addresses the **documentation of a system's architecture** and all of its details. The UML also provides the language for **expressing requirements** and for test. Finally, the UML provides a language for **modeling the activities of a project planning and release management**

UML is not:

- A **visual programming language** or environment
- A **database specification** tool
- A **development process** (i.e. an SDLC)
- A **panacea** (The ultimate solution)
- A **quality guarantee**

UML IMPORTANCE:

(Explain Definition according to marks :D)

Helps to reduce cost and time-to-market.

Helps managing a complex project architecture.

Helps to convey ideas between developers\designers\etc.

The UML: Terms and Concepts

A **system** is a collection of subsystems organized to accomplish a purpose and described by a set of models, possibly from different viewpoints.

A **subsystem** is a grouping of elements, of which some constitute a specification of the behavior offered by the other contained **elements**.

A **model** is a semantically closed abstraction of a system, meaning that it represents a complete and self-consistent simplification of reality, created in order to better understand the system.

A **view** is a projection into the organization and structure of a system's model, focused on one aspect of that system.

A **diagram** is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships).

To understand UML we require three major elements learning; the UML's building blocks, the rules that dictate how those building blocks may be put together and some common mechanism that apply throughout the UML.

The vocabulary of UML encompasses three kinds of building blocks:

- Things.
- Relationship.
- Diagrams

Things: Those are abstractions that are first class citizens in UML modeling.

There are four kinds of things in UML.

- Structural things.
- Behavioral things.
- Grouping things
- Annotational things.

Structural things:-

These are the nouns of the UML models. They are the static part of the model representing elements that are either conceptual or physical.

- Active Class
- Component
- Node etc.

Behavioral things

- Interaction
- State machine etc.

Grouping things

- Package

Annotational things

- Note

Relationships in UML

- Dependency
- Association
- Generalization
- Realization

3 WAYS TO APPLY UML (5marks Short Note)

- ## UML as sketch

  Informal and incomplete diagrams (often hand sketched on whiteboards) created to explore difficult parts of the problem or solution space, exploiting the power of visual languages.

- ## UML as blueprint

  Relatively detailed design diagrams used either for

  1) reverse engineering to visualize and better understanding existing code in *UML* diagrams, or
  2) code generation (forward engineering).

  - In reverse engineering, a *UML* tool reads the source or binaries and generates (typically) *UML* package, class, and sequence diagrams. These "blueprints" can help the reader understand the big-picture elements, structure, and collaborations.

  - Before programming, some detailed diagrams can provide guidance for code generation (e.g., in Java), either manually or automatically with a tool. It's common that the diagrams are used for some code, and other code is filled in by a developer while coding (perhaps also applying *UML* sketching).

- ## UML as programming language

  Complete executable specification of a software system in *UML*. Executable code will be automatically generated, but is not normally seen or modified by developers; one works only in the *UML* "programming language." This use of *UML* requires a practical way to diagram all behavior or logic (probably using interaction or state diagrams), and is still under development in terms of theory, tool robustness and usability.

UML PERSPECTIVES (3 perspectives to apply UML)- ShortNote 5marks

The *UML* describes raw diagram types, such as class diagrams and sequence diagrams. It does not superimpose a modeling perspective on these. For example, the same *UML* class diagram notation can be used to draw pictures of concepts in the real world or software classes in Java.

- **Conceptual.**

the diagrams are interpreted as describing things in a situation of the real world or domain of interest.

If you take the conceptual perspective, you draw a diagram that represents the concepts in the domain under study. These concepts will naturally relate to the classes that implement them, but there is often no direct mapping. Indeed, a conceptual model should be drawn with little or no regard for the software that might implement it, so it can be considered language-independent.

- **Specification.**

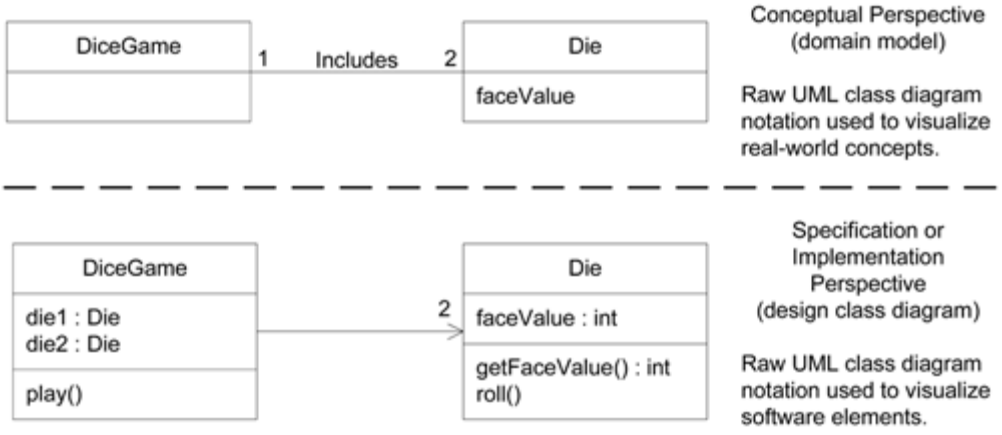
the diagrams (using the same notation as in the conceptual perspective) describe software abstractions or components with specifications and interfaces, but no commitment to a particular implementation (for example, not specifically a class in C# or Java).

Now we are looking at software, but we are looking at the interfaces of the software, not the implementation. Object-oriented development puts a great emphasis on the difference between interface and implementation, but this is often overlooked in practice because the notion of class in an OO language combines both interface and implementation. This is a shame, **because the key to effective OO programming is to program to a class's interface rather than to its implementation**.

- **Implementation.**

the diagrams describe software implementations in a particular technology (such as Java).

In this view, we really do have classes and we are laying the implementation bare. This is probably the perspective used most often, but in many ways the specification perspective is often a better one to take.



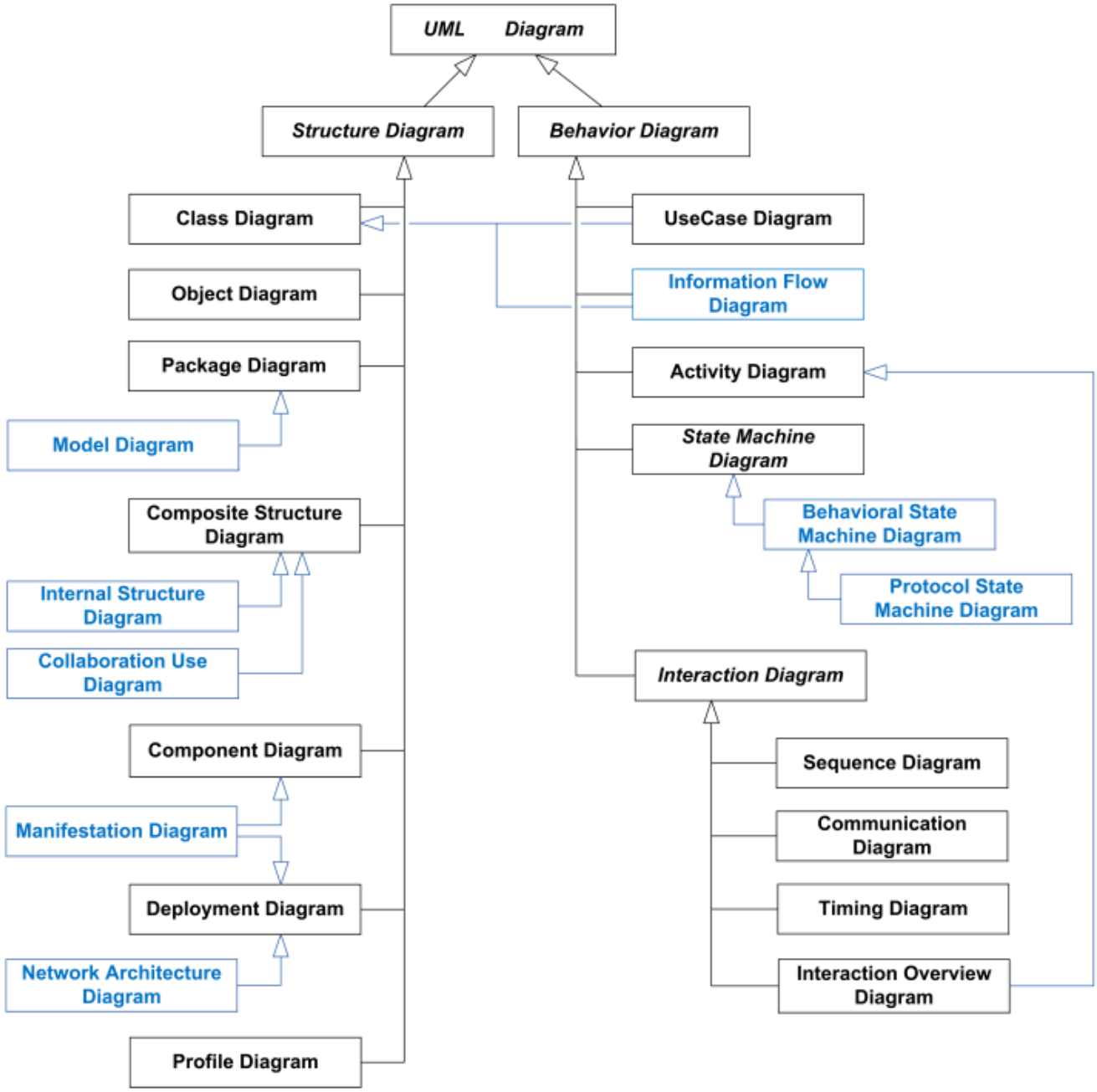
Question : List Notations used in UML- 7 marks (Refer to class notes)

UML DIAGRAMS

Classification of UML Diagrams

UML specification defines two major kinds of UML diagram: **structure diagrams** and **behavior diagrams**

UML diagrams could be categorized hierarchically as shown below:



A. STRUCTURAL DIAGRAMS

Static aspects of a software system : classes, interfaces, components, and nodes.

(STATIC; – features **that do not change** with time.)

Used to describe the **building blocks** of the system

These diagrams answer the question – **What's there?**

Structure diagram shows static structure of the system and its parts on different abstraction and implementation levels and how those parts are related to each other.

The elements in a structure diagram represent the meaningful concepts of a system, and may include abstract, real world and implementation concepts.

Structure diagrams are not utilizing time related concepts, do not show the details of dynamic behavior. However, they may show relationships to the behaviors of the classifiers exhibited in the structure diagrams.

1. Class diagram

It is a static structure diagram which describes structure of a system at the level of classifiers (classes, interfaces, etc.).

It shows some classifiers of the system, subsystem or component, different relationships between classifiers, their attributes and operations, constraints

Class diagrams commonly contain the following things:

- Classes
- Interfaces
- Collaborations
- Dependency, generalization, and association relationships

Class diagrams are static – display **what interacts** but **not what happens when interaction occurs.**

May contain notes and constraints.

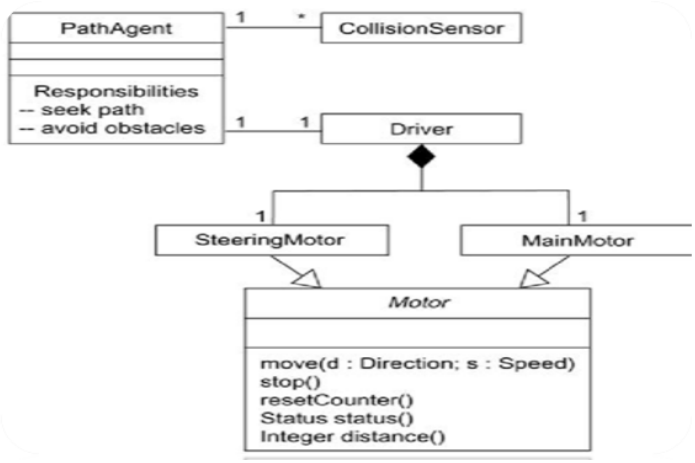
They are used :

1. To model the **vocabulary of a system**

Making a decision about which abstractions are apart of the system under consideration and which fall outside its boundaries. Used to specify these abstractions and their responsibilities.

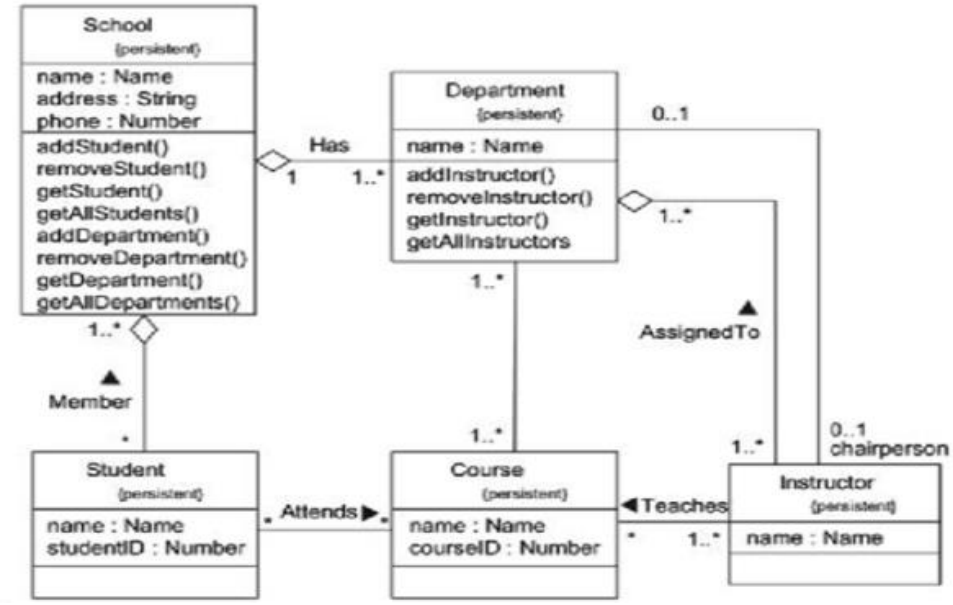
2. To model simple **collaborations**

A collaboration is a society of classes, interfaces, and other elements that work together to provide some cooperative behavior



3. To model a **logical database schema**

The blueprint for the conceptual design of a database.



Classes are represented by a rectangle divided to three parts: class name, attributes and operations.

Attributes are written as:

visibility name [multiplicity] :type-expression =initial-value

Operations are written as:

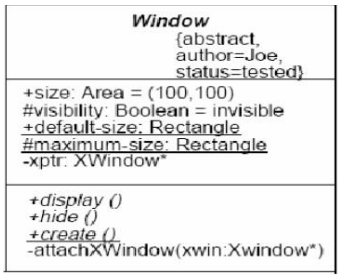
visibility name (parameter-list) :return type-expression

Visibility is written as:

+public

#protected

-private



Class Diagram Relationships

Association –

Two classes are associated if one class has to know about the other. (uses)

Aggregation –

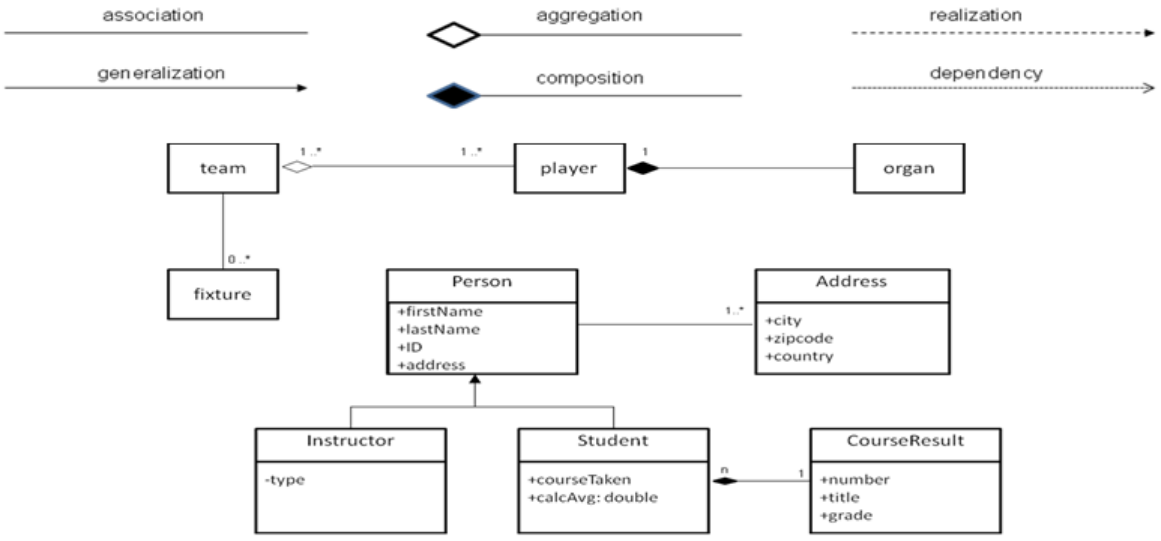
An association in which one class belongs to a collection in the other.(whole part)

Generalization –

An inheritance link indicating one class is a base class of the other.

Dependency –

A labeled dependency between classes (such as friend, classes)



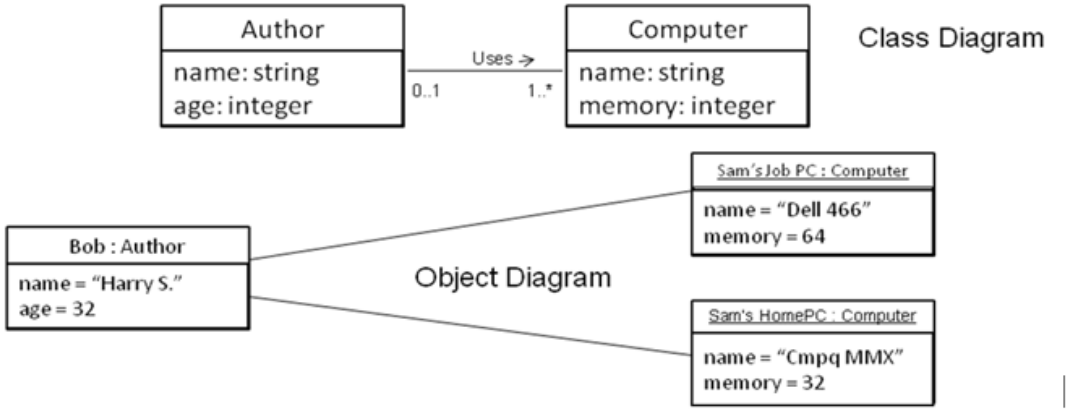
2. Object Diagram

Now obsolete, is defined as "a graph of instances, including objects and data values. A static object diagram is an instance of a class diagram; it shows a snapshot of the detailed state of a system at a point in time."

An *object diagram* shows a set of **objects and their relationships**.

Used to illustrate **data structures, the static snapshots of instances** of the things found in class diagrams.

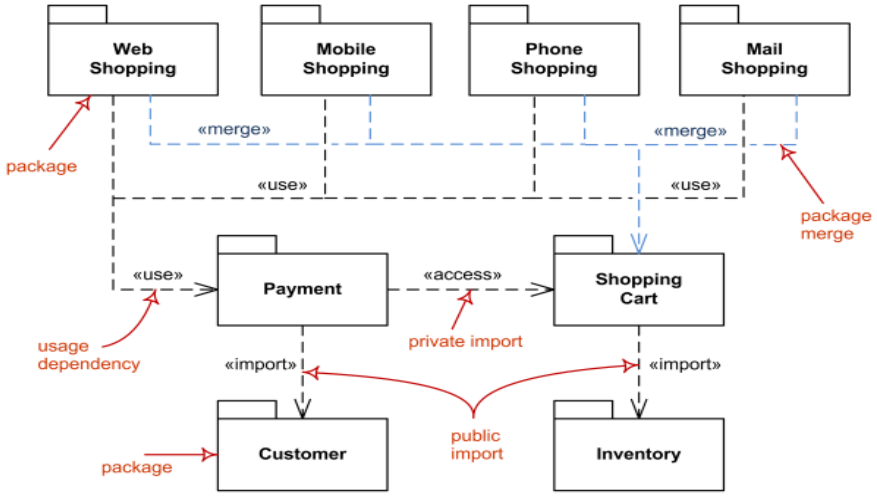
Object diagrams address the static design view or static process view of a system just as do class diagrams, but from the **perspective of real** or **prototypical cases.**



3) Package diagram

Shows packages and relationships between the packages.

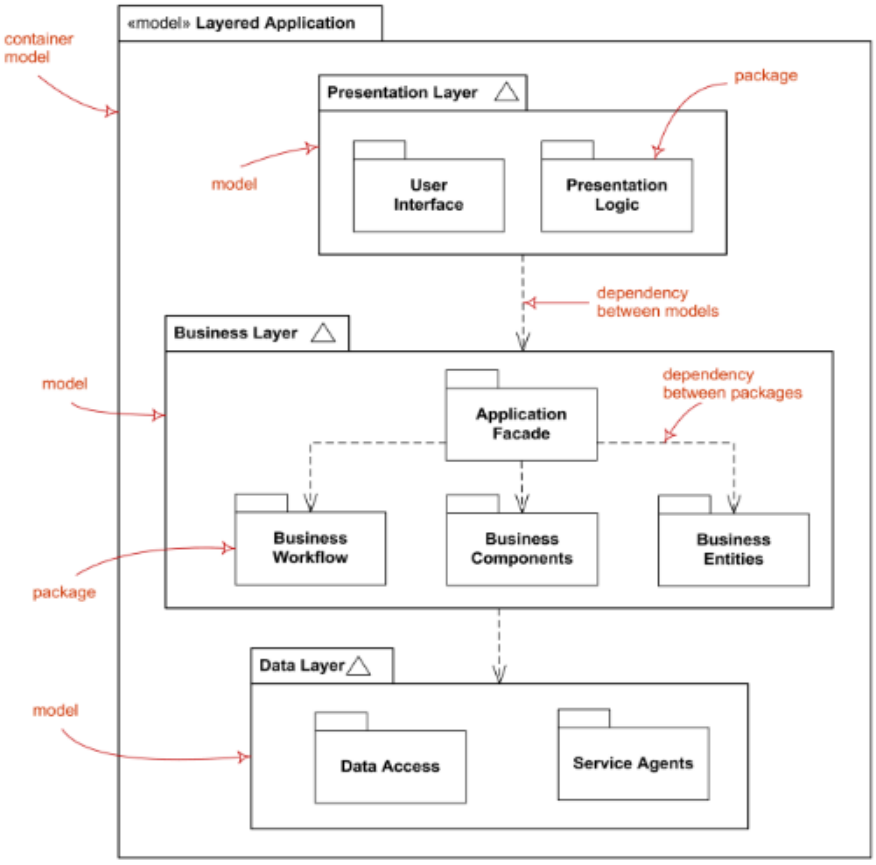
Package diagram includes the following elements: package, packageable element, dependency, element import, package import, package merge.



Model diagram

It is UML auxiliary structure diagram which shows some abstraction or specific view of a system, to describe architectural, logical or behavioral aspects of the system. It could show, for example, architecture of a multi-layered (aka multi-tiered) application - multi-layered application model.

In the diagram below, Layered Application is a "container" model which contains three other models - Presentation Layer, Business Layer, and Data Layer. There are dependencies defined between these contained models.

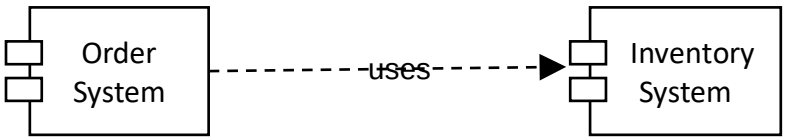


5. Component Diagram

Shows a set of **components and their relationships.**

Used to illustrate the **static implementation view** of a system.

Component diagrams are related to class diagrams in that a component **typically maps to one or more classes, interfaces, or collaborations**.



6. Deployment Diagram

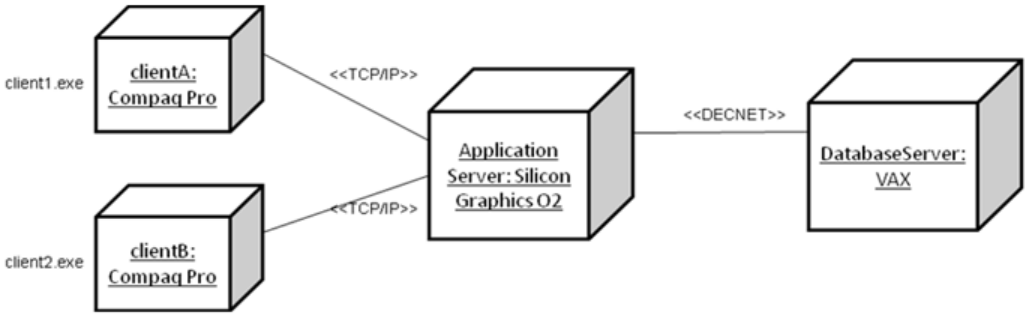
A *deployment diagram* shows a **set of nodes and their relationships**.

Used to illustrate the static deployment **view of an architecture**.

Deployment diagram shows architecture of the system as deployment (distribution) of software artifacts to deployment targets.

Specification level deployment diagram (also called type level) shows some overview of deployment of artifacts to deployment targets, without referencing specific instances of artifacts or nodes.

Instance level deployment diagram shows deployment of instances of artifacts to specific instances of deployment targets. It could be used for example to show differences in deployments to development, staging or production environments with the names/ids of specific build or deployment servers or devices.
While component diagrams show components and relationships between components and classifiers, and deployment diagrams - deployments of artifacts to deployment targets



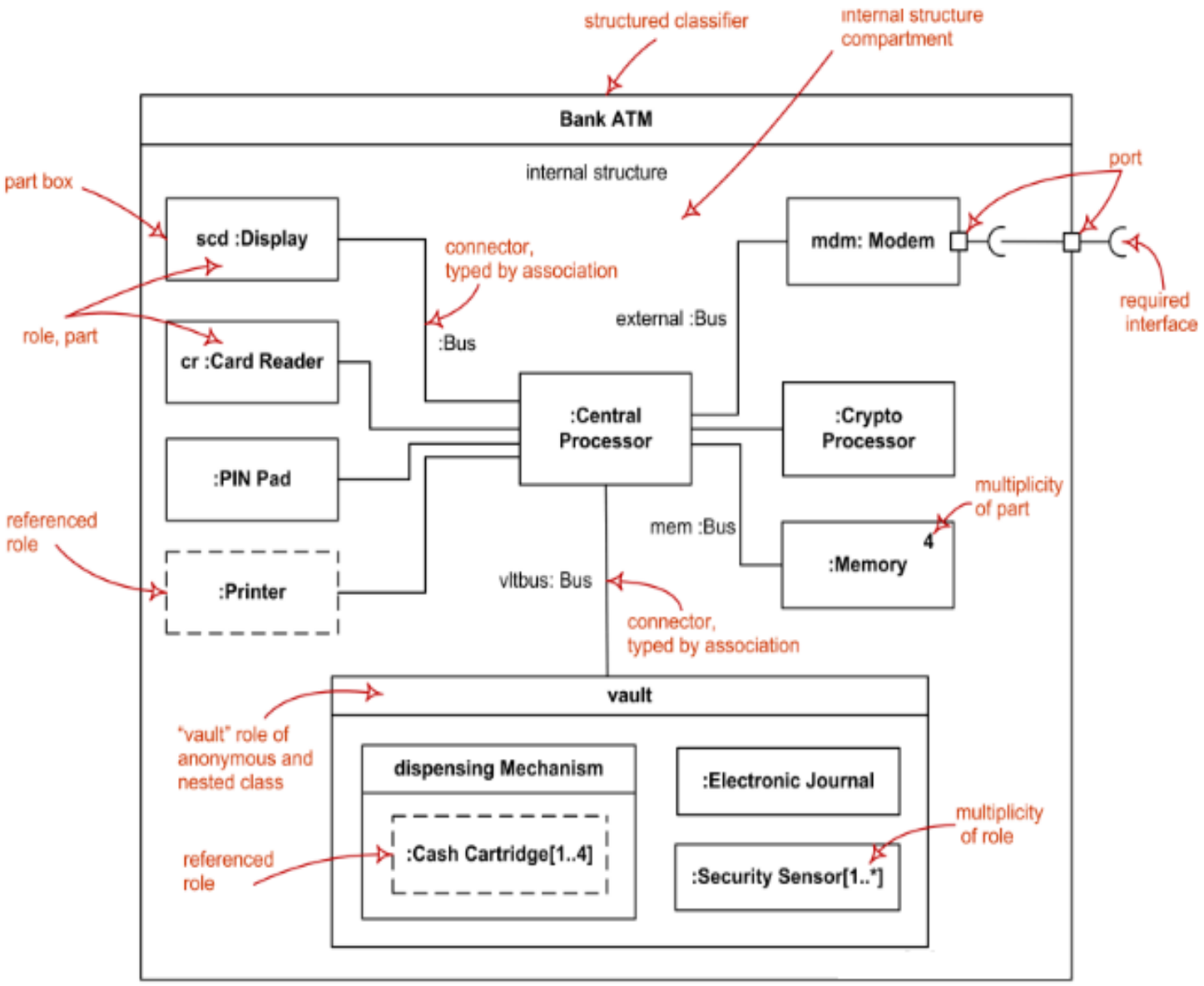
7. Composite structure diagram

It could be used to show:

- Internal structure of a classifier
- A behavior of a collaboration

a. Internal Structure diagrams show internal structure of a classifier - a decomposition of the classifier into its properties, parts and relationships.

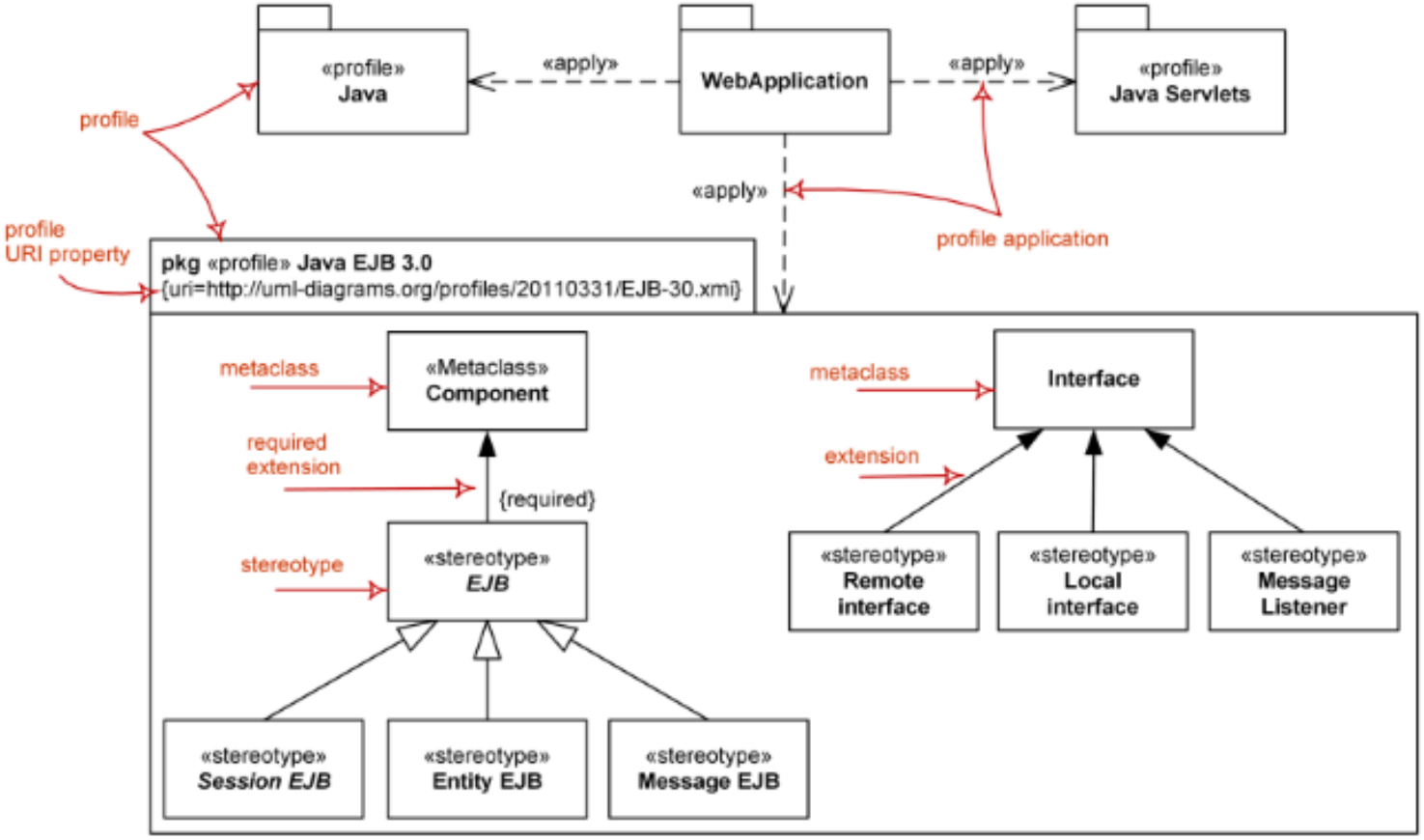
A **composite structure diagram** that shows internal structure of a classifier includes the following elements: class, part, port,connector, usage.


structured classifier
internal structure compartment
Bank ATM
internal structure
part box
scd :Display
connector, typed by association
:Bus
mdm: Modem
port
required interface
external :Bus
role, part
cr :Card Reader
:Central Processor
:Crypto Processor
:PIN Pad
multiplicity of part
4
referenced role
mem :Bus
:Memory
:Printer
vltbus: Bus
connector, typed by association
vault
"vault" role of anonymous and nested class
dispensing Mechanism
:Electronic Journal
multiplicity of role
referenced role
:Cash Cartridge[1..4]
:Security Sensor[1..*]

8. Profile diagram

It is auxiliary UML diagram which allows defining custom stereotypes, tagged values, and constraints. The Profile mechanism has been defined in UML for providing a lightweight extension mechanism to the UML standard. Profiles allow to adapt the UML metamodel for different

- platforms (such as J2EE or .NET), or
- domains (such as real-time or business process modeling).



B. BEHAVIORAL DIAGRAMS

The UML's behavioral diagrams are used to visualize, specify, construct, and document **the dynamic aspects of a system**.

Dynamic aspects of a system represent **its changing parts.**

They show the dynamic behavior of the objects in a system, which can be described as a series of changes to the system over time.

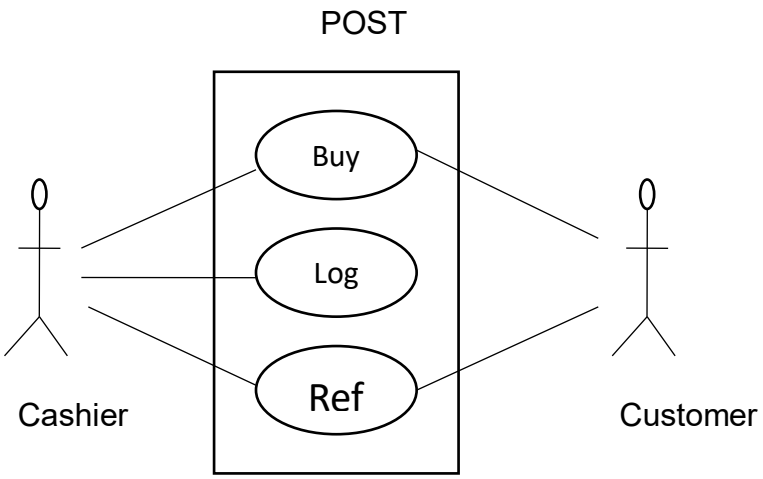
Used to show **how** the system evolves over time (responds to requests , events etc)
The UML's behavioral diagrams are roughly organized around the major ways you can model the dynamics of a system.

1. Use Case Diagram

A *use case diagram* shows **a set of use cases and actors** (a special kind of class) and their relationships.

Use case diagrams are especially important in organizing and modeling the behaviors of a system.

Describes **what a system does** from the standpoint of an external observer.



Emphasis on what *a system does* **rather than how.**

Scenario – Shows what happens **when someone interacts with system.**

Actor – A user or another system that interacts with the modeled system.

A use case diagram describes **relationships between actors and scenarios.**

Provides system requirements **from the user's point of View.**

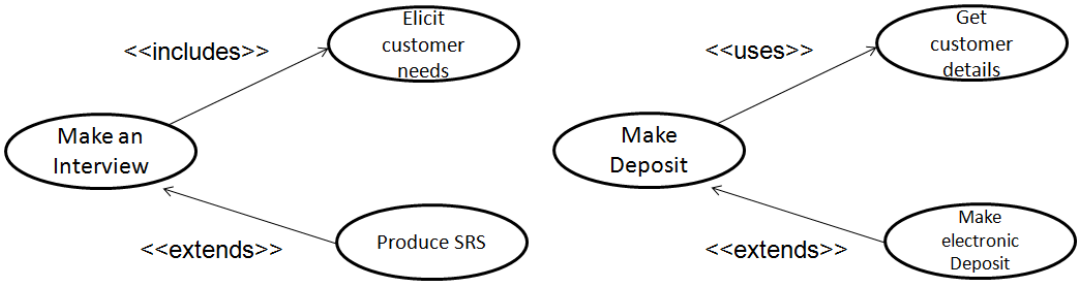
Use case Relationships:

Association – defines a relationship between an actor and a use case.

Extend - defines that instances of a use case may be augmented with some additional behavior defined in an extending *use case.*

Use/Include - drawn as a **dependency relationship** that points from the base use case to the used use case.

- defines that a use case uses a behavior defined in another use case.



Use Case Diagram

- Attention **focused on the part of the business process** that is going to be supported by the IS.
- It is the **end-user perspective** model.
- It is **goal driven**
- It helps to **identify system services.**
- It is **not used as DFDs**.
- **Sequences, branching, loops, rules**, etc. cannot (and should not) be directly expressed.

They are used to describe a set of actions (use cases) that some system or systems (subject) should or can perform in collaboration with one or more external users of the system (actors) to provide some observable and valuable results to actors or other stakeholders of the system(s).

2. Activity Diagram

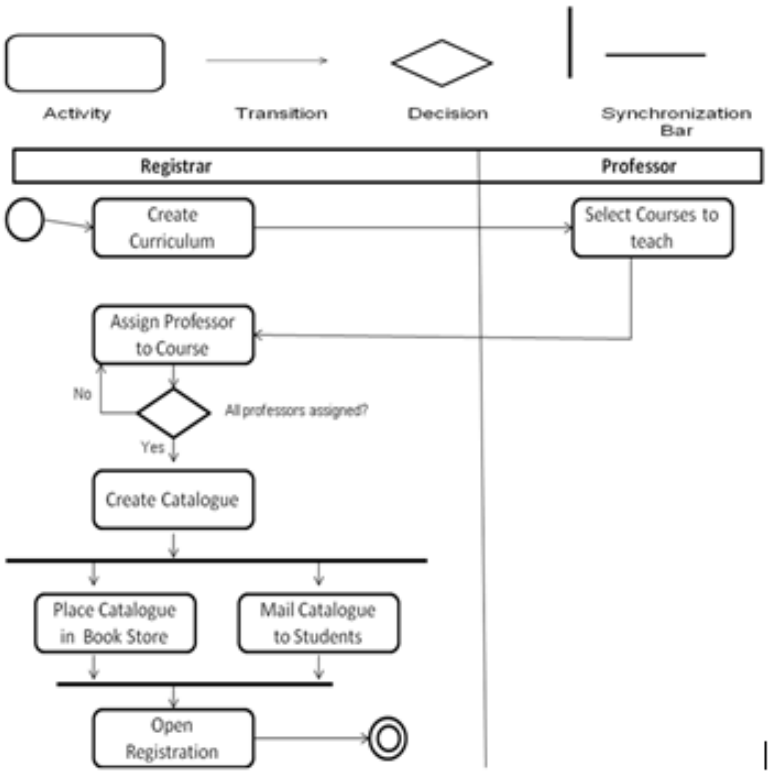
An *activity diagram* shows the flow from **activity to activity** within a system.

An activity shows a **set of activities, the sequential or branching flow** from activity to activity, and objects that act and are acted upon.

Shows what activities can be done in parallel, and any alternate paths through the flow

Activity diagrams contain **activities, transitions between the activities, decision points, and synchronization bars**

Activity diagrams emphasize the **flow of control** among objects.



USE OF SWIMLANES (REFER TO CLASS NOTES)

3. State machine diagram

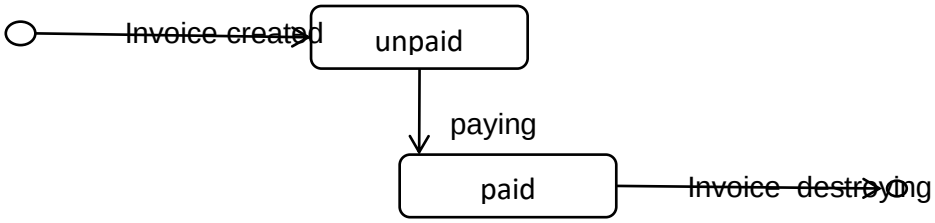
Describes various states of an object that it goes through its lifetime.

It is used for modeling discrete behavior through finite state transitions.

In addition to expressing the behavior of a part of the system, state machines can also be used to express the usage protocol of part of a system.

A *statechart diagram* shows a **state machine**, consisting of states, transitions, events, and activities.

They are especially important in **modeling the behavior** of an interface, class, or collaboration.



Statechart diagrams emphasize the **event-ordered behavior** of an object, which is especially useful in modeling reactive systems.

(Example: Refer to class notes)

INTERACTION DIAGRAMS

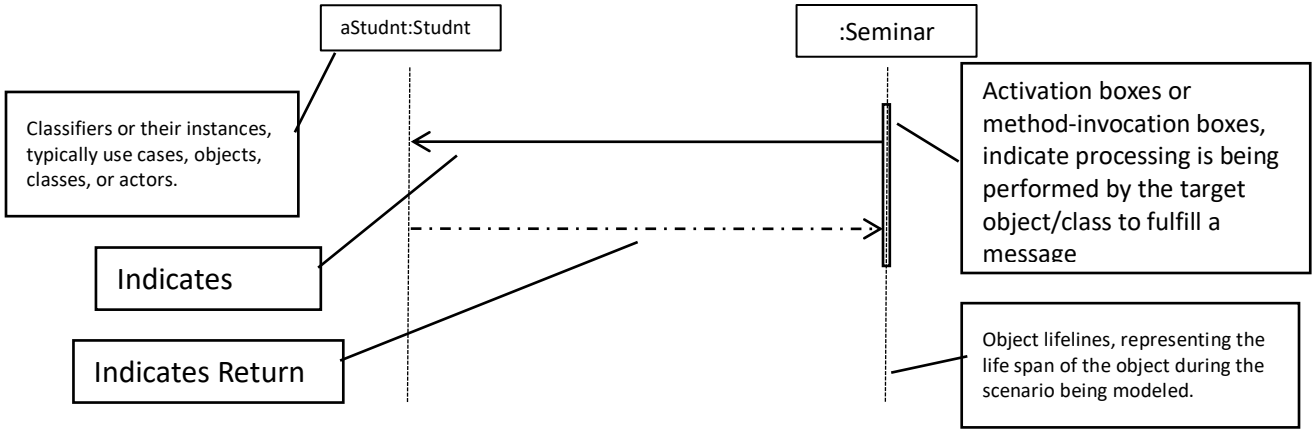
Interaction diagrams include several different types of diagrams:

- sequence diagrams,
- interaction overview diagrams,
- communication diagrams, (known as collaboration diagrams in UML 1.x)
- timing diagrams.

4. Sequence Diagram

A *sequence diagram* is an **interaction diagram** that **emphasizes the time ordering of messages** and shows a set of objects and the messages sent and received by those objects.

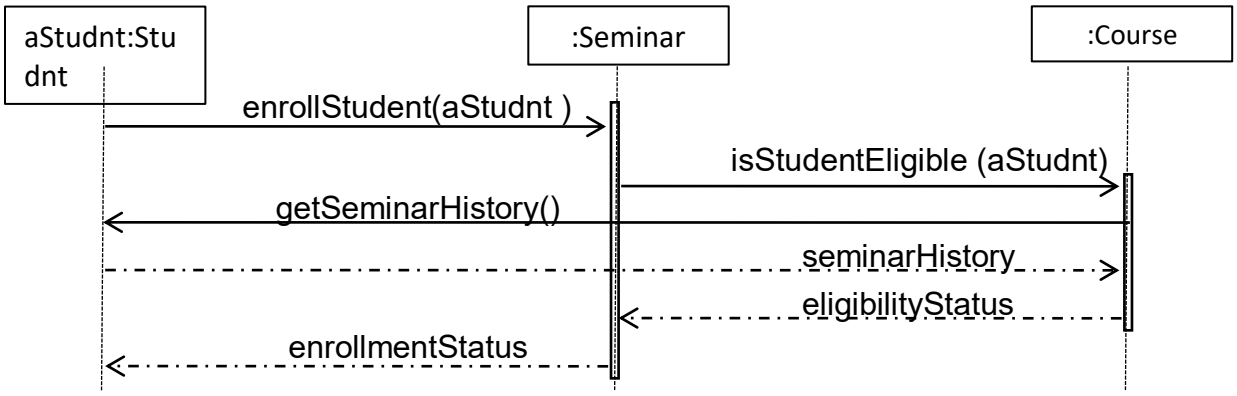
The objects are typically named or anonymous instances of classes, but may also represent instances of other things, such as collaborations, components, and nodes.



A *sequence diagram* is an **interaction diagram** that **emphasizes the time ordering of messages.**

A sequence diagram shows a set of objects and the messages sent and received by those objects.

The objects are typically named or anonymous instances of classes, but may also represent instances of other things, such as collaborations, components, and nodes.



5. Communication diagram (previously known as Collaboration Diagram)

It is a kind of interaction diagram, which focuses on the interaction between lifelines where the architecture of the internal structure and how this corresponds with the message passing is central.

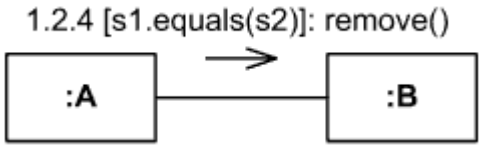
The sequencing of messages is given through a sequence numbering scheme.

It is an **interaction diagram** that emphasizes the structural organization of the objects that send and receive messages.

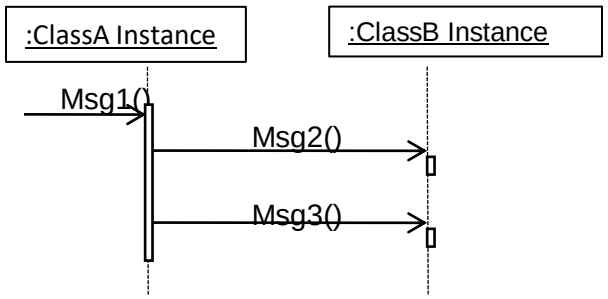
A collaboration diagram **shows a set of objects, links among those objects, and messages** sent and received by those objects.

Message in communication diagram is shown as a line with sequence

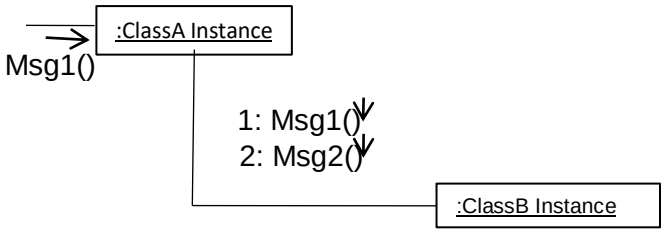
expression and arrow above the line. The arrow indicates direction of the communication.



Sequence and collaboration diagrams are **isomorphic**, meaning that you can convert from one to the other without loss of information.

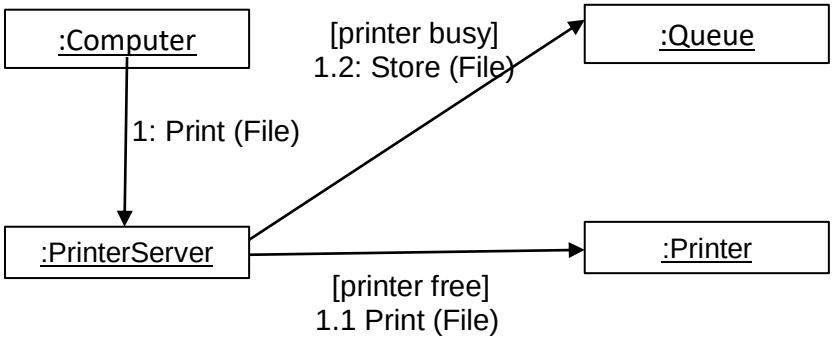


Sequence Diagram



Communication Diagram

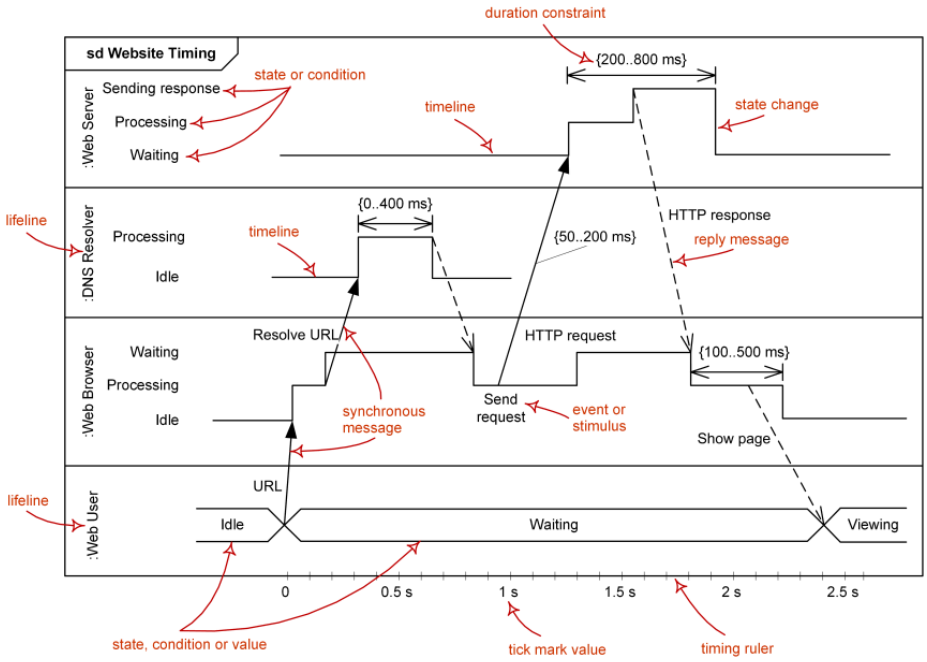
Communication diagram emphasizes on the **organization structure**



6. Timing diagrams

They are used to show interactions when a primary purpose of the diagram is to reason about time and focus on conditions changing within and among Lifelines along a linear time axis.

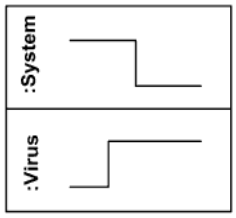
Elements used are lifeline, state or condition timeline, destruction event,duration constraint, time constraint.



Lifeline

Lifeline is a named element which represents an individual participant in the interaction. While parts and structural features may have multiplicity greater than 1, lifelines represent only one interacting entity. See lifeline from sequence diagrams for details.

Lifeline on the timing diagrams is represented by the name of classifier or the instance it represents. It could be placed inside diagram frame or a "swimlane".



Lifelines representing instances of System and Viru

7. Interaction overview diagram

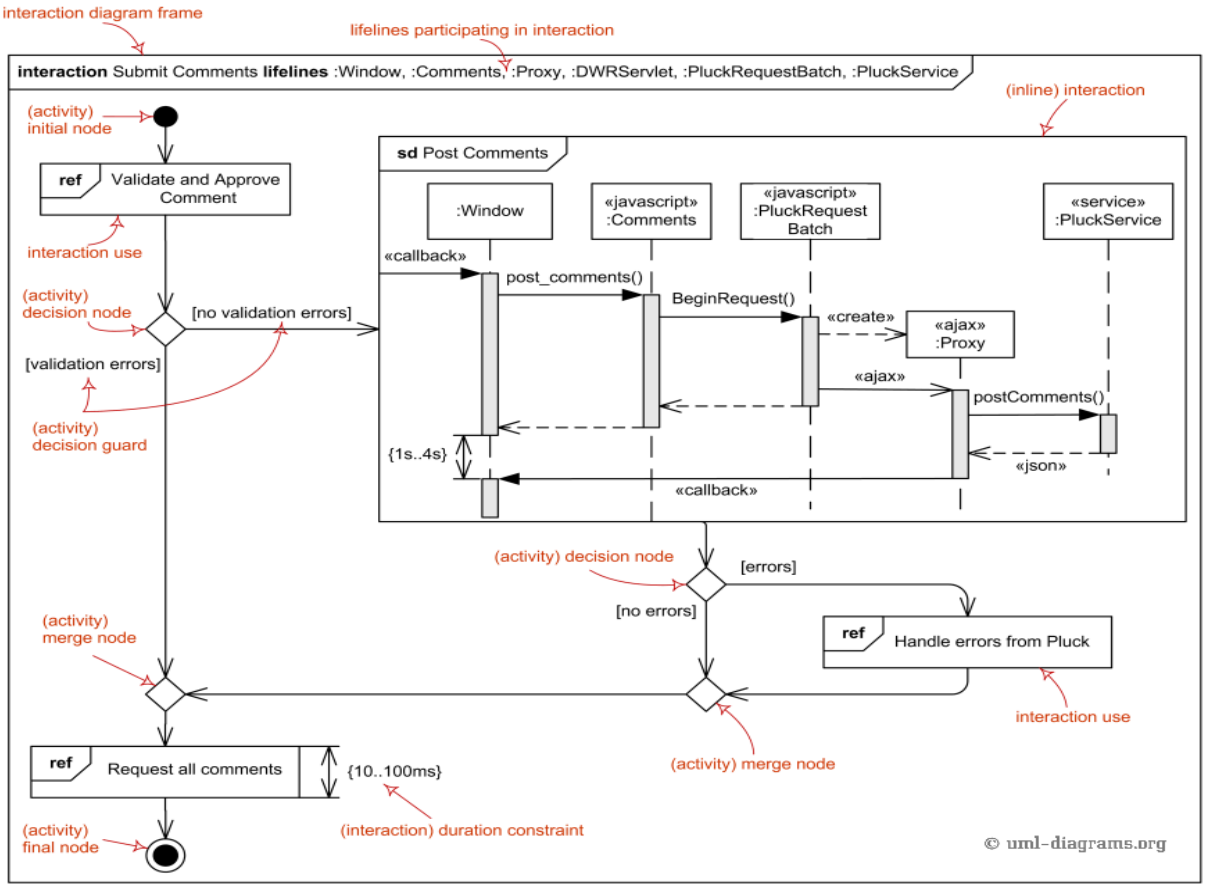
Interaction Overview Diagram=Activity Diagram + Interaction Diagram

It defines interactions through a variant of activity diagrams in a way that promotes overview of the control flow.

Interaction overview diagrams focus on the overview of the flow of control where the nodes are interactions or interaction uses.

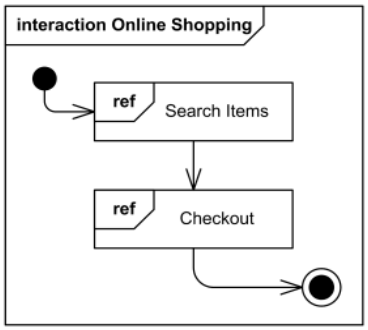
The lifelines and the messages do not appear at this overview level.

UML interaction overview diagram combines elements from activity and interaction diagrams.



Frame

Interaction overview diagrams are framed by the same kind of frame that encloses other forms of interaction diagrams - a rectangular frame around the diagram with a name in a compartment in the upper left corner. Interaction kind is interaction or sd (abbreviated form). Note, that UML has no io or iod abbreviation as some would expect.



Interaction overview diagram Online Shopping

The heading text may also include a list of the contained lifelines (that do not appear graphically).

Elements of Activity Diagram

Interaction overview diagrams are defined as specialization of activity diagrams and as such they inherit number of graphical elements.

Interaction overview diagrams can only have inline interactions or interaction uses instead of actions, and activity diagram actions could not be used.

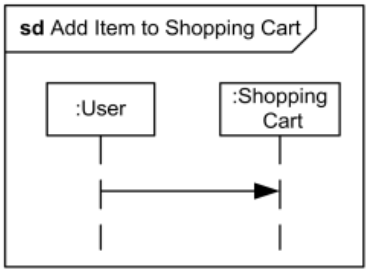
The elements of the activity diagrams used on interaction overview diagrams are initial node, flow final node, activity final node, decision node, merge node, fork node, join node

Elements of Interaction Diagram

The elements of the interaction diagrams used on interaction overview diagrams are interaction, interaction use, duration constraint, time constraint

Interaction

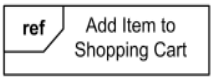
An interaction diagram of any kind may appear inline as an invocation action. The inline interaction diagrams may be either anonymous or named.



Interaction Add Item to Shopping Cart may appear inline on some interaction overview diagram

Interaction Use

An interaction use may appear as an invocation action.



Interaction use Add Item to Shopping Cart may appear on some interaction overview diagram

CHAPTER 2

OBJECT ORIENTED ANALYSIS

CHAPTER 2

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?"

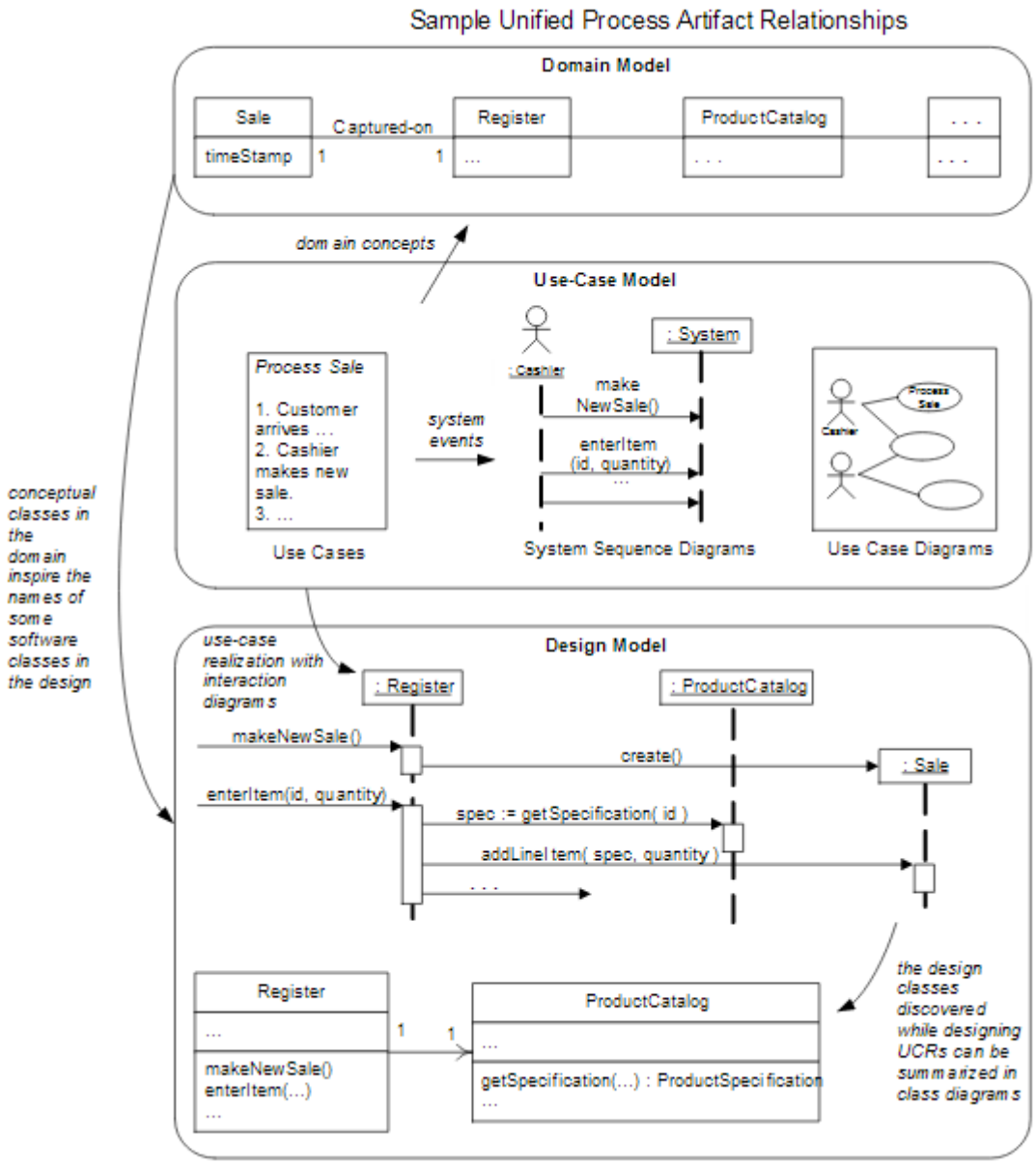
Design emphasizes on a **logical solution**, how the system fulfill the requirements.

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.

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.

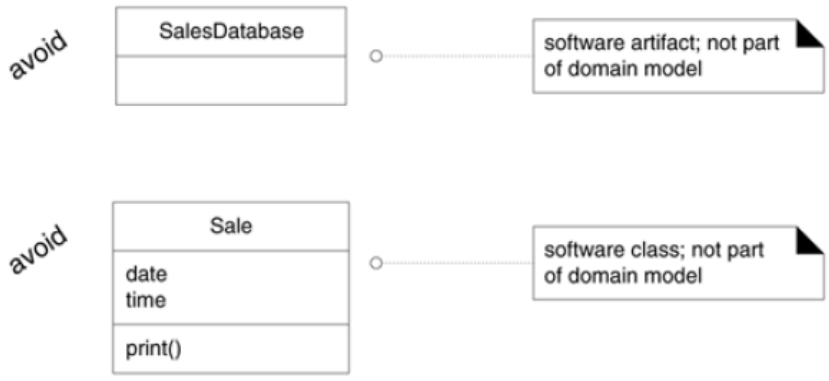
Using UML notation, a domain model is illustrated with a set of **class diagrams** in which **no operations** are defined.

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



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.

WHY CREATE A DOMAIN MODEL?

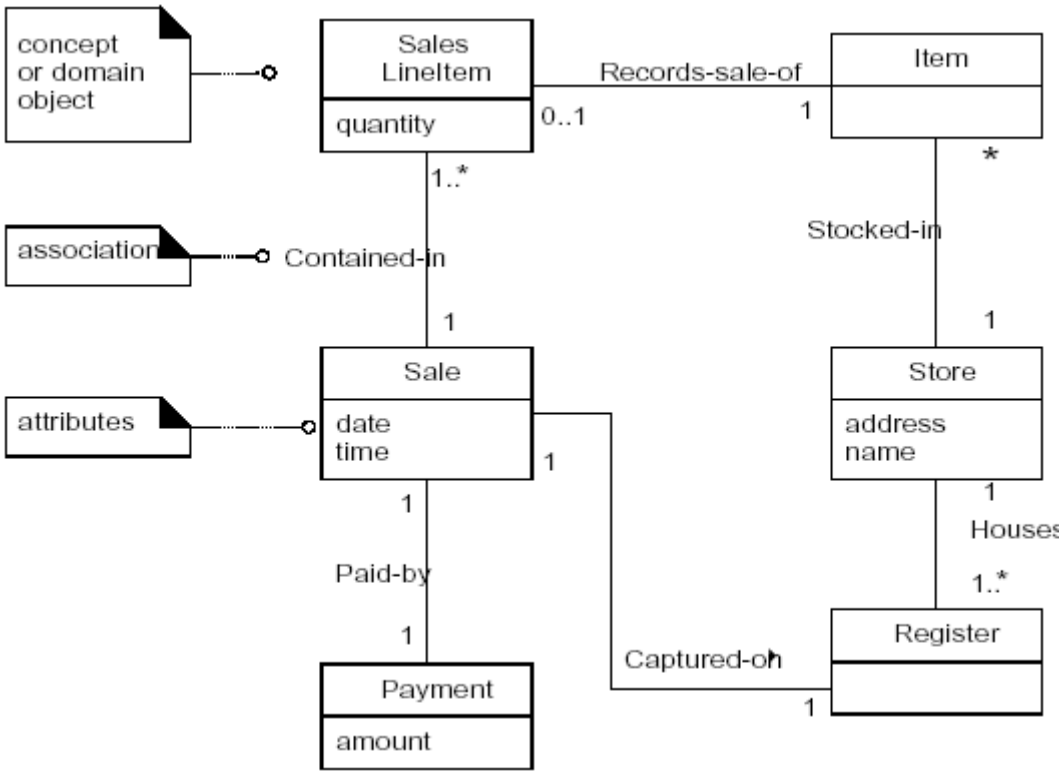
- Domain models are created **to get a better understanding of the key concepts and vocabulary used in a particular domain**. If it is required to develop a software for an aircraft manufacturing company, one should do proper domain modeling to understands the key concepts and jargons or technology specific 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.

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.

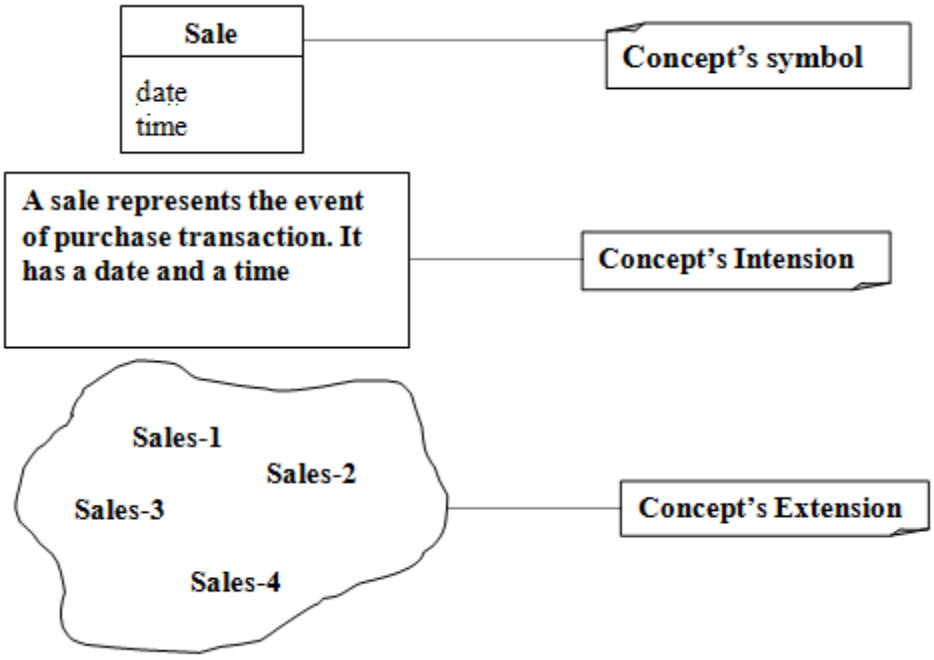
• Extension—the set of examples to which the conceptual class applies.

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.

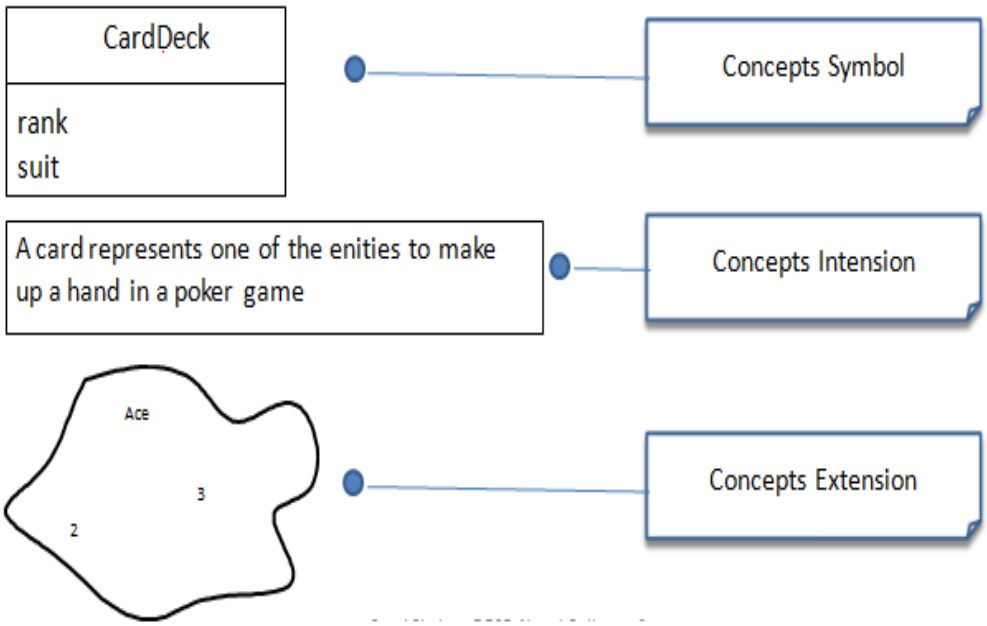
Extension: All the examples of sales; in other words, the set of all sales.



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 the parties involved in a transaction	Cashier, customer, airline, passenger, Pilot
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	POST, die, Airplane, board, item

simulations	
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

Item

Store

Sale

Payment

ProductCatalog

ProductDescription

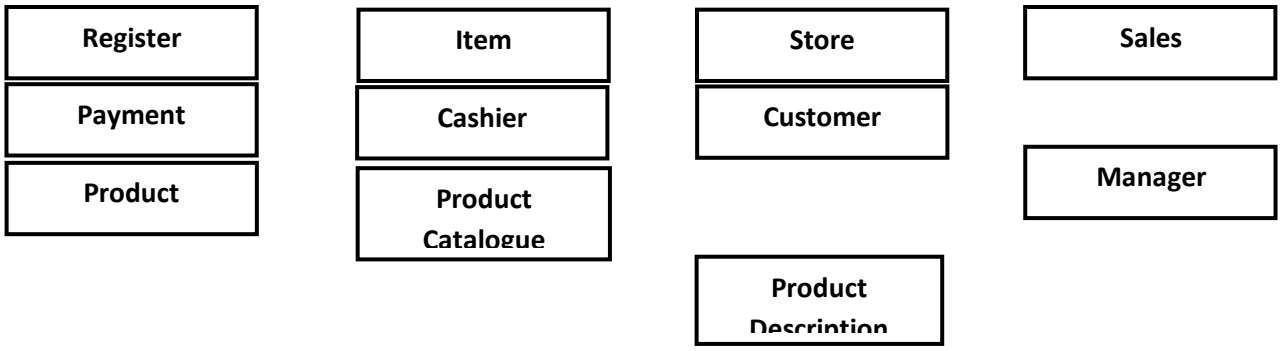
SalesLineItem

Cashier

Customer

Manager

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

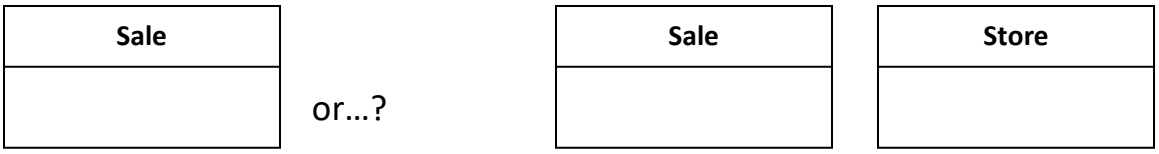
A Common Mistake in Identifying Conceptual Classes

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 :

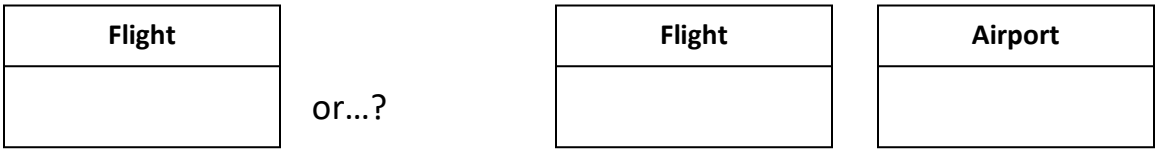
If we do not think of some conceptual class X as a number or text in the real world, X is probably a conceptual class, not an attribute.

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



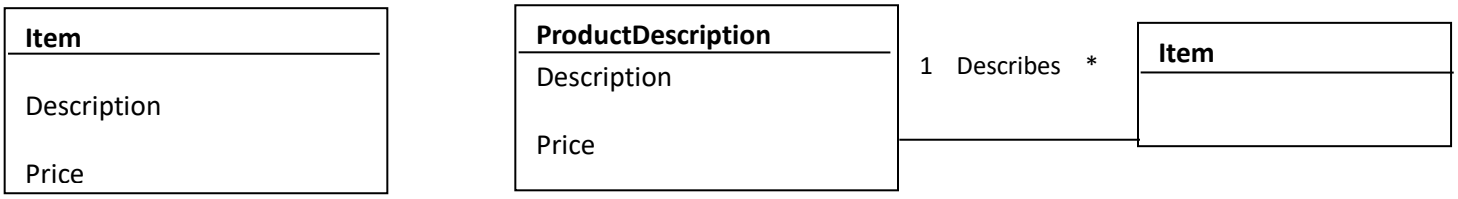
In the real world, a store is not considered a number or text—the term suggests a legal entity, an organization, and something occupies space. Therefore, Store should be a concept.

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

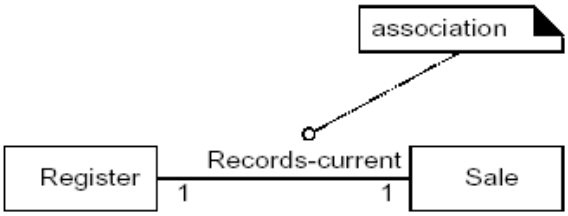


Description or specification objects are strongly related to the things they describe. In a domain model, it is common to state that an XDescription Describes an X

ii. Domain Modeling : Adding Associations

Associations

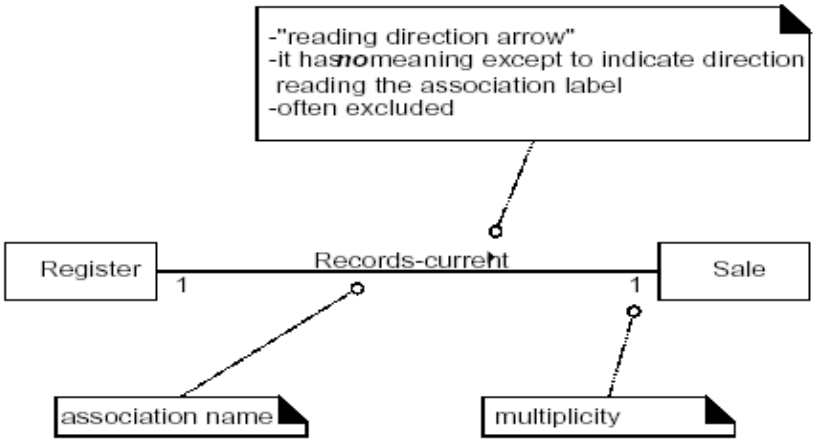
It is **a relationship between concepts** that indicates some meaningful interesting connection.



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.



Finding Associations- Common Association List:

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

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 a physically or logically contained in B

A is recorded 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

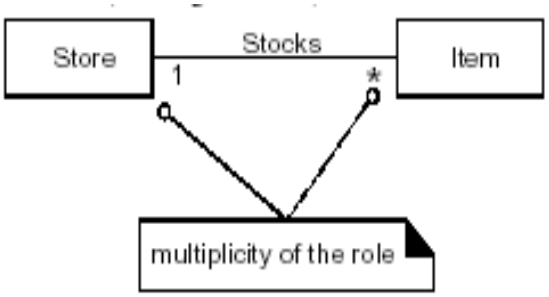
- multiplicity expression

- navigability

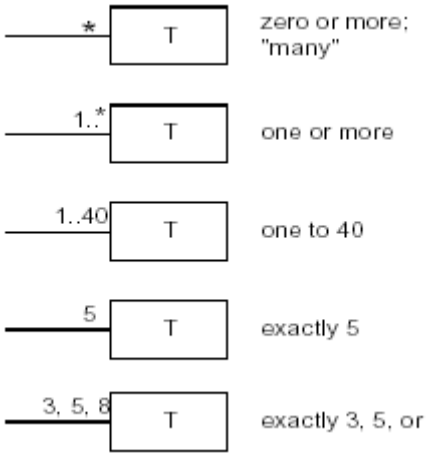
i. Multiplicity

- 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.

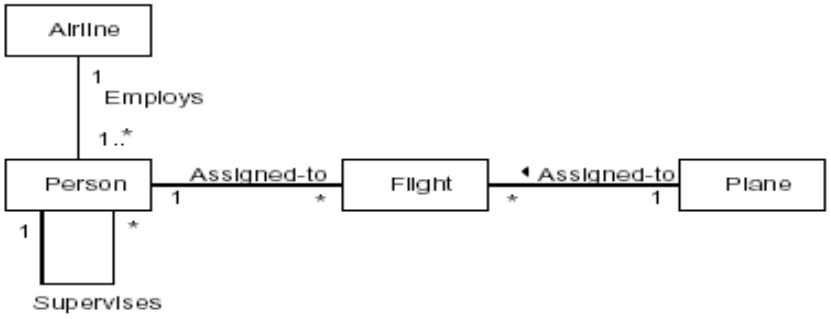
- Association names should **start with a capital letter**, since an association represents a classifier of links between instances; in the UML, classifiers should start with a capital letter.

  Good: Sale PaidBy CashPayment Bad: Sale uses CashPayment

Good: Player IsOn Square Bad: Player has Square

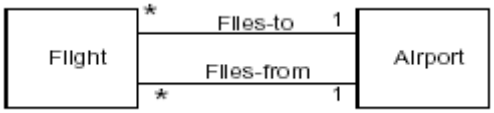
Two common and equally legal formats for a compound association name are:

- *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

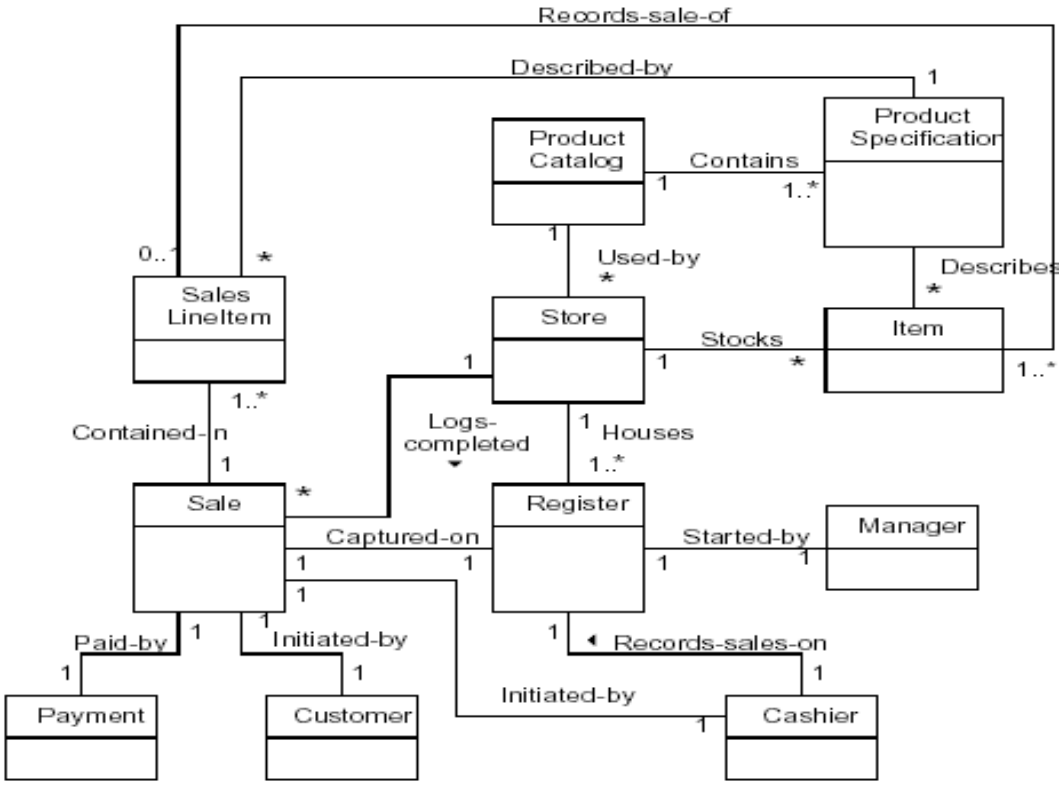
Based on previously identified types and considering the current use case requirements.

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}
```

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.

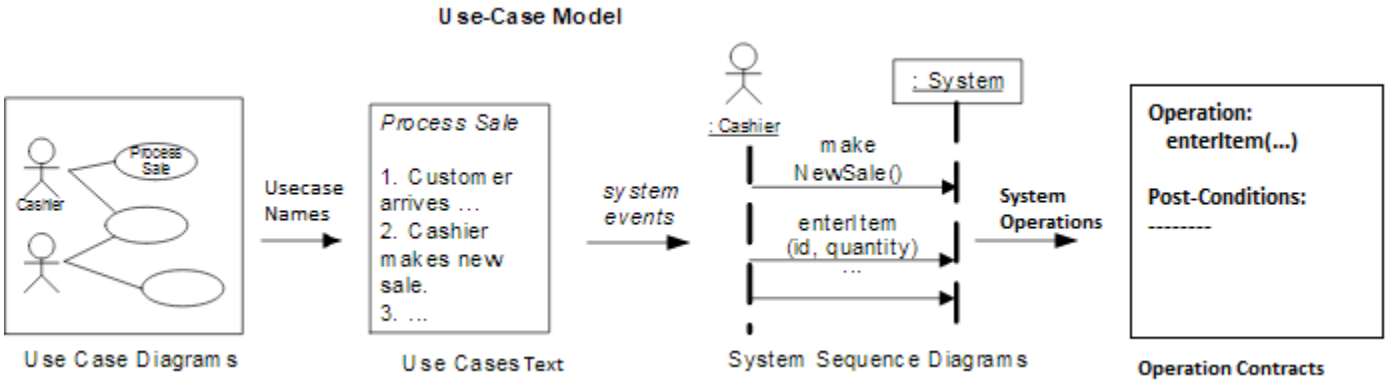
It is desirable to **isolate and illustrate the operations that an external actor requests** of a system, because they are an important part of understanding system behavior.

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.



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 f or 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.

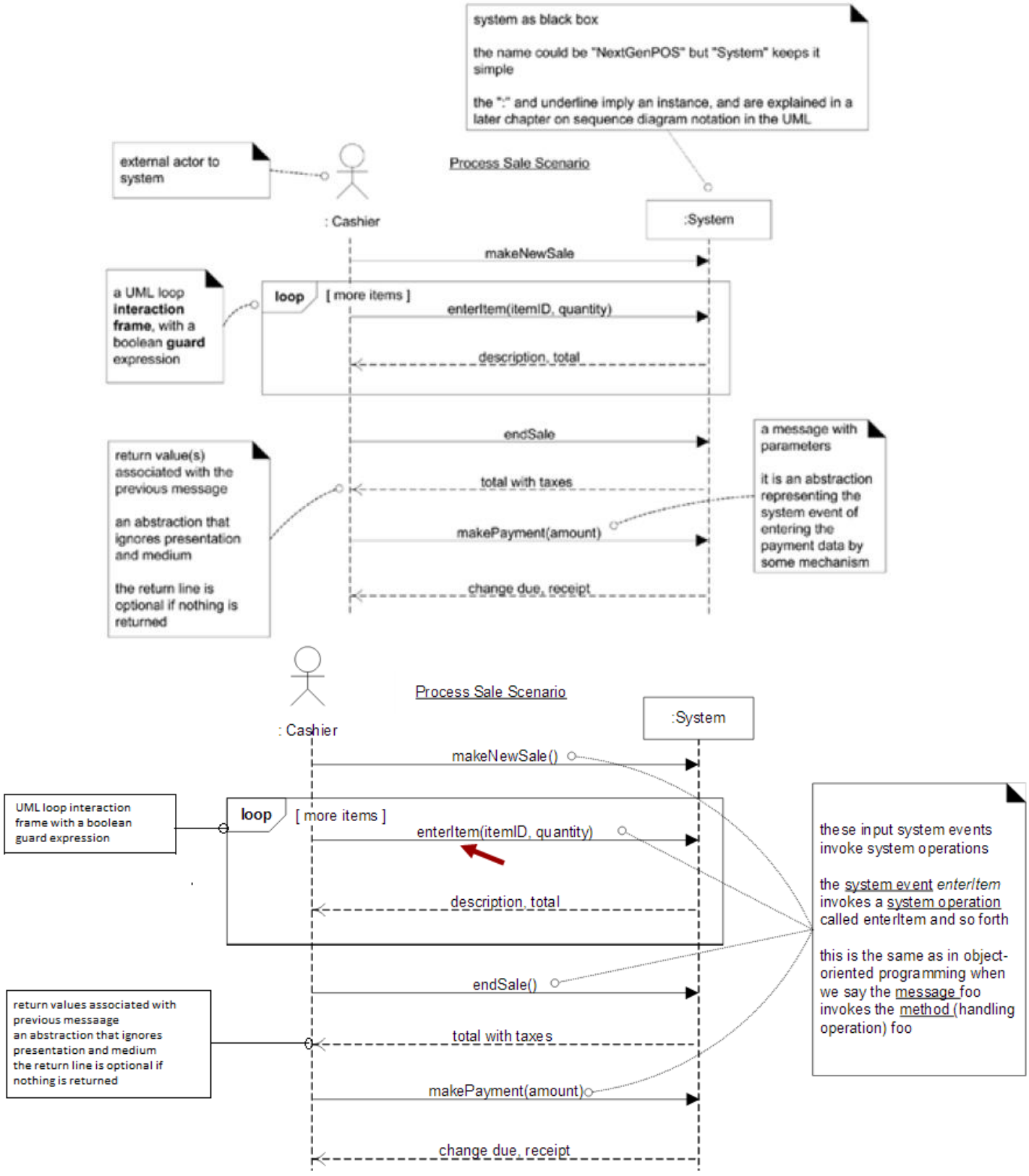
Example of a sequence diagram An SSD shows, for a particular course of events within a use case, the external actors that interact directly with the system, the system (as a black box), and the system events that the actors generate .

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.**



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

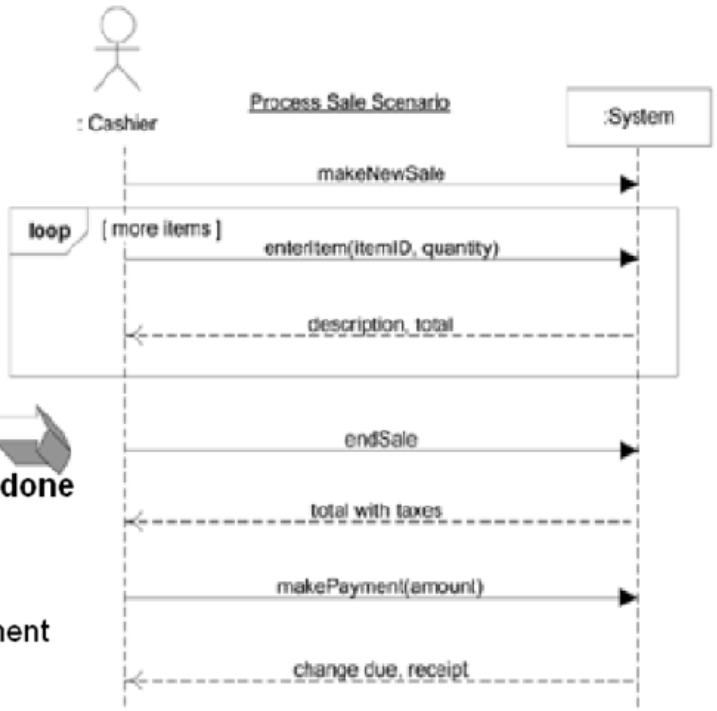
System sequence diagram is a part of that description

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

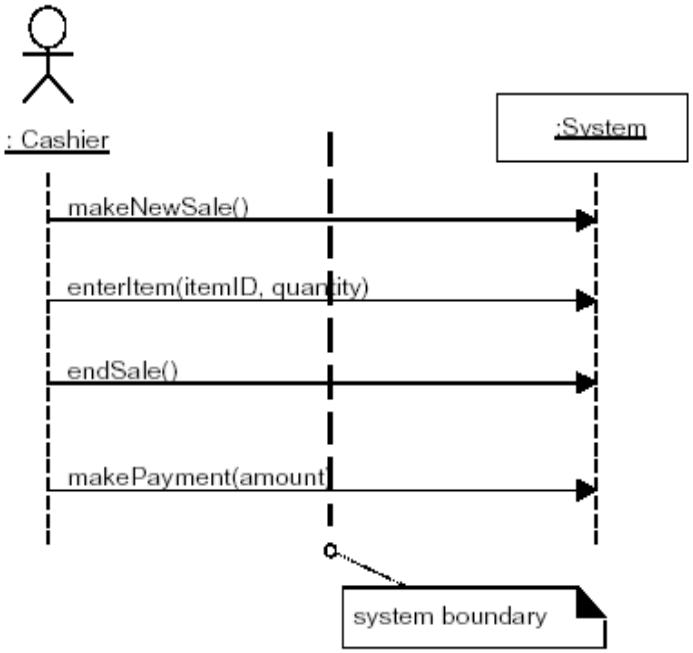


Like in case of an ATM system the use case scenarios `make_withdrawal`, `make_balance_enquiry`, `maintain_ATM` etc can be elaborated further by individual system sequence diagrams to identifying the system events pertaining to the individual scenario.

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

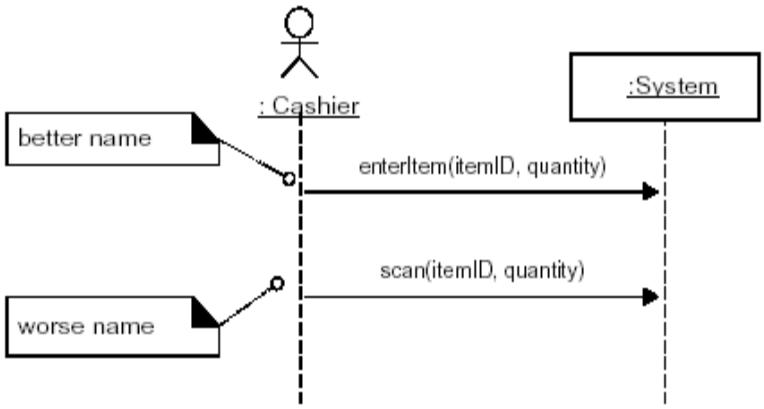


Consider the *Process Sale* use case to identify system events. First, we must determine the actors that directly interact with the software system. The customer interacts with the cashier, but for this simple cash-only scenario, does not directly interact with the POS system—only the cashier does. Therefore,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

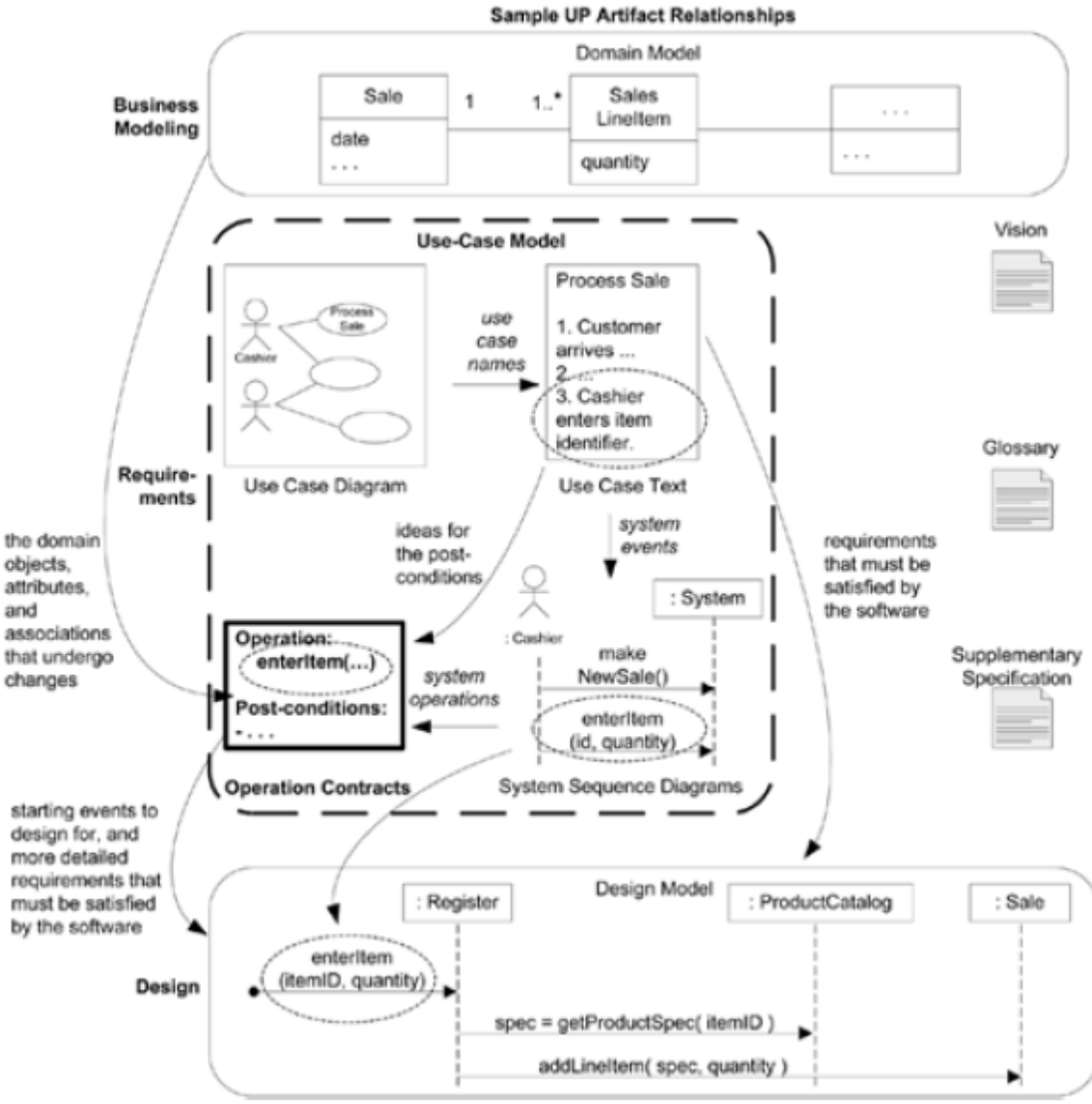
When the Cashier generates the enterItem system event it causes the execution of the enterItem system operation (in this case the event is the named stimulus and the operation is the response)

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)`

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.

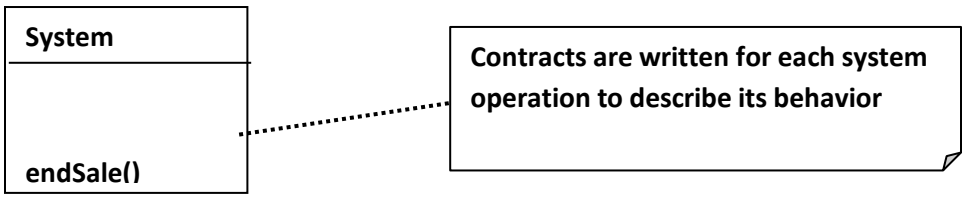
Contracts **describe detailed system behavior in terms of state changes** to objects in the Domain Model, after a system operation has executed.

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



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.

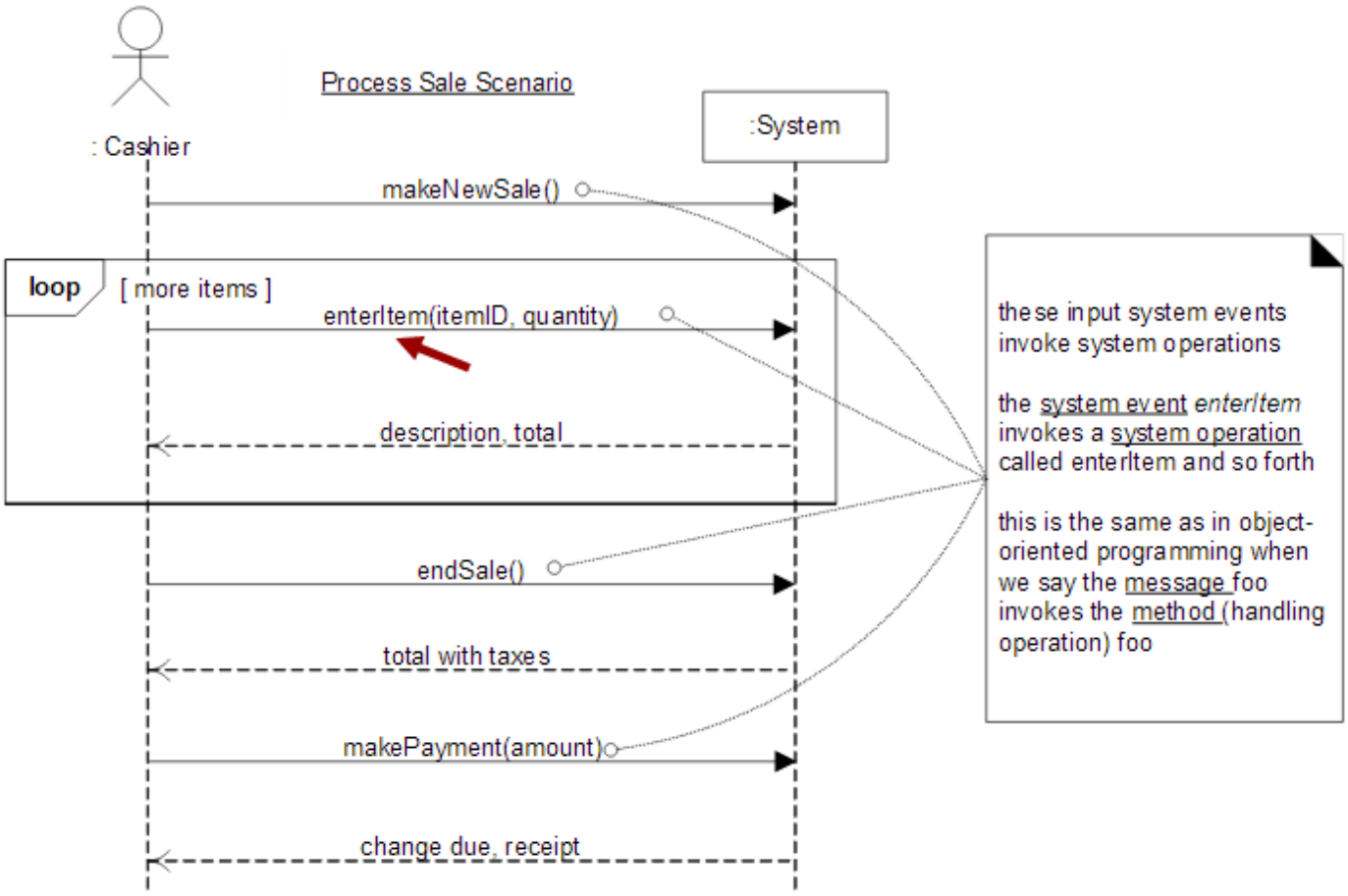
System operations can be identified by discovering these system events

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



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 References:	enterItem(itemID : ItemID, quantity : integer) Use Cases: Process Sale There is a sale underway.
Preconditions:	
Postconditions:	- A SalesLineItem instance sli was created (instance cre ation). - 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 *SalesLineItem 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 *SalesLineItem* was created. .

(worse) Create a *SalesLineItem.*

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 *SalesLineItem* instance is created when the *enterItem* operation occurs. After the operation is complete, it should also be true that the newly created instance was associated with *Sale;*

thus:

The *SalesLineItem* 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 *SalesLineItem.* Thus:

. A *SalesLineItem* 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 *SalesLineItem* 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 *SalesLineItem* 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 *SalesLineItem'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)

CHAPTER 3

OBJECT ORIENTED DESIGN

CHAPTER 3

OBJECT ORIENTED DESIGN

THE OODM PHASES

The **Analysis phase** ends with finding out

- the **Essential use cases** that identifies the domain processes
- the conceptual or **domain model** that identifies the concept classes or terms existing in the domain
- the **system sequence diagram** that shows the system events and operations and contracts that shows what the system operations do.

Design Phase

OOA: Investigation of the **requirements, concepts,** and **operations** related to a system.

OOD: Designing a solution for this iteration **in terms of collaborating software objects.**

During object design, **a logical solution,** based on the object-oriented paradigm, is developed. The heart of this solution is the **creation of interaction diagrams,** which illustrate how objects collaborate to fulfill the requirements.

After or in parallel with drawing interaction diagrams, (design) **class diagrams** can be drawn. These summarize the definition of the software classes (and interfaces) that are to be implemented in software.

In terms of the UP, these artifacts are part of the **Design Model.**

In practice, the creation of interaction and class diagrams happens in parallel and synergistically, but their introduction is linear in this case study, for simplicity and clarity.

The term **interaction diagram** is a generalization of specialized UML diagrams used to express message interactions like:

- Communication diagrams
- Sequence diagrams

A. INTERACTION DIAGRAMS

The creation of an interaction diagram occurs within the design phase of the development cycle.

The creation of interaction diagram is dependent on **Conceptual Model** that helps in defining software classes corresponding to concepts.

Objects of these classes participate in interactions illustrated in the interaction diagrams

System operation and contract

This helps the designer **to identify the responsibilities and the post conditions** that the interaction diagram must fulfill. An Interaction diagram is used for **dynamic object modeling** and **illustrates object interaction via messages** for fulfilling goals.

The starting point of these interactions is the **fulfillment of the post conditions** of the operation contracts

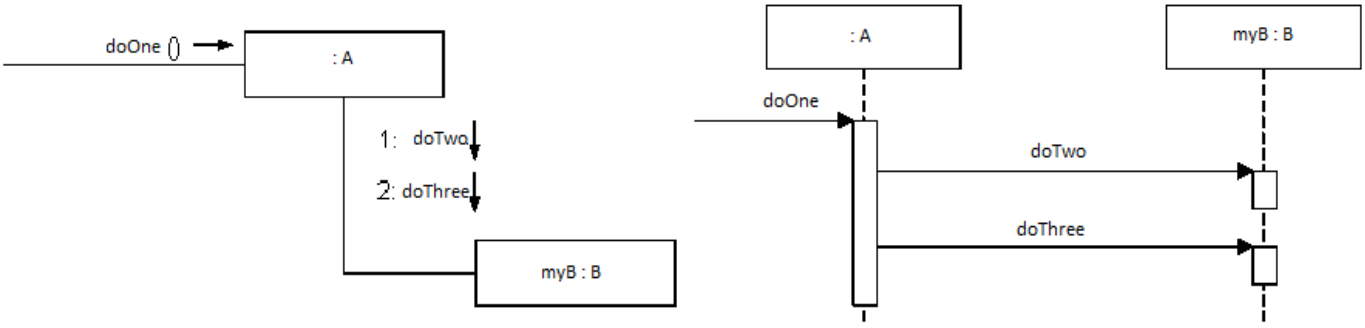
The two common types of interaction diagrams in the UML are

- Communication diagrams
- Sequence diagrams

A related diagram is the **interaction overview diagram** that provides a big picture **overview of how a set of interaction diagrams are related** in terms of logic and process flow

Communication diagrams illustrate the object interaction in a graph or network format, in which **objects can be placed anywhere** on the diagram

Sequence Diagrams illustrate interaction in a kind of fence format, in which **each new object is added to the right**



Related Code:

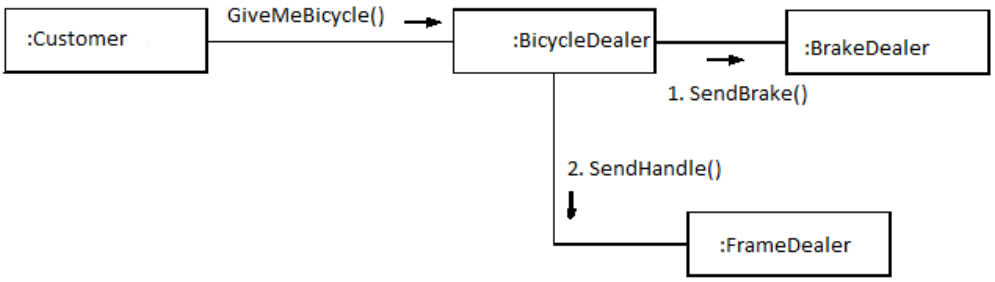
This depiction of the message passing in the figures above can be translated in logic as :

Class A has one method named doOne and one instance of class B called myB. Class B has methods named doTwo and doThree. Whenever an object sends a message to an instance of class A asking it to doOne, it sends a message to an instance of class B (myB) to doTwo and doThree. It means that an instance of class A needs to seek collaboration from an instance of class B to fulfill its responsibility of doing the task doOne.

```
Public class A{
      Private B myB = new B();
      Public void doOne(){
            myB.doTwo();
            myB.doTwo();
      }
```

```
}
```

For example, a customer asks a bicycle store owner for a bicycle, the bicycle dealer asks various parts dealers like brake dealer, handle dealer, tyre dealer etc and then assembles the parts and hands them over to the customer , here the customer does not need to know about the details regarding how the dealer gets the parts.




```
Public class BicycleDealer{
      Private BrakeDealer myBreakDealer = new BreakDealer();
      Private FrameDealer myFrameDealer = new FrameDealer();
            ……
      Public void assembleBicycle(){
            myBreakDealer.sendBrake();
            myFrameDealer.sendFrame();
            ……
            screwThem all;
      }
}
```

Difference between Sequence and Communication Diagram

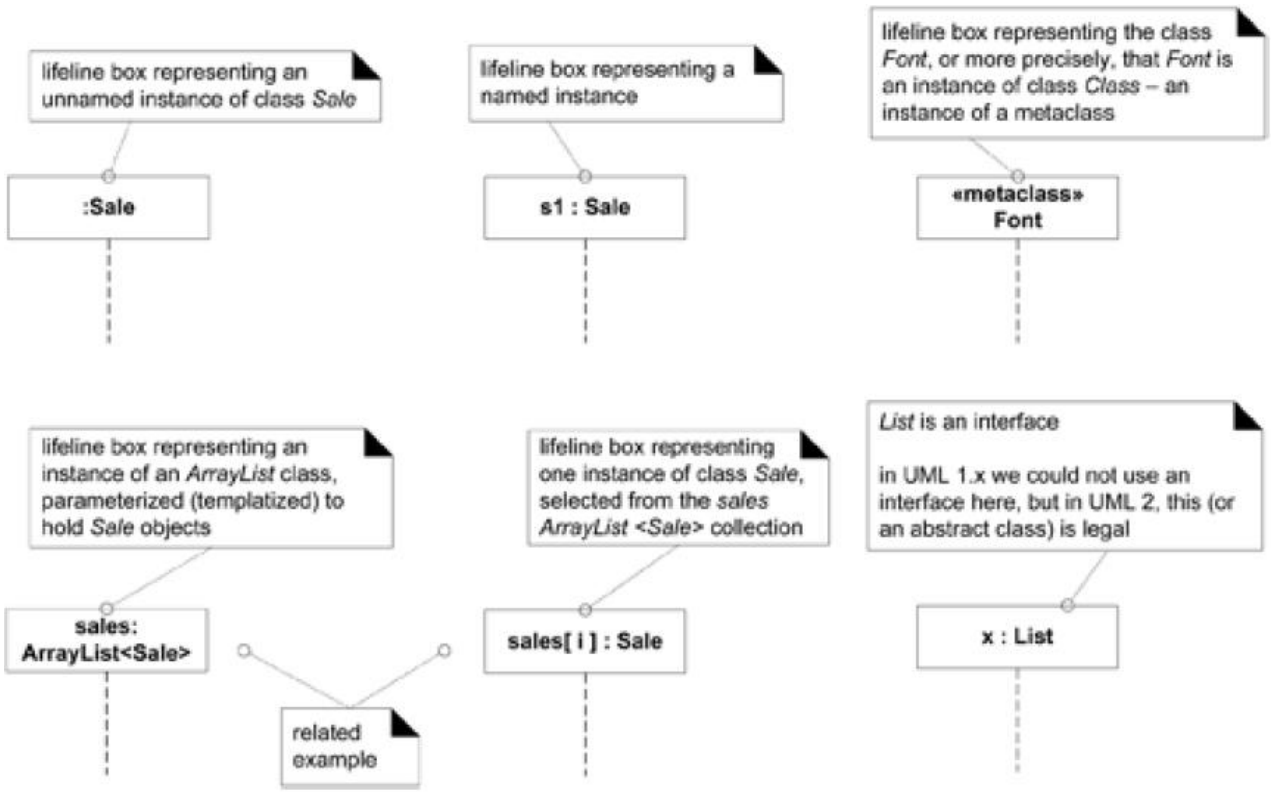
Type	Strength	Weaknesses
Sequence	Clearly shows sequence or time ordering of messages (call flow sequence) Simple but large set of detailed notation options Excellent for documentation	Forced to extend to the right when adding new objects: consumes horizontal space
Communication	Space economical-flexibility to add New objects in two dimensions Better to illustrate complex branching, Iteration, and concurrent behavior	Difficult to see message sequence denoted by numbers 1: 2: More complex notation

BASIC COMMON UML INTERACTION DIAGRAM NOTATIONS

- **Illustrating Participants and Lifeline Boxes**

The **boxes** in the interaction diagrams **denote lifeline boxes,** they represent the participants in the interaction

The participants can be interpreted as a **representation of an instance of a class**



Basic Message Expression Syntax

The UML has a standard syntax for message expressions:

```
return := message(parameter : parameterType) : returnType
```

Type information may be excluded if obvious or unimportant. For example:

```
faceValue := roll() : integer

spec := getProductSpect(id)

spec := getProductSpect(id:ItemID)

spec := getProductSpect(id:ItemID) : ProductDescription
```

COMMUNICATION DIAGRAM NOTATION

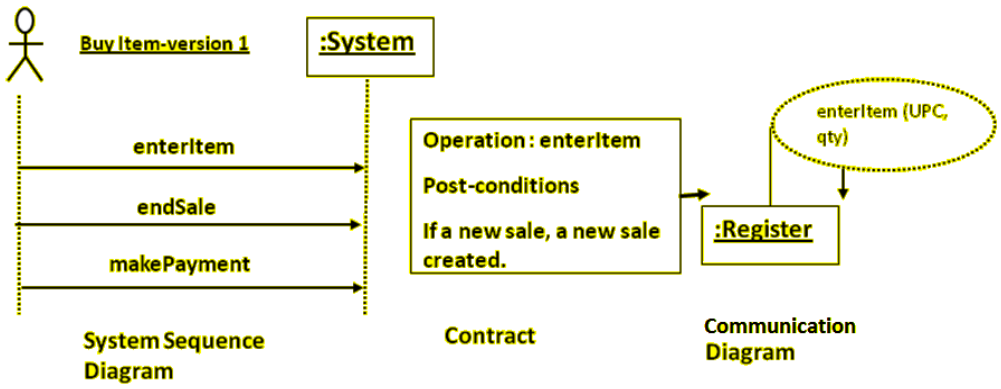
Making Communication Diagrams

Create a separate **diagram for each system operation** under development under the current development cycle.

For **each system operation message**, make a diagram with it as the starting message

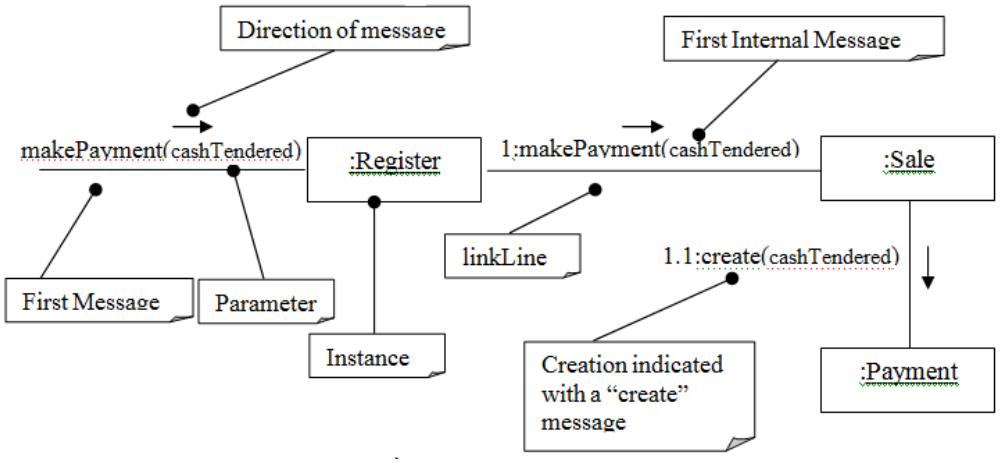
If the diagram gets complex **split it** into smaller diagrams

Using the **operation contract responsibilities and post conditions and the use case description** as a starting point, design a system of interacting objects to fulfill the task

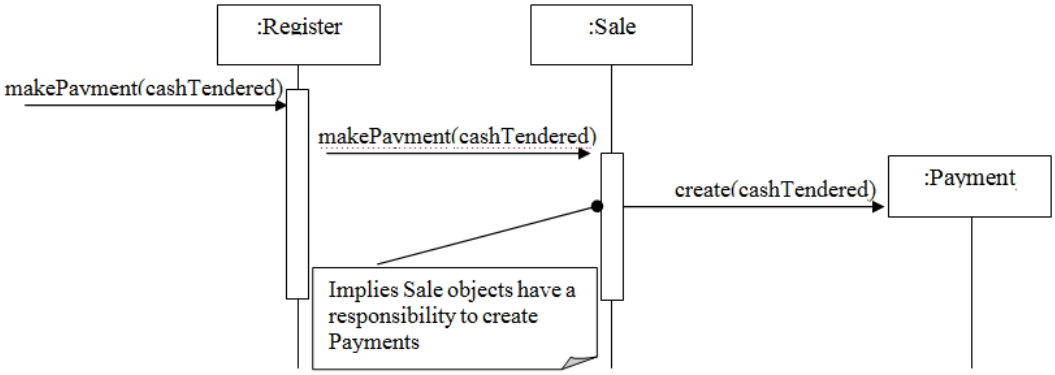


Example Communication Diagram: makePayment

1. The message `makePayment` *is* sent to an instance of a `Register`. The sender is not identified.
2. The `Register` instance sends the `makePayment` message to a `Sale` instance.
3. The `Sale` instance creates an instance of a `Payment`



Example Sequence Diagram : makePayment



Related Code:

```
Public class Sale{
      Private Payment payment;
      Public void makePayment(Money cashTendered){
            payment = new Payment(cashTendered);
            --------
      }
}
```

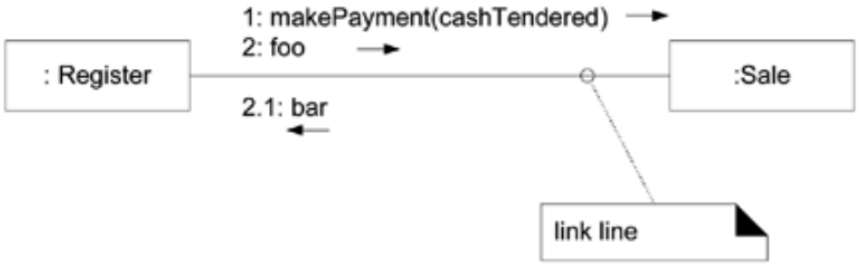
Links

A link is a **connection path between two objects**; it indicates some form of navigation and visibility between the objects is possible

More formally, a link is an **instance of an association**.

For example, there is a link or path of navigation from a `Register` to a `Sale`, along which messages may flow, such as the `makePayment` message.

Note that multiple messages, and messages both ways, can flow along the same single link.

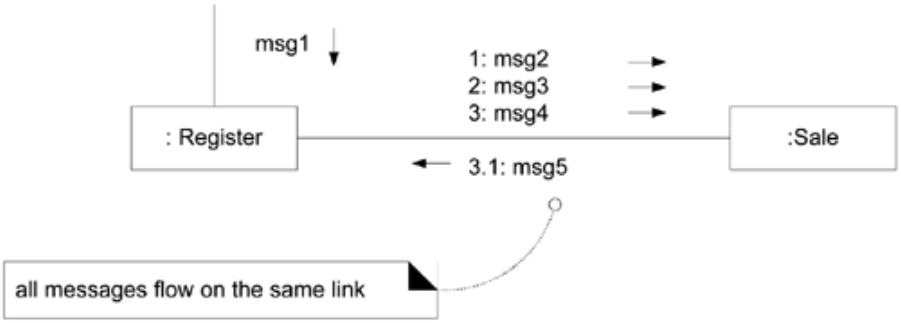


Messages

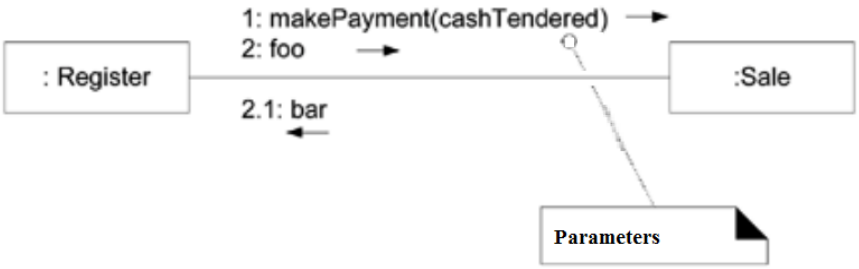
Each message between objects is represented with a **message expression** and **small arrow** indicating the direction of the message.

Many messages may flow along this link .

A sequence number is added to show the sequential order of messages in the current thread of control.



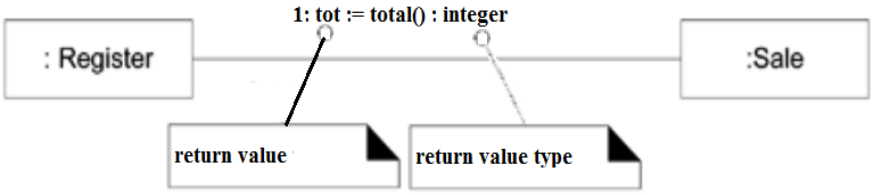
Illustrating Parameters



Parameters of a message may be shown within parentheses following the message name. The type of parameter may optionally be shown.

Illustrating a return type

A return value may be shown by preceding the message with a return value variable name and an assignment operator (:=). The type of the return value may be optionally shown.



Illustrating the message to "self" or "this"

A message can be sent from an object to itself

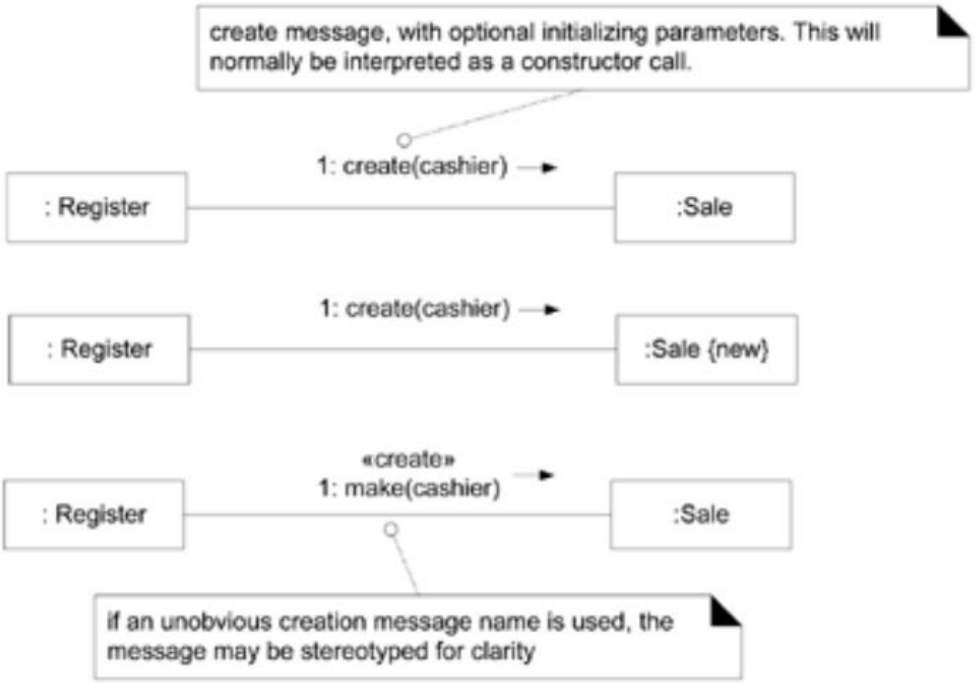
Illustration of creation of instances

A convention in the UML to use a message named **create** for this purpose.

If another (perhaps less obvious) message name is used, the message may be annotated with a special feature called a UML stereotype, like so: `«create»`.

The `create` message **may include parameters,** indicating the passing of initial values. e.g. a constructor call with parameters in Java.

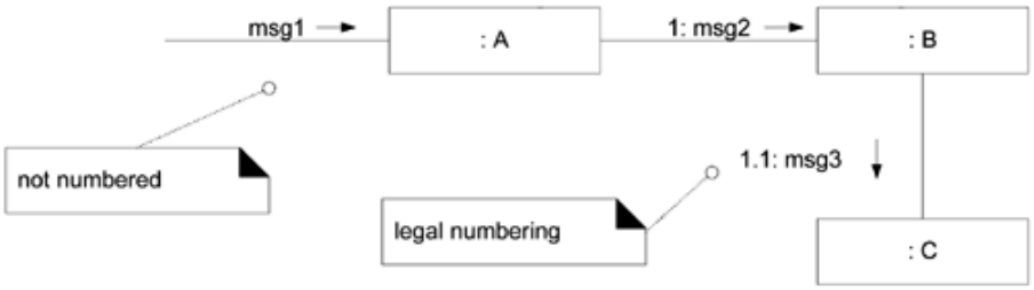
The UML tagged value ***{new}*** may optionally be added to the lifeline box to highlight the creation *tagged values are extension in the UML for adding semantically meaningful information to the UML element*



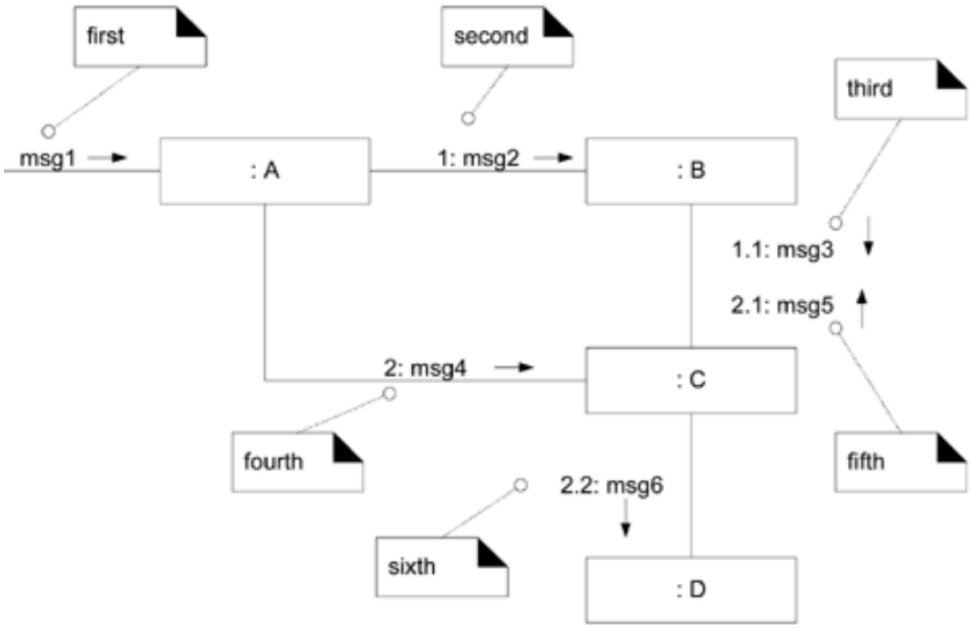
Illustrating Message Number Sequencing

The order of message is illustrated with the sequence numbers

1. The first message is not numbered
2. The order and nesting of subsequent messages is shown with a legal numbering scheme in which nested messages have appended to them a number.



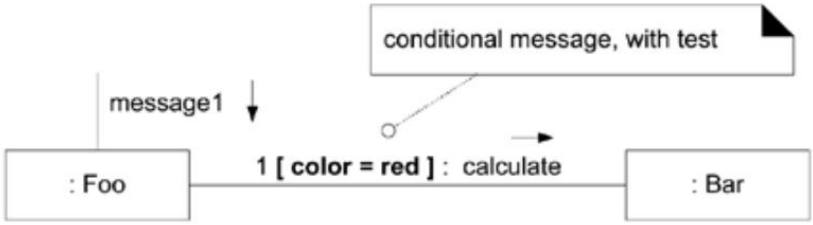
Nesting is denoted by prepending the incoming message number to the outgoing message number



Illustrating Conditional Message

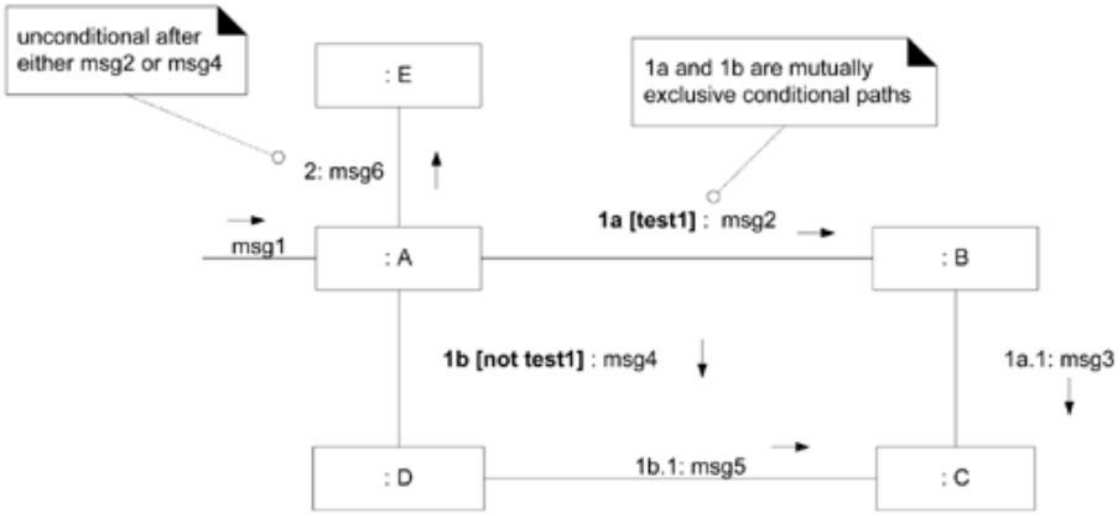
A conditional message is shown by f**ollowing a sequence number with a conditional clause in square brackets**, similar to an iteration clause.

The message is only **sent if the clause evaluates to *true.***



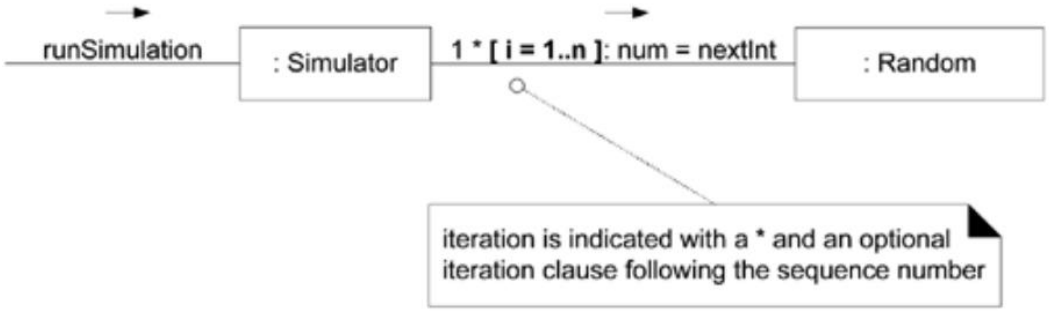
Illustrating Mutually Exclusive Conditional Paths

The example illustrates the sequence numbers with mutually exclusive conditional paths.



Here either 1a or 1b could execute after msg1

Illustrating iteration



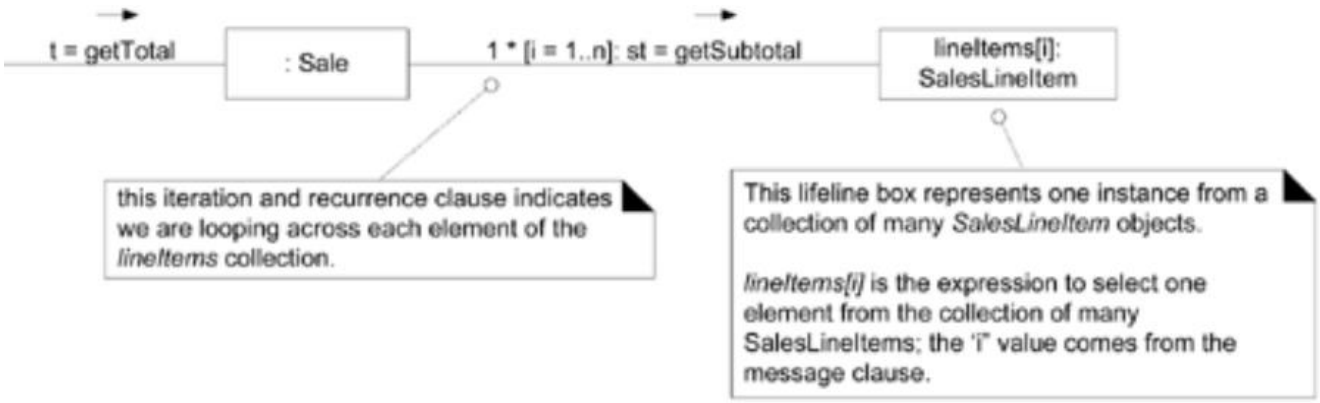
Iteration is indicated by following the sequence number with a star (‘*’)

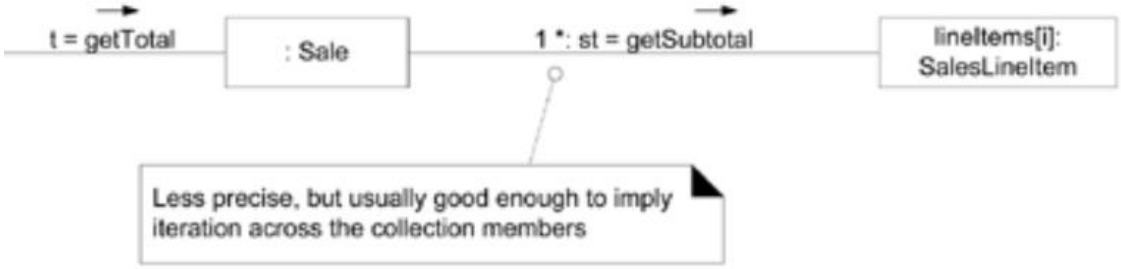
This expresses that the message is being sent repeatedly, in a loop to the receiver

To express more than one message happening within the same iteration clause, repeat the iteration clause on each message

Illustrating Collections

A common algorithm is to **iterate over all members of a collection** (such as a list or map), sending a message to each.





Often, some kind of iterator object is ultimately used, such as an implementation of *java.util.Iterator* or a C++ standard library iterator.

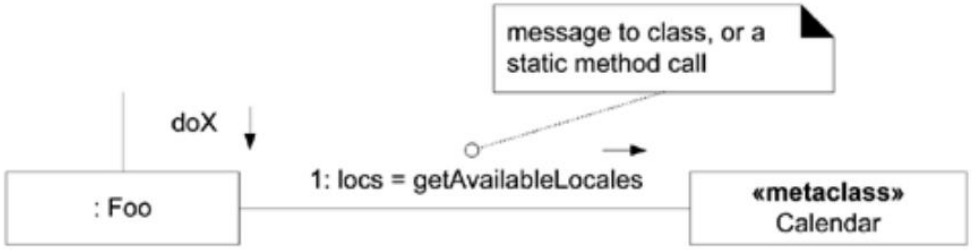
In the UML, the term **multi object** is used to denote a set of instances .a collection.

The **"*"** multiplicity marker at the end of the link is used to indicate that the message is being sent to each element of the collection, rather than being repeatedly sent to the collection object itself.

Illustrating message to a Class Object

Messages may be **sent to a class itself**, rather than an instance, to invoke class or **static methods.**

A message is shown to a class box whose name is not underlined, indicating the message is being sent to a class rather than an instance.



Message may be sent to a class itself, rather than an instance, in order to invoke class methods.

SEQUENCE DIAGRAM NOTATION

Lifeline Boxes and Lifelines

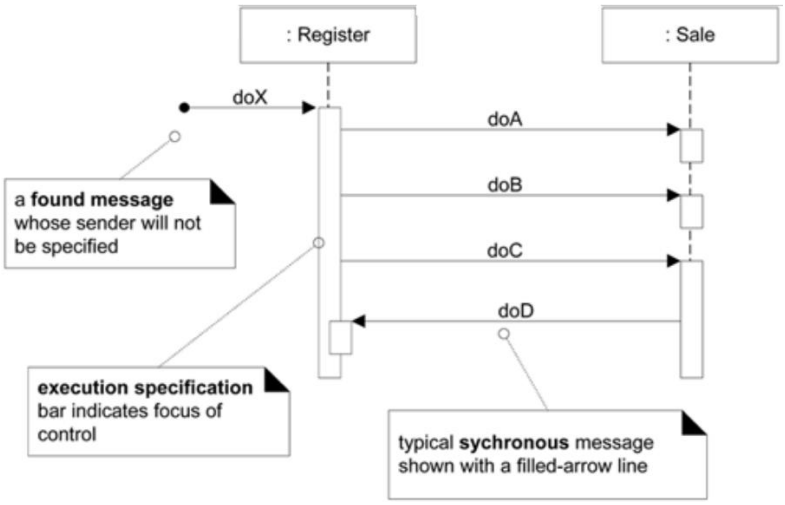
In contrast to communication diagrams, in sequence diagrams the lifeline boxes include a vertical line extending below them, these are the actual lifelines.

Although virtually all UML examples show the lifeline as dashed (UML 1 influence), in fact the UML 2 specification says it may be solid or dashed.

Messages

Each (typical synchronous) message between objects is represented with a **message expression on a filled-arrowed solid line** between the vertical lifelines

The time ordering is organized **from top to bottom** of lifelines.



Here, the starting message is called a **found message** in the UML, shown with an opening solid ball; it implies the sender will not be specified, is not known, or that the message is coming from a random source. However, by convention a team or tool may ignore showing this, and instead use a regular message line without the ball, intending by convention it is a found message.

Focus of Control and Execution Specification Bars

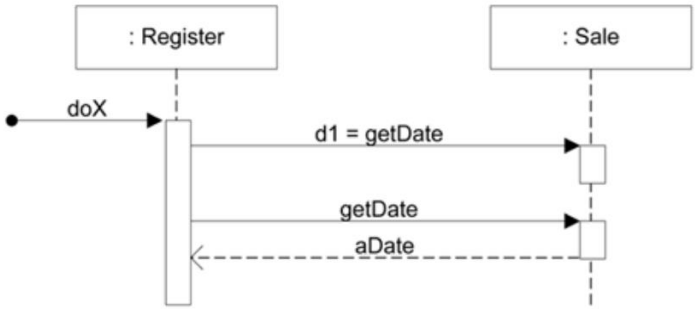
Sequence diagrams may also show the **focus of control** (informally, in a regular blocking call, the operation is on the call stack) using an **execution specification bar** (previously called an activation bar or simply an activation in UML 1). The bar is optional.

Guideline : Drawing the bar is more common (and often automatic) when using a UML CASE tool, and less common when wall sketching.

Illustrating Reply or Returns

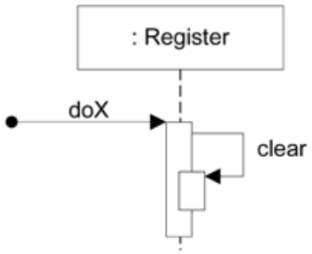
There are two ways to show the return result from a message:

1. Using the message syntax return `var = message(parameter)`
2. Using a reply (or return) message line at the end of an activation bar.



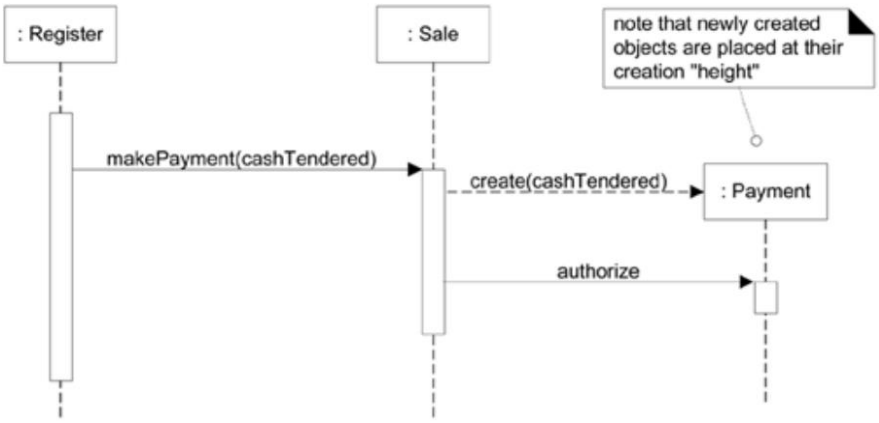
Messages to "self" or "this"

You can show a message being sent from an object to itself by using a nested activation bar



Creation of Instances

The UML-mandated dashed line. The arrow is filled if it's a regular synchronous message (such as implying invoking a Java constructor), or open (stick arrow) if an asynchronous call. The message name create is not required anything is legal but it's a UML idiom



The typical interpretation (in languages such as Java or C#) of a create message on a dashed line with a filled arrow is "**invoke the new operator and call the constructor**".

Object Lifelines and Object Destruction

In some circumstances it is desirable to show **explicit destruction** of an object. For example, when using C++ which does not have automatic garbage collection, or when you want to especially indicate an object is no longer usable (such as a closed database connection). The UML lifeline notation provides a way to express this destruction

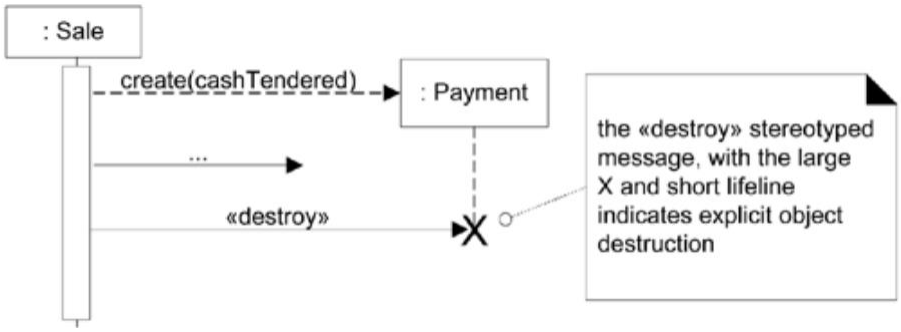


Diagram Frames in UML Sequence Diagrams

To support **conditional** and **looping constructs** (among many other things), the UML uses

frames

Frames are regions or fragments of the diagrams; they have an **operator** or **label** (such as loop) and a **guard** (conditional clause).

The following table summarizes some common frame operators:

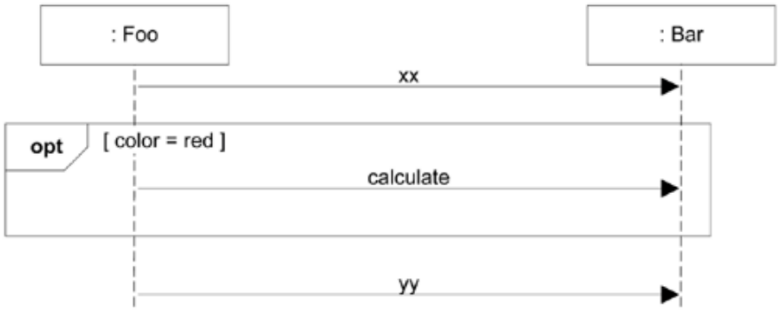
Frame Operator	Meaning
alt	Alternative fragment for mutual exclusion conditional logic expressed in the guards.
loop	Loop fragment while guard is true. Can also write *loop(n)* to indicate looping n times. There is discussion that the specification will be enhanced to define a *FOR* loop, such as *loop(i, 1, 10)*
opt	Optional fragment that executes if guard is true.
par	Parallel fragments that execute in parallel.
region	Critical region within which only one thread can run.

Looping

The LOOP frame notation to show looping is shown above

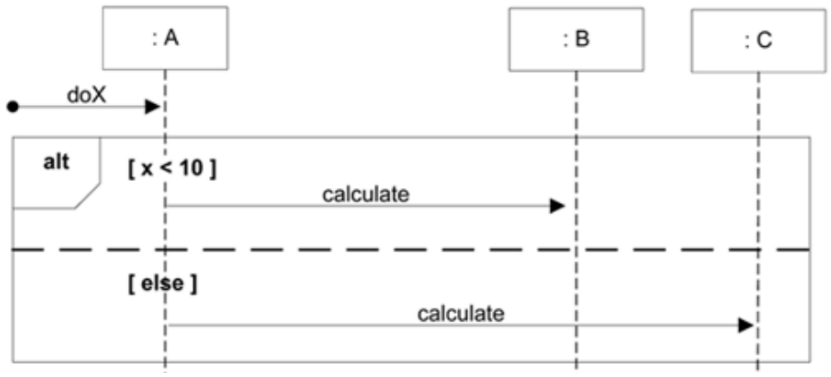
Conditional Messages

An **OPT frame** is placed around one or more messages. The **guard** is placed over the related lifeline.



Mutually Exclusive Conditional Messages

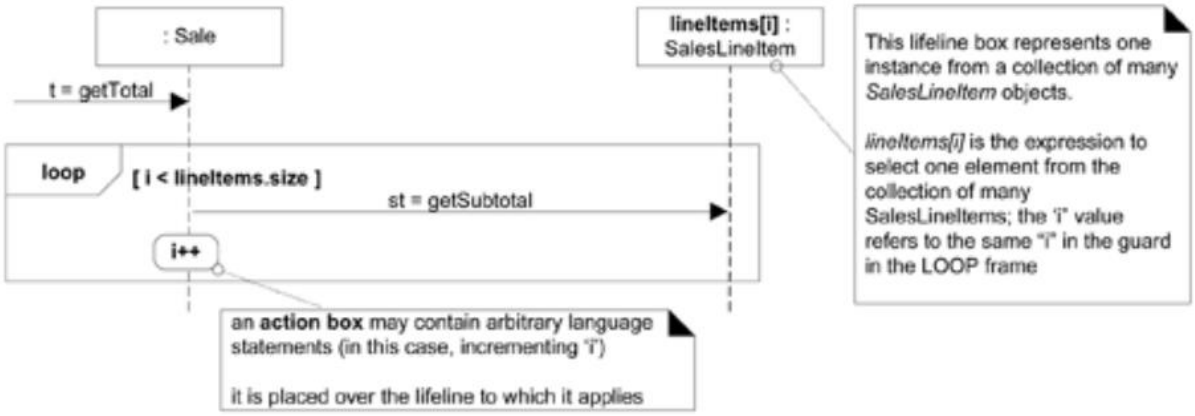
An **ALT frame** is placed around the mutually exclusive alternatives



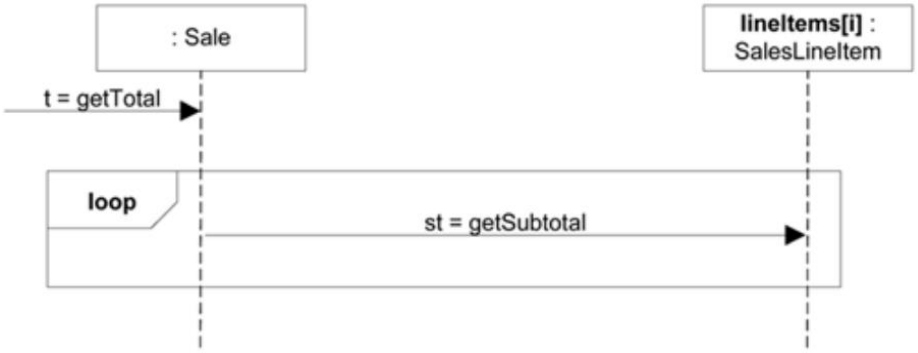
Iteration over a Collection

A common algorithm is to iterate over all members of a collection (such as a list or map), **sending the same message to each**. Often, some kind of iterator object is ultimately used, such as an implementation of java.util.Iterator or a C++ standard library iterator, although in the sequence diagram that low-level "mechanism" need not be shown in the interest of brevity or abstraction.

Explicit



Implicit



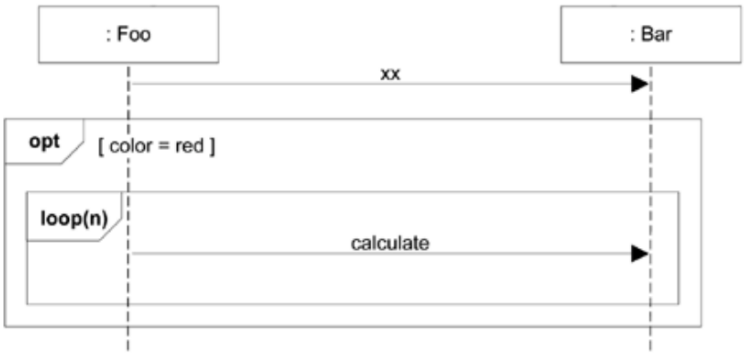
The selector expression is used to select one object from a group. Lifeline participants should represent one object, not a collection.

In Java, for example, the following code listing is a possible implementation that maps the explicit use of the incrementing variable

```
public class Sale
{
      private List<SalesLineItem> lineItems =
      new ArrayList<SalesLineItem>();
      public Money getTotal()
      {
            Money total = new Money();
            Money subtotal = null;
            for ( SalesLineItem lineItem : lineItems )
            {
                  subtotal = lineItem.getSubtotal();
                  total.add( subtotal );
            }
            return total;
      }
      // ...
}
```

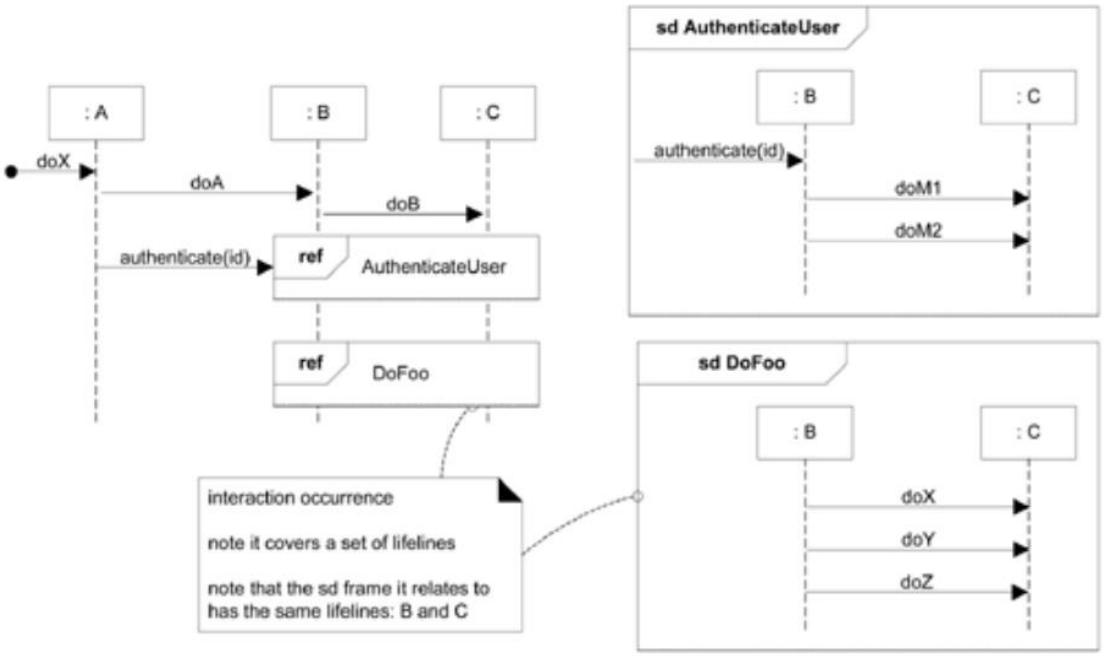
Nesting of Frames

Frames can be nested.



Relating Interaction Diagrams

An interaction occurrence (also called an **interaction use**) is a reference to an interaction within another interaction. It is useful, for example, when you want to **simplify a diagram** and **factor out a portion into another diagram**, or there is a reusable interaction occurrence. UML tools take advantage of them, because of their usefulness in relating and linking diagrams, Example interaction occurrence, sd and ref frames.



They are created with two related frames:

- a frame around an entire sequence diagram, labeled with the tag **sd** and a **name**, such as AuthenticateUser
- a frame tagged **ref** , called a reference , that **refers to another named sequence diagram**; it is the actual interaction occurrence

Interaction overview diagrams also contain a set of reference frames (interaction occurrences). These diagrams organized references into a larger structure of logic and process

flow.

Guideline: Any sequence diagram can be surrounded with an sd frame, to name it. Frame and name one when you want to refer to it using a ref frame.

RESPONSIBILITIES AND METHODS

The UML defines a responsibility as **"a contract or obligation of a classifier"**

Responsibilities are related to the **obligations** of an object in terms of its **behavior**.

Basically, these responsibilities are of the following two types:

Doing responsibilities of an object include:

- doing something itself, such as creating an object or doing a calculation
- initiating action in other objects
- controlling and coordinating activities in other objects

Knowing responsibilities of an object include:

- knowing about private encapsulated data
- knowing about related objects
- knowing about things it can derive or calculate

Responsibilities are assigned to classes of objects **during object design**.

For example,

"a `Sale` is responsible for **creating** `SalesLineItems`" (a doing),

"a `Sale` is responsible for **knowing its** `total`" (a knowing).

Relevant responsibilities related to **"knowing"** are often inferable from the **domain model**, because of the attributes and associations it illustrates.

The translation of responsibilities into classes and methods is influenced by the **granularity of the responsibility.**

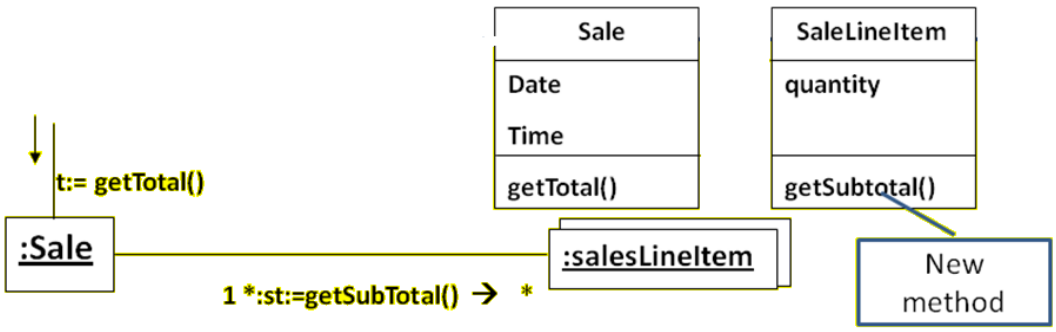
Responsibility to "provide access to relational databases" involves dozens of classes and hundreds of methods, packaged in a subsystem.

Responsibility to "create a `Sale`" may involve only one or few methods.

A responsibility is not the same thing as a method, but **methods are implemented to fulfill responsibilities.**

Responsibilities are implemented using methods that either **act alone** or **collaborate** with other methods and objects. the `Sale` class might define one or more methods to know its total; say, a method named `getTotal`.

To fulfill that responsibility, the `Sale` may collaborate with other objects, such as sending `getSubtotal` message to each `SalesLineItem` object asking for its subtotal.



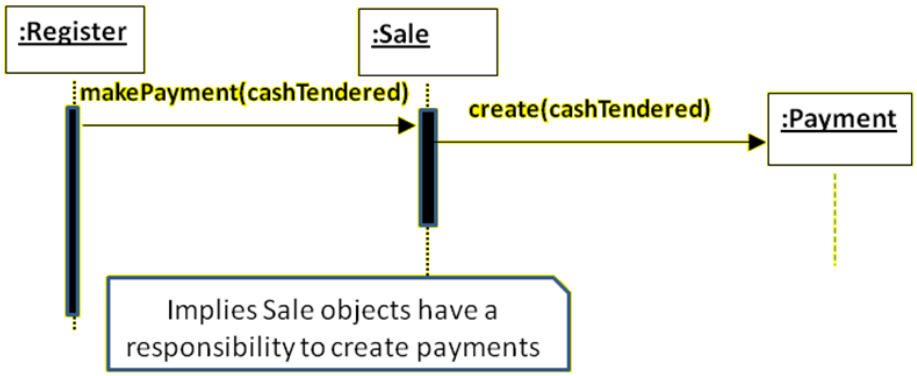
Responsibilities and Interaction Diagrams

Within the UML artifacts, a common context where these responsibilities (implemented as methods) are considered is **during the creation of interaction diagrams**

`Sale` objects have been given a responsibility to create `Payments`, which is invoked with a `makePayment` **message** and handled with a corresponding `makePayment` **method.**

Interaction diagrams show choices in assigning responsibilities to objects.

When created, decisions in responsibility assignment are made, which are reflected in what messages are sent to different classes of objects.



PATTERNS

A pattern is a named description of a problem and solution that can be applied to new context, patterns guide the assignment of responsibilities to objects

Experienced object-oriented developers build up **a repertoire of both general principles and idiomatic solutions** that guide them in the creation of software.

These principles and idioms, if codified in a structured format describing the problem and solution, and given a name, may be called **patterns**.

Patterns are **named problem/solution pairs that codify good advice** and principles often related to the assignment of responsibilities.

For example:

Pattern Name:

Solution:

Problem It Solves:

e.g. Information Expert : Assign a responsibility to the **class that has the information** needed to fulfill it.

All patterns ideally have suggestive names.

Naming a pattern, technique, or principle has the following advantages:

- It **supports chunking** and incorporating that concept into our understanding and memory.
- It **facilitates communication**. Naming a complex idea such as a pattern is an example of the power of abstraction
- **reducing a complex form** to a simple one by eliminating detail.

GRASP: GENERAL RESPONSIBILITY ASSIGNMENT SOFTWARE PATTERNS.

GRASP patterns have concise names such as *Information Expert, Creator, Protected Variations.*

The GRASP patterns **guide choices** in where to assign responsibilities.

These choices are reflected in **interaction diagrams.**

Applying Patterns

Pattern Types

1. Creator
2. Controller
3. Pure Fabrication
4. Information Expert
5. High Cohesion
6. Indirection
7. Low Coupling
8. Polymorphism
9. Protected Variations

i. Information Expert

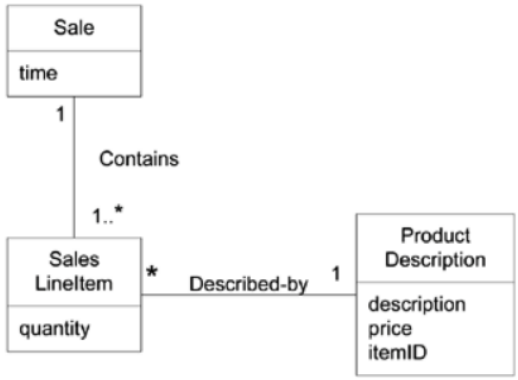
This pattern guides us to assign a responsibility to the information expert (class) that **has the *information*** necessary to fulfill the responsibility.

In the NextGEN POS application, **some class needs to know the grand total of a sale**.

Who should be responsible for knowing the grand total of a sale"? (Look up in the design model and the domain model) `

By Information Expert, we should look for that class of objects that has the information needed to determine the total.

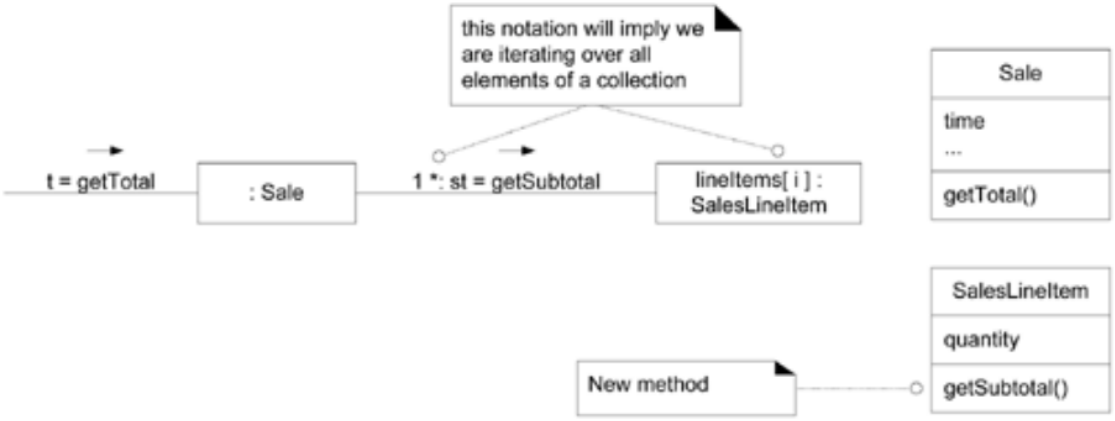
Only `Sales` instance in conceptual model knows the information about the grand total by having the all the information about the `SalesLineItem` in the Sale and the sum of their subtotals. Hence sale in information expert



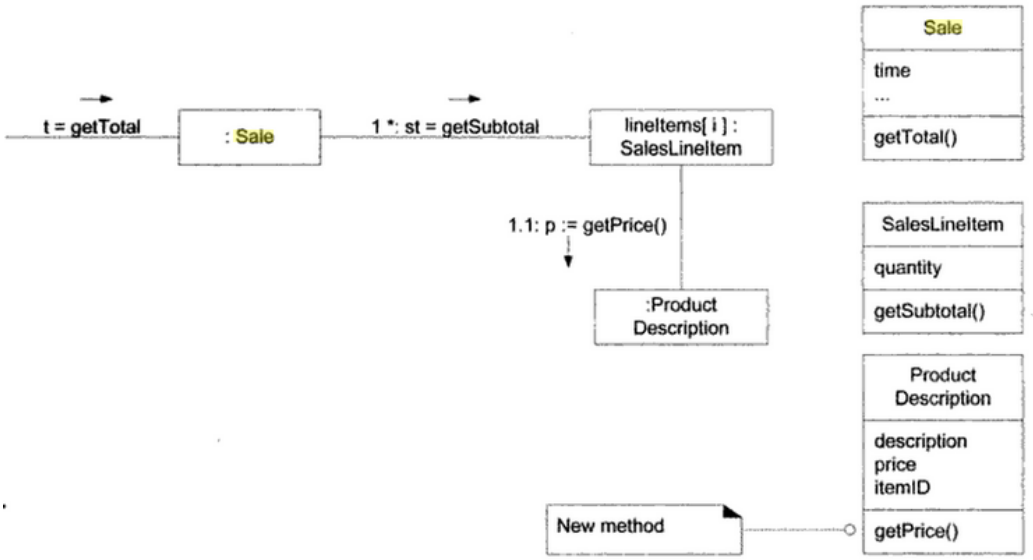
In terms of an interaction diagram, this means that the `Sale` needs to send `get-Subtotal` messages to each of the `SalesLineItems` and sum the results

To fulfill the responsibility of knowing and answering its `subtotal`, a `SalesLineItem` needs to know the product price.

The `ProductDescription` is an information expert on answering its `price`; therefore, a message must be sent to it asking for its `price`.



Design Class	Responsibility
Sale	knows sale total
SalesLineItem	knows line item subtotal
ProductDescription	knows product price



Design Class	Responsibility
Sale	knows sale total
SalesLineItem	knows line item subtotal
ProductSpecification	knows product price

ii. Creator

The class which has the responsibility of **creating an instance of other class**

Assign class B the **responsibility to create an instance of class A** if one or more of

the following is true:

- B *aggregates* A objects.
- B *records* instances of A objects.
- B *contains* A objects.
- B *closely uses* A objects.
- B *has the initializing data* that will be passed to A when it is created (thus B is an Expert with respect to creating A).

B is a *creator* of A objects.

If more than one option applies, prefer a class B which *aggregates* or *contains* class A.

Aggregate *aggregates* Part, Container *contains* Content, and Recorder *records*

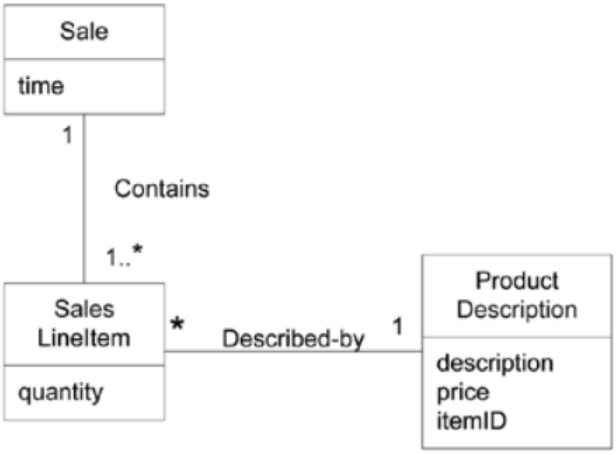
Recorded are all very common relationships between classes in a class diagram.

Creator suggests that the enclosing container or recorder class is a good candidate for the responsibility of creating the thing contained or recorded.

Who should be responsible for creation of a new class?

In the POS application, who should be responsible for creating a `SalesLineItem` instance? By Creator, we should look for a class that aggregates, contains, and so on, `SalesLineItem` instances.

In the POS application, who should be responsible for creating a `SalesLineItem` instance? By Creator, we should look for a class that aggregates, contains, and so on, `SalesLineItem` instances



Since a `Sale` contains (in fact, aggregates) many `SalesLineItem` objects, the `Creator` pattern suggests that `Sale` is a good candidate to have the responsibility of creating `SalesLineItem` instances.

iii. High Cohesion

This pattern advices us on **keeping objects focused, understandable, and manageable, and as a side effect, support Low Coupling**

Assign responsibility so that **cohesion remains high**

cohesion (or more specifically, functional cohesion) is a measure of **how strongly related** and focused the responsibilities of an element are.

An element with **highly related responsibilities**, and **which does not do a tremendous amount of work**, has **high cohesion.**

A class with **low cohesion** does **many unrelated things**, or **does too much work.**

Such classes are undesirable; they suffer from the following problems

- hard to **comprehend**
- hard to **reuse**
- hard to **maintain**
- delicate; constantly **effected by change**

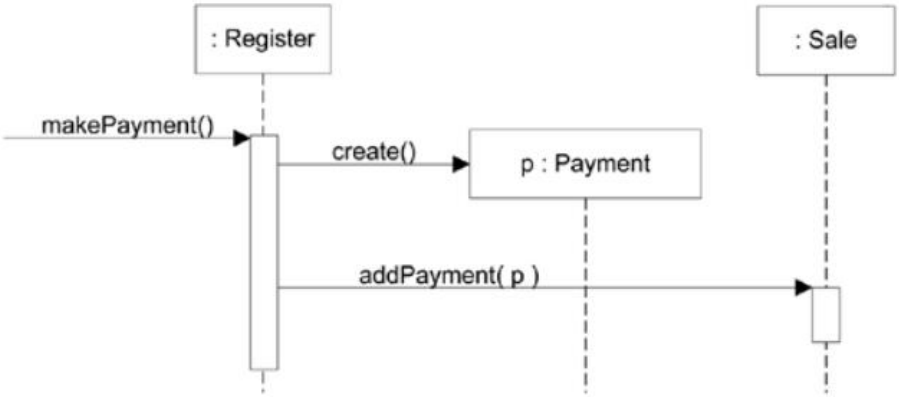
Low cohesion classes often represent a very "large grain" of abstraction, or have taken on responsibilities that **should have been delegated to other objects**

Assume we have a need to create a (cash) `Payment` instance and associate it with the `Sale`. What class should be responsible for this? Since `Register` records a `Payment` in the real-world domain, the Creator pattern suggests `Register` as a candidate for creating the `Payment`. The `Register` instance could then send an `addPayment` message to the `Sale`, passing along the new `Payment` as a parameter,

This assignment of responsibilities places the responsibility for making a payment in the `Register`. The `Register` is taking on part of the responsibility for fulfilling the `makePayment` system operation.

But if we continue to make the `Register` class responsible for doing some or most of the work related to more and more system operations, it will become increasingly burdened with tasks and become incohesive.

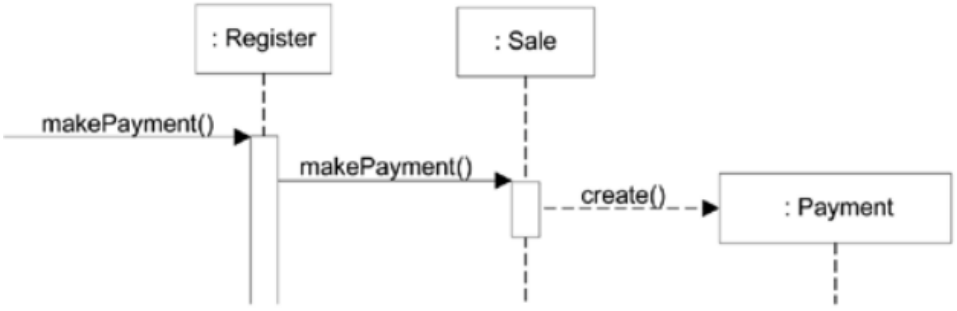
Here the `Register` is taking on part of the responsibility fulfilling the `makePayment` function.



The second design delegates the payment creation responsibility to the `Sale`

Since the second design supports both high cohesion and low coupling, it is desirable.

Here the `Register` is taking on part of the responsibility fulfilling the `makePayment` function.



Hence when a class does tremendous amount of work it is not considered cohesive

Here we see that the Register is giving the payment creation responsibility to the `Sale`. This supports higher cohesion

iv. Low Coupling

This pattern advises us on supporting low dependency, low change impact, and increased reuse

Assign a responsibility so that **the coupling remains low.**

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

An element with low (or weak) coupling is **not dependent on too many other elements**

A class with the high coupling relies upon many other classes. Hence such classes are undesirable

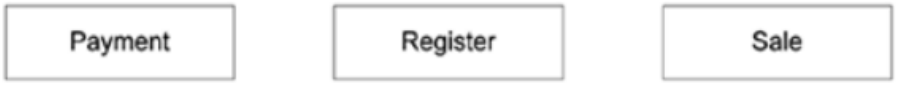
A class with **low cohesion does many unrelated things**, or does too much work.

Such classes are undesirable; they suffer from the following problems:

- hard to **comprehend**
- hard to **reuse**
- hard to **maintain**
- delicate; constantly **effected by change**

Low cohesion classes often represent a very "large grain" of abstraction, or have taken on responsibilities that should have been delegated to other objects

Consider the following partial class diagram from a NextGen case study:

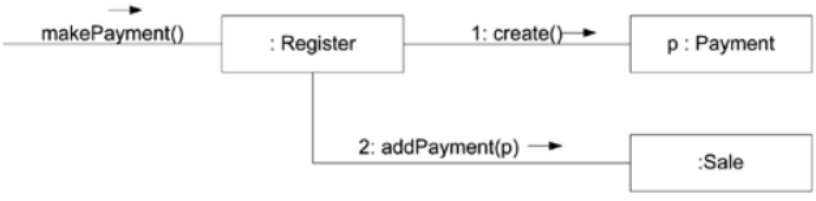


Assume we have a need to create a `Payment` instance and associate it with the `Sale`. What class should be responsible for this? Since a `Register` "records" a `Payment` in the real-world domain, the Creator pattern suggests `Register` as a candidate for creating the `Payment`. The `Register` instance could then send an `addPayment` message to the `Sale`, passing along the new `Payment` as a parameter.

A need to create a `Payment` instance and associate it with the `Sale`.

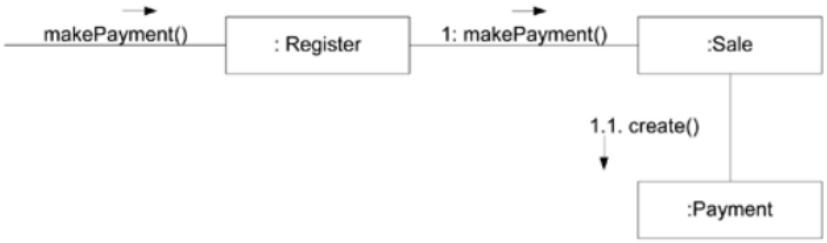
What class should be responsible for this?

Since a `Register` "records" a `Payment` in the real-world domain, the Creator pattern suggests `Register` for creating the `Payment`.



The `Register` instance then sends an `addPayment` message to the `Sale`, passing along the new `Payment` as a parameter.

This assignment of responsibilities **couples the `Register` class to knowledge of the `Payment` class.**



In both the above examples we find that the second does not increase the coupling hence it is better.

Design 1, the `Register` creates the `Payment`, adds coupling of `Register` to `Payment`

Design 2, the `Sale` does the creation of a `Payment`, does not increase the coupling.

Purely from the point of view of coupling, prefer Design 2 because it maintains overall lower coupling.

Low Coupling is a principle to keep in mind during all design decisions; it is an underlying goal to continually consider. It is an evaluative principle that a designer applies while evaluating all design decisions.

In object-oriented languages such as C++, Java, and C#, common forms of coupling from *TypeX* to *TypeY* include:

- *TypeX* has an attribute (data member or instance variable) that refers to a *TypeY* instance, or *TypeY* itself.
- A *TypeX* object calls on services of a *TypeY* object.
- *TypeX* has a method that references an instance of *TypeY,* or *TypeY* itself, by any means. These typically include a parameter or local variable of type *TypeY,* or the object returned from a message being an instance of *TypeY.*
- *TypeX* is a direct or indirect subclass of *TypeY.*

- *TypeY* is an interface, and *TypeX* implements that interface.

Low Coupling supports the design of classes that are more independent, which reduces the impact of change.

A subclass is strongly coupled to its superclass. The decision to derive from a superclass needs to be carefully considered since it is such a strong form of coupling.

Benefits

- not affected by changes in other components
- simple to understand in isolation
- convenient to reuse

Scenarios that illustrate varying degrees of functional cohesion:

1. Very low cohesion: A class is solely responsible for many things in very different functional areas.

   Assume the existence of a class called RDB -RPC-Interface which is completely

   responsible for interacting with relational databases and for handling remote procedure

   calls. These are two vastly different functional areas, and each requires lots of

   supporting code. The responsibilities should be split into a family of classes related to

   RDB access and a family related to RPC support.

2. Low cohesion: A class has sole responsibility for a complex task in one functional area

   Assume the existence of a class called

   RDBInterface which is completely responsible for interacting with relational databases. The methods of the class are all related, but there are lots of them, and a tremendous amount of supporting code; there may be hundreds or thousands of methods. The class should split into a family of lightweight classes sharing the work to provide RDB access.

3. High cohesion: A class has moderate responsibilities in one functional area and collaborates with other classes to fulfill tasks.
   Assume the existence of a class called RDBInterface that is only partially responsible

   for interacting with relational databases. It interacts with a dozen other classes related

   to RDB access in order to retrieve and save objects.

4. Moderate cohesion: A class has lightweight and sole responsibilities in a few different areas that are logically related to the class concept but not to each other.
   Assume the existence of a class called Company that is completely responsible for (a)

   knowing its employees and (b) knowing its financial information. These two areas are

not strongly related to each other, although both are logically related to the concept of

a company. In addition, the total number of public methods is small, as is the amount

of supporting code.

v. Controller

This pattern gives advice on identifying the first object beyond the UI layer that receives and coordinates ("controls") a system operation

System operations explored during the analysis of SSD are the major input events in the system. For example, when a cashier using a POS terminal presses the "End Sale" button, he is generating a system event indicating "the sale has ended." Similarly, when a writer using a word processor presses the "spell check" button, he is generating a system event indicating "perform a spell check." A controller is the first object beyond the UI layer that is responsible for receiving or handling a system operation message.

Assign the responsibility for **receiving or handling a system event message** to a class representing one of the following choices:

- Represents the overall "system," a "root object," a device that the software is running
  within, or a major subsystem these are all variations of a facade controller

- Represents a use case scenario within which the system event occurs, often named <UseCaseName> Handler, <UseCaseName> Coordinator, or <Use-CaseName> Session *(use-case or session controller).*

Use the same controller class for all system events in the same use case scenario.

Informally, a session is an instance of a conversation with an actor. Sessions can be of

any length but are often organized in terms of use cases (use case sessions).

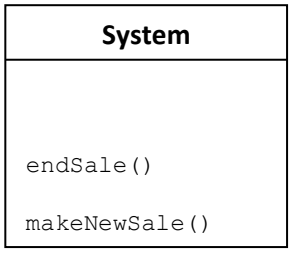
Who should be responsible for handling an input system event?

An input **system event** is an event generated by an external actor. They are associated with **system operations**.operations of the system in response to system events, just as messages and methods are related.

For example, when a cashier using a POS terminal presses the "End Sale" button, he is generating a system event indicating "the sale has ended." Similarly, when a writer using a word processor presses the "spell check" button, he is generating a system event indicating "perform a spell check."

A Controller is a non-user interface object responsible for receiving or handling a system event. A Controller defines the method for the system operation.

Several system operations of POST system:



Use the same controller class for all system events in the same use case scenario.

Informally, a session is an instance of a conversation with an actor.

Sessions can be of any length, but are often organized in terms of use cases

A Controller is a **non-user interface object** responsible for **receiving or handling a system event**.

A Controller defines the method for the system operation.

During analysis system operations maybe assigned to the class System, to indicate that they are system operations. During design, a Controller class is assigned the responsibility of system operations

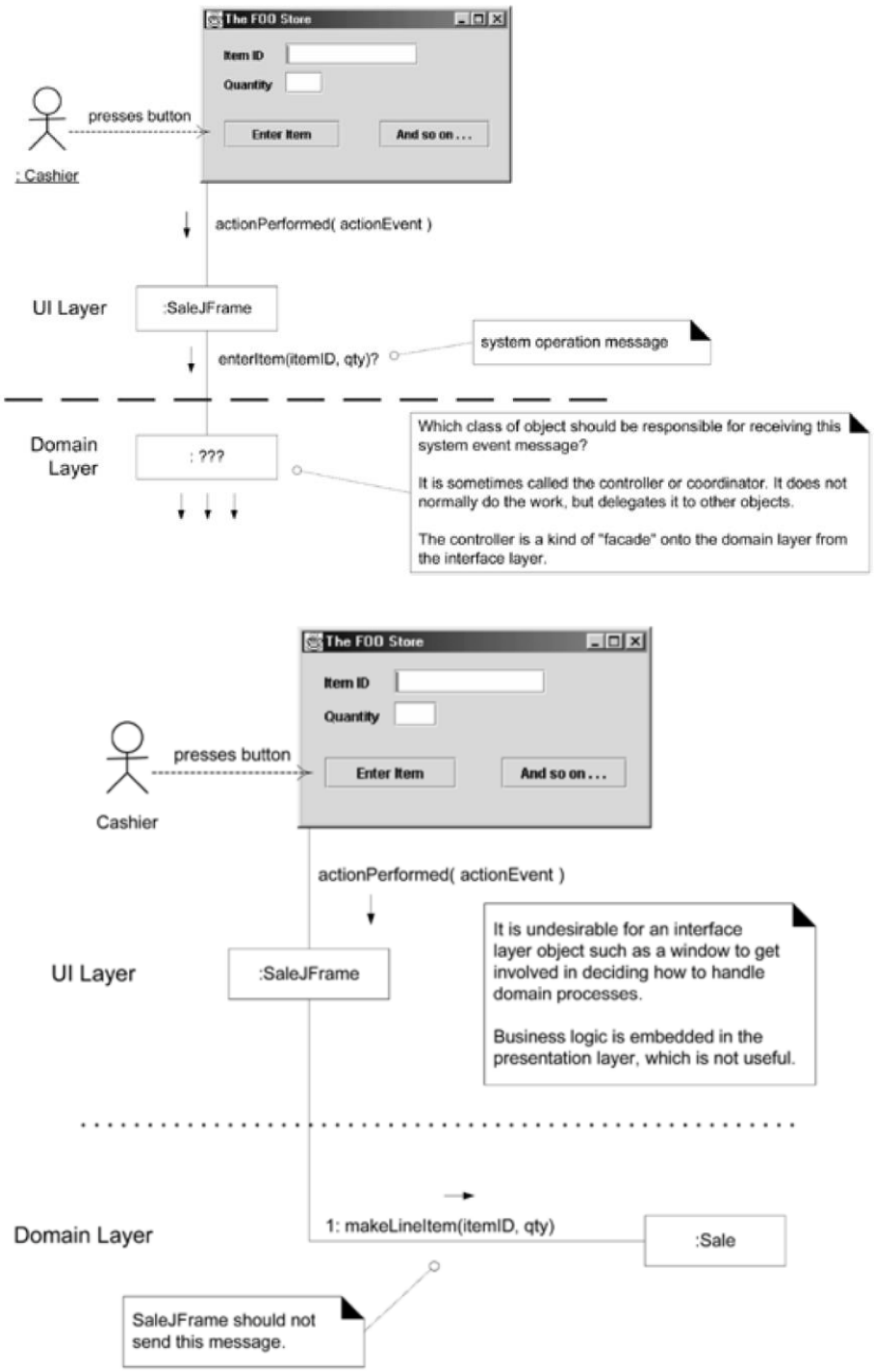
Choosing the Controller Class

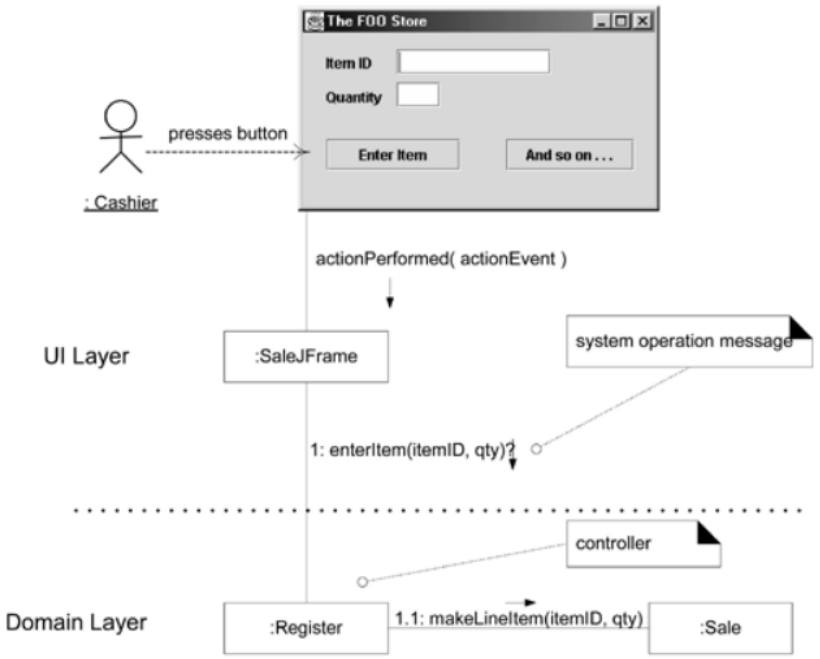
Our first choice involves choosing the controller for the system operation message `enterItem`

Choices: -

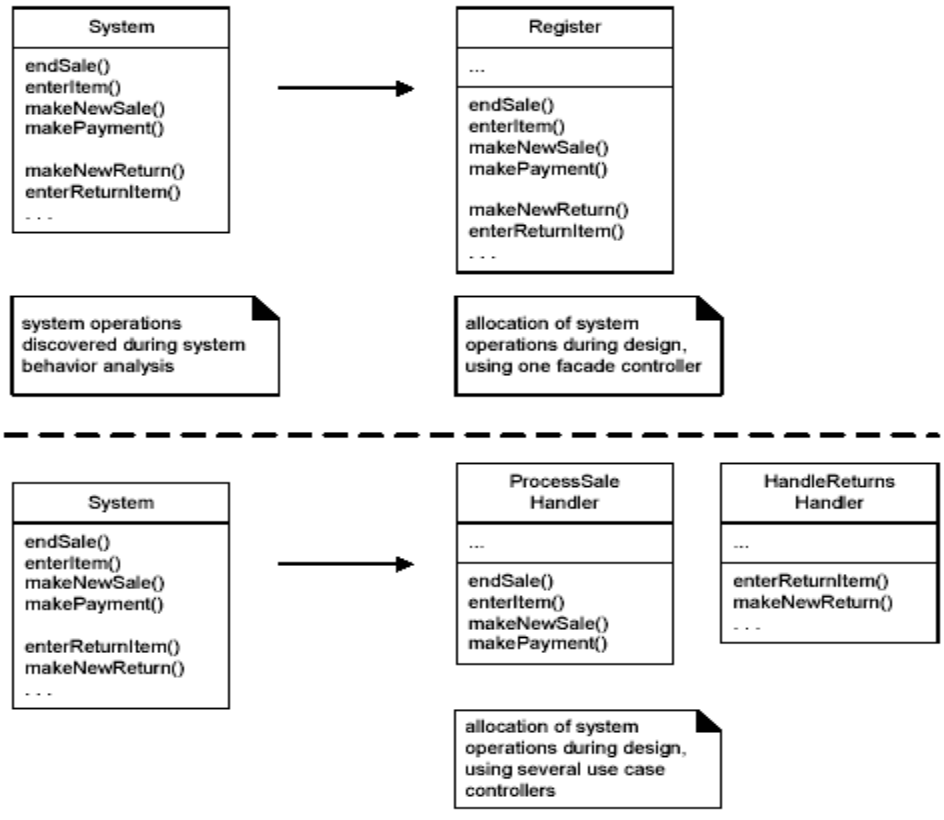
represents the overall "system," device, or `Register`, `POSTSystem` subsystem represents a receiver or handler of all system `ProcessSaleHandler`, events of a use case scenario `ProcessSaleSestsion`

Thus the communication diagram begins by sending the enterItem message with a UPC and quantity parameter to a POST instance.


The FOO Store
Item ID
Quantity
Enter Item
And so on . . .
presses button
: Cashier
actionPerformed(actionEvent)
UI Layer
:SaleJFrame
enterItem(itemID, qty)?
system operation message
Domain Layer
: ???
Which class of object should be responsible for receiving this system event message?
It is sometimes called the controller or coordinator. It does not normally do the work, but delegates it to other objects.
The controller is a kind of "facade" onto the domain layer from the interface layer.
The FOO Store
Item ID
Quantity
Enter Item
And so on . . .
presses button
Cashier
actionPerformed(actionEvent)
UI Layer
:SaleJFrame
It is undesirable for an interface layer object such as a window to get involved in deciding how to handle domain processes.
Business logic is embedded in the presentation layer, which is not useful.
Domain Layer
1: makeLineItem(itemID, qty)
:Sale
SaleJFrame should not send this message.



Allocation of system operations



VISIBILITY

In common usage, **visibility** is the ability of an object to "see" or have a reference to another object.

It is related to the issue of scope: Is one resource (an instance) within the scope of another?

To send a message from one object to another, the receiver object must be visible to the sender, so the sender has to have a pointer or reference to the receiver.

There are four common ways that visibility can be achieved from object A to object *B:*

• **Attribute visibility**—B is an attribute of A.

• **Parameter visibility**—B is a parameter of a method of A.

• **Local visibility**—B is a (non-parameter) local object in a method of A.

• **Global visibility**—B is in some way globally visible.

i. Attribute Visibility

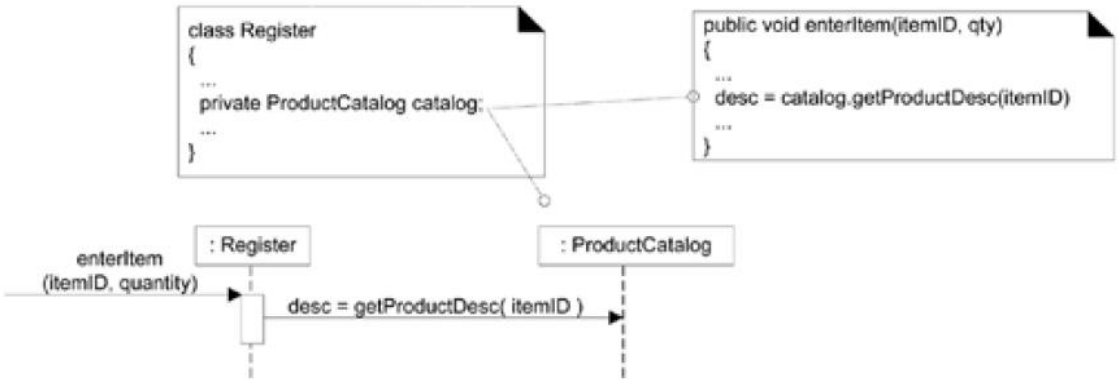
It exist from A to B when B is an attribute of A.

It is permanent visibility because it persists as long as A and B exists.

It is very common form of visibility in object oriented systems.

```
public class Register{
  …
  private ProductCatalog catalog;
```

```
    ...
}
```

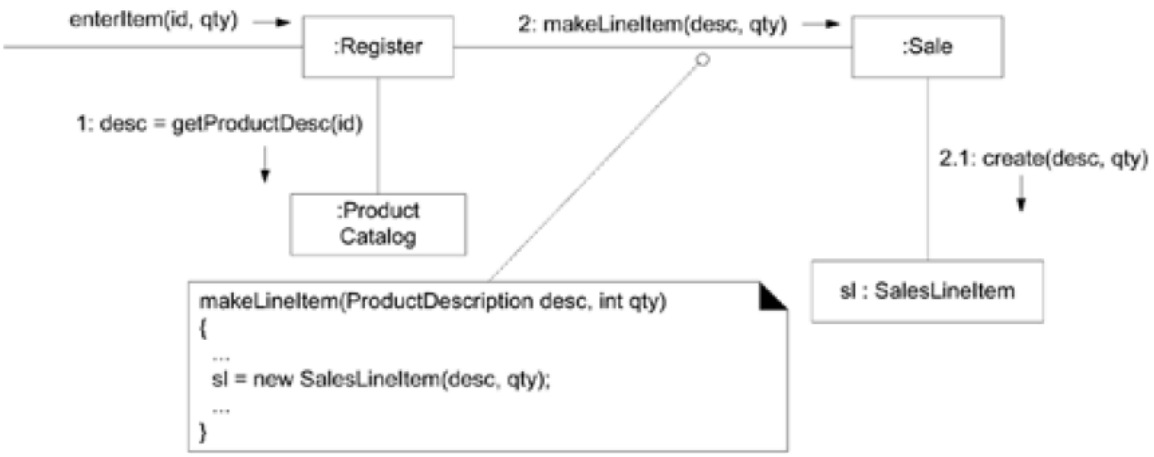


This visibility allows the `Register` to send the `getProductDescription` message to a `ProductCatalog`

ii. Parameter Visibility

It exists from A to B when B is passed as a parameter to a method of A.

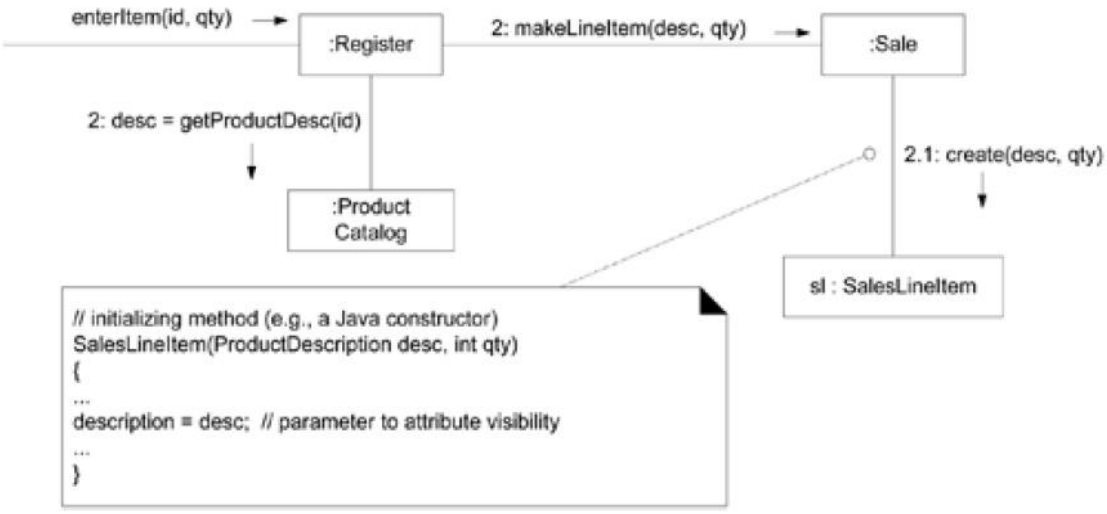
It is temporary as it persists only within the scope of the method.



When the `makeLineItem` message is sent to Sale instance, a `ProductDescription` instance is passed as a parameter. Within the scope of the `makeLineItem` method, the sale has a Parameter visibility to a `ProductSpeification`.

It is common to transform parameter visibility into attribute visibility.

When `Sale` creates a new `SalesLineItem`, it passes `ProductDescription` (`desc` object in this case) in to its initializing method (constructor). Within the initializing method, the parameter is assigned an attribute, thus establishing attribute visibility



iii. Locally Declared Visibility

Local visibility from A to B exists when B is declared as a local object within a method of A.

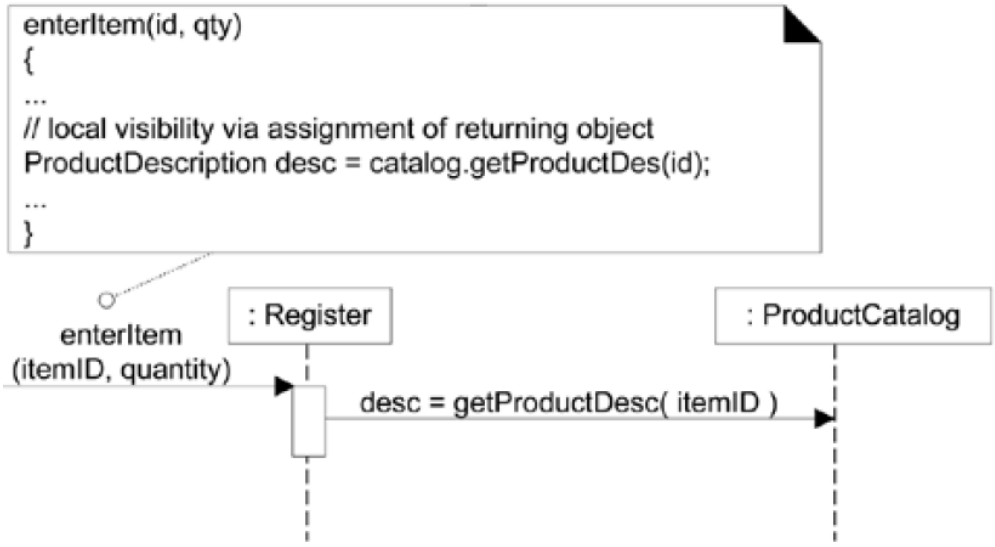
It is a relatively temporary visibility because it persists only within the scope of the method

Two common means by which local visibility is achieved are:

- Create a new local instance and assign it to a local variable.
- Assign the returning object from a method invocation to a local variable.

Ex: `anObject.getAnotherObject.doSomething();`

As with parameter visibility, it is common to transform locally declared visibility into attribute visibility.



A method may not explicitly declare a variable but one may implicitly exist as the **result of a returning object from a method invocation**

//there is an implicit local visibility to the foo object returned via getFoo call

```
anObject.getFoo().doBar();
```

iv. Global Visibility

It exists between A to B when B is global to A.

It is relatively permanent visibility because it persists as long as A and B exists

It exists between A to B when B is global to A.

It is relatively permanent visibility because it persists as long as A and B exists

One way to achieve global visibility is to assign an instance to a global variable,

which is possible in some languages, such as C++, but not others, such as Java.

The least common form of visibility in OO Systems.

One way to achieve global visibility is to assign an instance to a global variable,

which is possible in some languages, such as C++, but not others, such as Java.

Public:

Any outside classifier with visibility to the given classifier can use the feature; specified by pre-pending the symbol "+"

Protected:

Any descendant of the classifier can use the feature; specified by pre-pending the symbol "#"

Private:

Only the classifier itself can use the feature; specified by pre-pending the symbol "-"

Q. Which would you use if you wanted a relatively permanent connection?

A. attribute, or global

Q. Which would you use if you didn't want a permanent connection?

A. parameter, or local

Q. How would you create a local visibility?

A. Create a new instance - use result of a method call

Q. How would you achieve a global visibility?

A. Use a global variable in C++, static (or class) variable (in C++ or Java) - use the Singleton pattern (a static method that returns the object)

DESIGN CLASS DIAGRAMS (DCD)

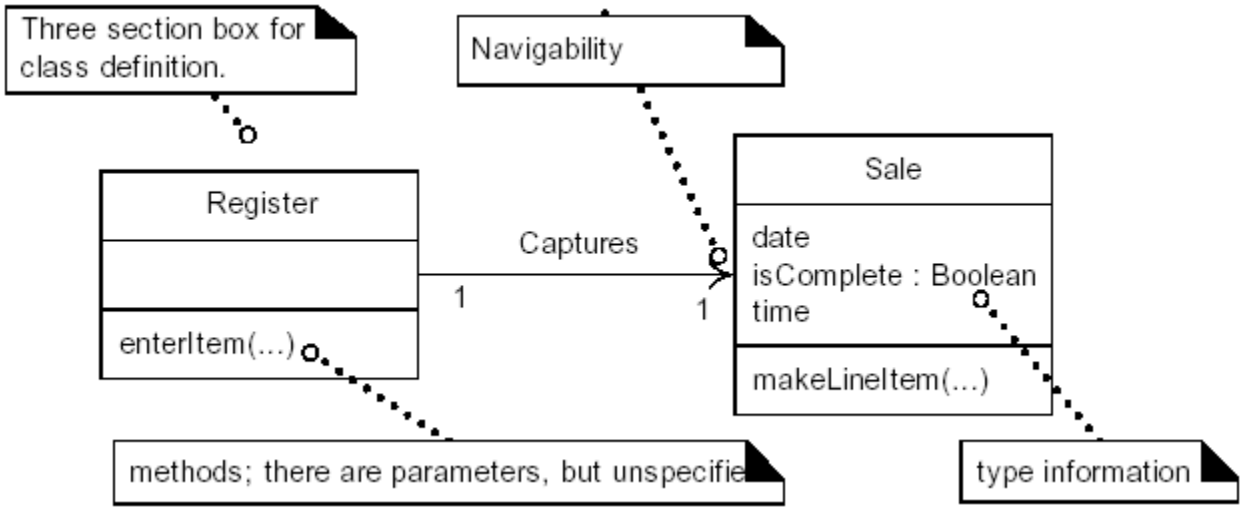
Activities and Dependencies

The creation of a class diagram is **dependent on Interaction Diagram**: this tells the designer about the software classes that participate in the solution

Conceptual Model: The designers adds details to this class definitions.

"Class Diagrams are created in parallel with Interaction Diagrams"

Example DCD



DCD and UP Terminology

A design class diagram (DCD) illustrates the specifications for software classes and interfaces (for example, Java interfaces) in an application. Typical information includes:

• classes, associations and attributes

- interfaces, with their operations and constants
- methods
- attribute type information
- navigability
- dependencies

Conceptual classes in the Domain Model show real-world concepts, where as design classes in the DCDs show definitions for software classes

During analysis -> emphasize domain concepts

During design -> shift to software artifacts

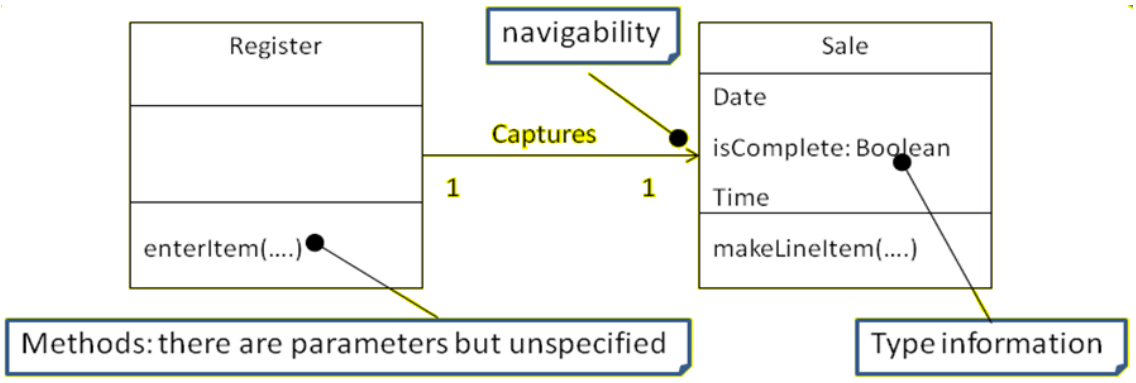
UML has no explicit notation for DCDs

It illustrates the specification for software classes and interfaces in an application.

Typical information that it includes is classes, association and attributes interface with their operation and constants

Methods Attribute type information

Navigability Dependencies



STEPS FOR MAKING DESIGN CLASS DIAGRAM:

- **Identify all the classes** participating in the software solution. This should be done by analyzing the interaction diagram.

- **Draw them** in a class diagram

- **Duplicate the attributes from** the associated concepts in the **conceptual model**

- **Add methods names** by analyzing the **interaction diagrams**

- **Add type information** to the attributes and methods

- **Add the associations** necessary to support the required attribute visibility

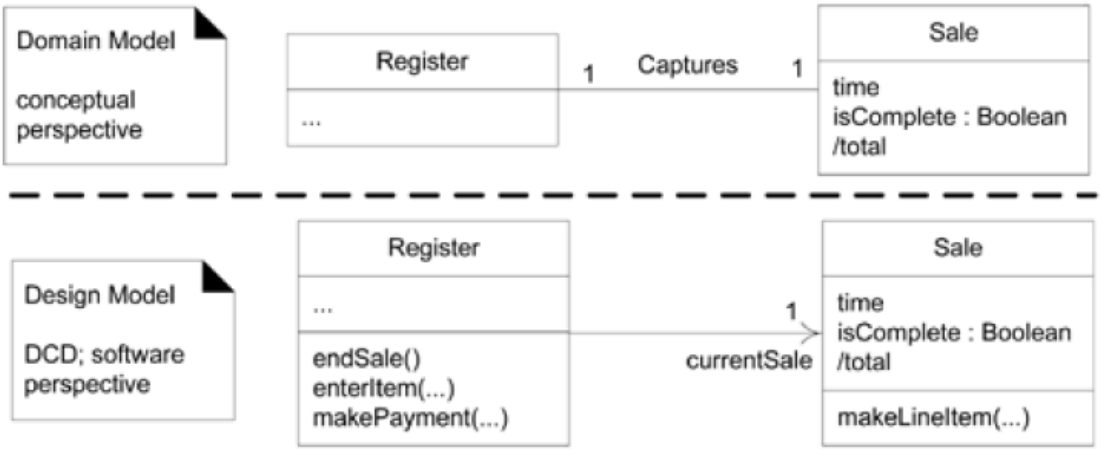
- **Add navigability arrow** to the association to indicate direction of attribute visibility

- **Add dependency relationship** line to indicate non-attribute visibility

Domain Model vs. Design Model Classes

In the UP Domain Model, a `Sale` does not represent a software definition;

rather, it is an abstraction of a real-world concept about which we are interested in making a statement. By contrast, DCDs express—for the software application—the definition of classes as software components.

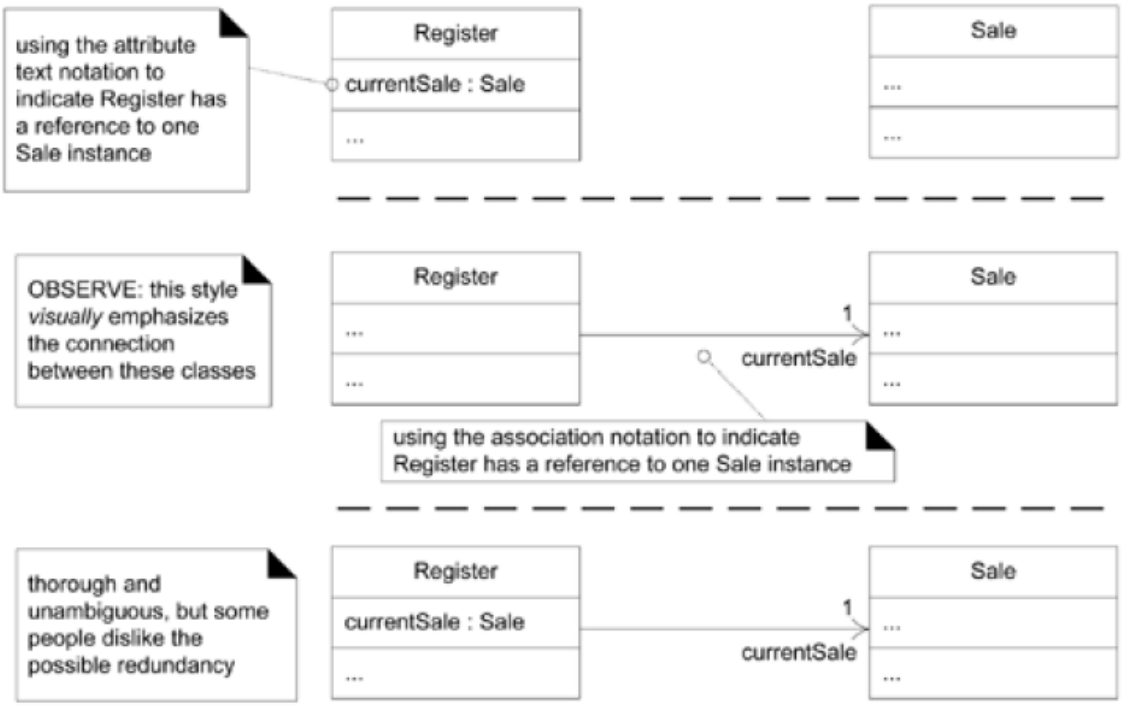


UML uses Class Diagram for Design Class Diagram

Showing UML Attributes

Attributes of a classifier are show in several ways:

- Attribute text notation, such as currentSale:Sale
- Association line notation
- Both together

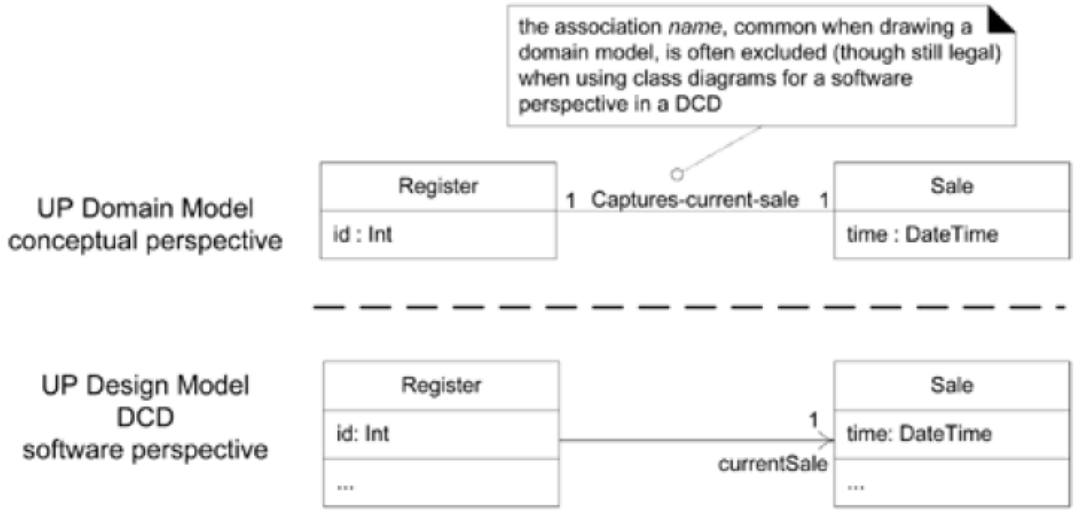


The full format of text attribute notation:

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

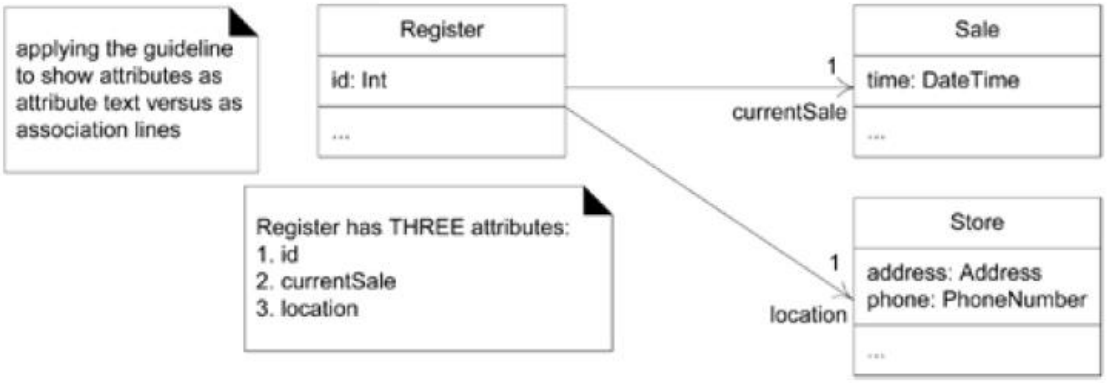
The full format of attribute as association line:

- A navigability arrow pointing from source (`Register`) to target (`Sale`) object indicating that `Register` object has an attribute of one `Sale`
- A multiplicity at the target end
- A role name only at the target end to show the attribute name
- No association name



Association name is excluded when using class diagrams for a software perspective in a DCD

Although different style exist in the UML notations they boil down to the same thing when implemented in code




```
public class Register{
      private int id;
      private Sale currentSale;
      private Store location;
}
```

UML Notation for Association End

The end of an association can have

- a navigability arrow
- a role name to indicate the name of the attribute
- multiplicity value * or 0..1
- a property string e.g. {ordered} or {ordered, list}

  {ordered} is a keyword in UML that signifies that the elements in the collection are ordered

  {unique} implies a set of unique elements

Operations and Methods

Operations:

The bottom most section of a Class box is for showing the signatures of operations

The syntax of an operation:

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

The **property string contains additional information**, such as exceptions that may be raised if the operation is abstract etc

Operation signatures may also be **written in programming languages as well**

```
+getPlayer (name:String):Player{exception IOException}

Public Player getPlayer(String name) throws IOException
```

Difference between an Operation and a Method

- An operation is not a method
- An UML operation is a declaration with name, parameters, return type, exception list, a set of constraints of pre and post conditions but **has no implementations**
- Methods are implementations

Showing Methods in UML Class Diagram

UML method is the implementation of an operation so if constraints are defined the method must satisfy

A method may be shown in several ways

- In interaction diagrams, by the details and sequence of messages
- In Class diagrams with a UML note symbol stereotyped with <<method>>

Keywords:

A UML keyword is a textual adornment to categorize a model element

The keyword used to categorize that a classifier box is an interface is <<interface>>

The keyword used to categorize that a classifier box is an actor is <<actor>>

Some key words maybe show in curly braces {abstract}

Keyword	Meaning	Example Usage
«actor»	classifier is an actor	in class diagram, above classifier name
«interface»	classifier is an interface	in class diagram, above classifier name
{abstract}	abstract element; can't be instantiated	in class diagrams, after classifier name or operation name
{ordered}	a set of objects have some imposed order	in class diagrams, at an association end

Stereotypes, Profiles and Tags

Stereotypes are not keywords but they are also show in guillemets symbols

<<authorship>>

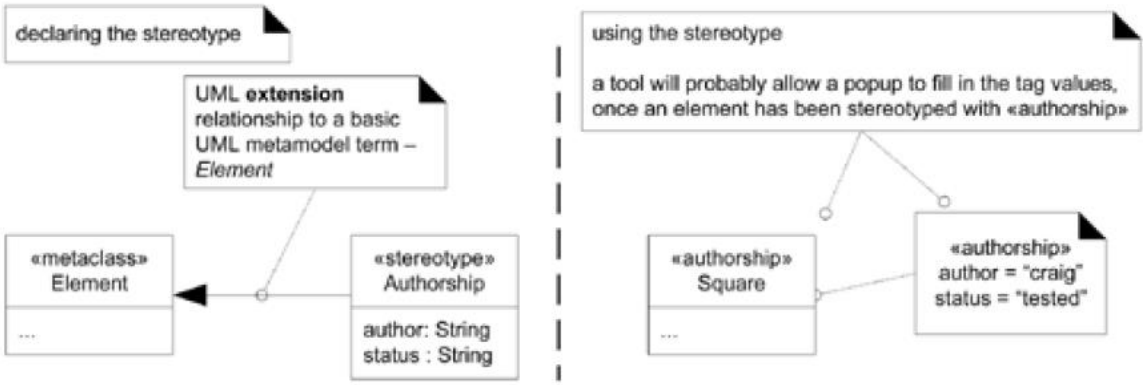
A stereotype represents a refinement of an existing modeling concept and is defined within a UML profile

UML provides many stereotypes such as <<destroy>> and allows user defined ones as well

They provide an extension mechanism in the UML

A stereotype declares a set of tags using attribute syntax

When an element or class is marked with a stereotype, all tags apply to the element and can be assigned values



UML Property and Property Strings

A property is a named value denoting a characteristic of an element

Some properties are predefined in the UML like visibility a property of an operation

A textual approach to represent property of elements uses the UML property string

`{name1 =value1, name2=value2…}` format e.g. `{abstract, visibility=public}`

Generalization, Abstract Classes, Abstract Operations

Generalization in the UML is shown with a solid line and a fat triangular arrow from the subclass to superclass

Abstract classes and operations can be shown with an {abstract} tag or by italicizing the name

Final classes and operations cannot be overridden in subclasses and are shown with the {leaf} tag

Dependency

Dependency lines are especially common on class and package diagrams. The UML includes a general dependency relationship that 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 is **illustrated with a dashed arrow line from the client to supplier**.

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

There's a special UML line to **show the superclass**, one to **show implementation of an interface**, and one **for attributes** (the attribute-as-association line).

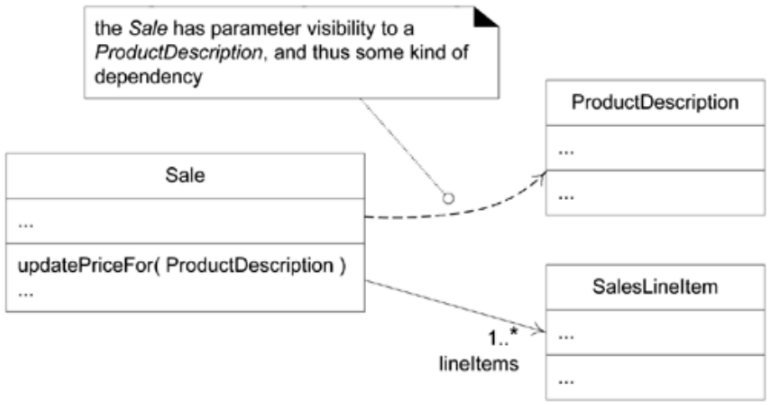
In class diagrams use the dependency line **to depict global, parameter variable, local variable, and static-method** (when a call is made to a static method of another class) **dependency** between objects.

For example, the following Java code shows an updatePriceFor method in the Sale class:

```
public class Sale
{
      public void updatePriceFor( ProductDescription description
      )
      {
            Money basePrice = description.getPrice();
            //...
      }
      // ...
}
```

The updatePriceFor method receives a ProductDescription parameter object and then sends it a getPrice message. Therefore, the **Sale object has parameter visibility to the ProductDescription**, and message-sending coupling, and thus a dependency on the ProductDescription.

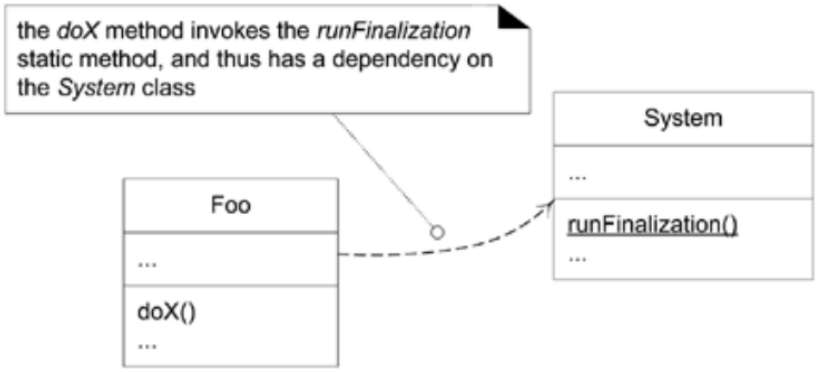
If the ProductDescription class is changed, the **Sale class could be affected**.



Another example: A doX method in the Foo class:

```
public class Foo
{
      public void doX()
      {
            System.runFinalization();
            //...
      }
// ...
}
```

The doX method **invokes a static method on the System class**. Therefore, the Foo object has a **static-method dependency** on the System class.



Dependency Labels

To show the type of dependencythe dependency line can be labeled with keywords or stereotypes.

COMPOSITION OVER AGGREGATION

Aggregation is a vague kind of association in the UML that loosely suggests whole-part relationships

Composition (composite aggregation), is a **strong kind of whole-part aggregation** and is useful to show in some models.

A composition relationship implies that

- an instance of the part (such as a Square) **belongs to only one composite instance** (such as one Board) at a time
- the part **must always belong** to a composite (no free-floating Fingers)
- 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.

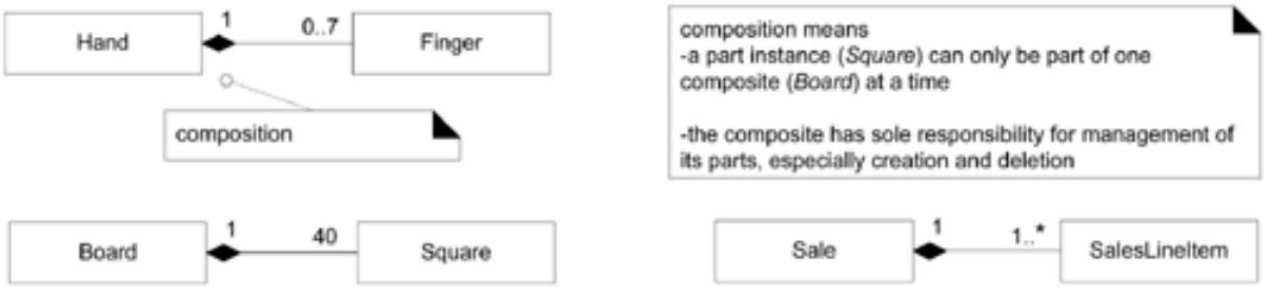
Related to this constraint is that **if the composite is destroyed**, its parts must either be destroyed, or attached to another composite

For example, if a physical paper Monopoly game board is destroyed, we think of the squares as being destroyed as well (a conceptual perspective).

Likewise, if a software Board object is destroyed, its software Square objects are destroyed, in a DCD software perspective.

The UML **notation for composition is a filled diamond** on an association line, at the composite end of the line

The association name in composition is always implicitly some variation of "Has-part"

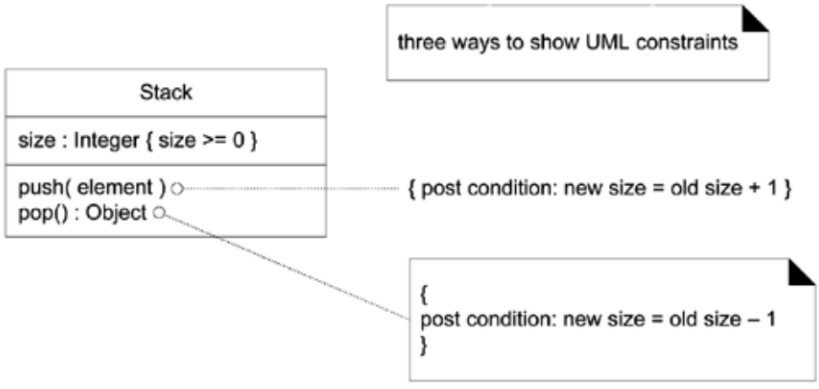


Constraints

A UML constraint is **a restriction or condition** on a UML element. It is visualized in text between braces for example:

```
{ size >= 0 }
```

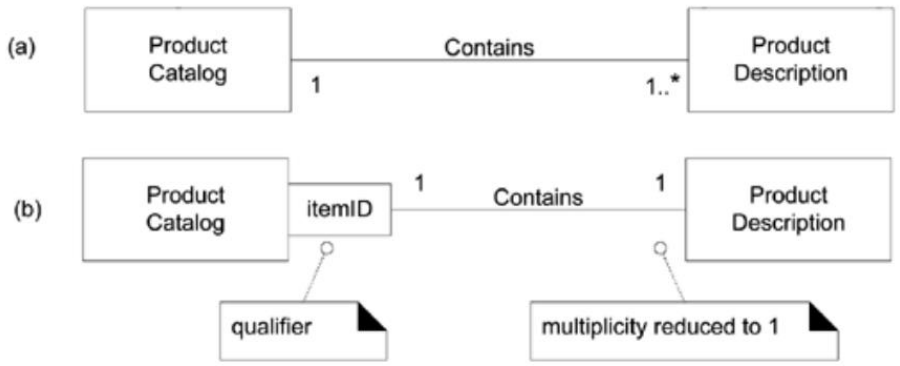
The text may be natural language or anything else, such as UML's formal specification language, the Object Constraint Language (OCL)



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.

Informally, in a software perspective, it suggests looking things up by a key, such as objects in a HashMap. For example, if a ProductCatalog contains many ProductDescriptions , and each one can be selected by an itemID, then the UML notation below can be used to depict this

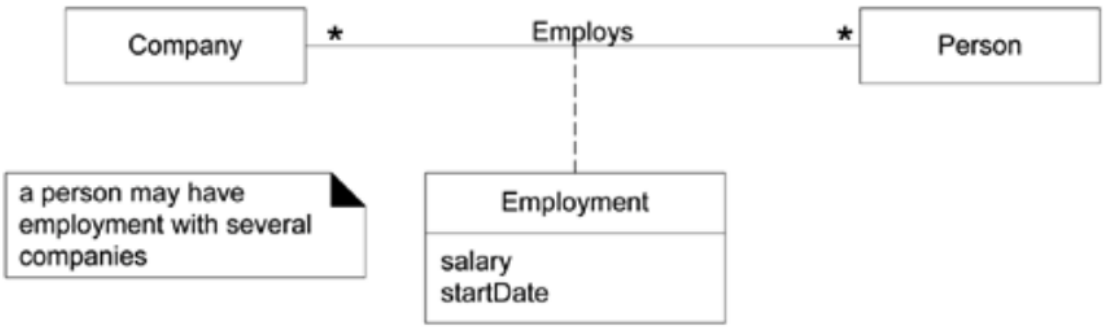


There's one subtle point about qualified associations: the change in multiplicity.

Qualification reduces the multiplicity at the target end of the association, usually down from many to one, because it implies the selection of usually one instance from a larger set.

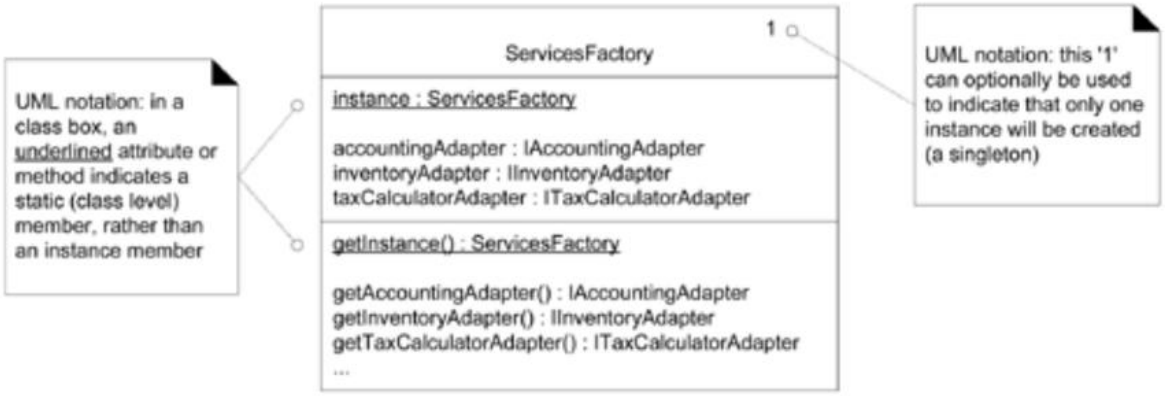
Association Class

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



Singleton Classes

An implication of the Singleton pattern is that **there is only One instance of a class instantiated** never two (a "singleton" instance) In a UML diagram, such a class **can be marked with a '1' in the upper right corner** of the name compartment.



ISSUES REGARDING METHOD NAMES

The following special issues must be considered with respect to method names

- Interpretation of the `create` message
- Depiction of accessing methods
- Interpretation of messages to multiobjects
- Language dependent syntax

i. Method names – Create

The **create message** is the UML language independent form to indicate instantiation and initialization.

In java it implies the invocation of the new operator followed by a constructor call

In DCD this create message is mapped to a constructor definition, using stereotype <<constructor>>

ii. Method Names – Accessing methods

Accessing methods are those **which retrieve or set attributes**.

For example, the `ProductDescription's` price (or `getPrice`) method is not shown, although present, because price is a simple *accessor* method.

iii. Method Names – MultiObjects

A message to multiobject is interpreted as a message to the **container/collection** object itself.

The `find` message is to the container object, not to a `ProductDescription`

A message to multiobject is interpreted as a message to the container/collection object itself.

Thus the `find` method is not a part of the `ProductDescription` class; rather it is part of the hash table or dictionary class definition.

iv. Method names – Language Dependent Syntax

Some languages such as small talk, have a syntactic form for methods that is different from that of the basic UML format of methodName(parameterList). It is recommended that the Basic UML format be used even if the planned implementation language uses a different syntax.

The translation should ideally take place during the code generation time, instead of during the creation of the class diagram.

Adding More Type- Information

The Design Class Diagram should be considered by considering the audience

If it is being in a CASE tool with automatic code generation, full and exhaustive details are necessary.

If it is being created for the software developer to read, exhaustive detail may adversely affect the noise – to –value ratio.

CHAPTER 4

OBJECT ORIENTED IMPLEMENTATION

IMPLEMENTATION MODEL: MAPPING DESIGNS TO CODE

Introduction

Interaction diagrams and DCDs provide sufficient detail to generate code for the domain layer of objects.

The UML artifacts created during the design work, the **interaction diagrams and DCDs,** will be used as input to the code generation process.

The UP defines the Implementation Model that contains the implementation artifacts such as the source code, database definitions, JSP/XML/HTML pages etc.

Programming and the Development Process

There is prototyping or designing **while programming** as well!!

Modern development tools provide an excellent environment to quickly explore alternate approaches, and some (or even lots) **design-while-programming** is usually worthwhile.

Visual modeling before programming is helpful

The creation of code in an object-oriented programming language—such as Java or C#—is **not part of OOA/D**; it is an end goal.

The artifacts created in the UP Design Model provide some of the information necessary to generate the code.

A strength of OOA/D and OO programming—when used with the UP—is that they **provide an end-to-end roadmap** from requirements through to code.

Creativity and Change during Implementation

Some decision-making and creative work, are accomplished during design work.

However, the **programming work is not a trivial code generation step**—quite the opposite.

Realistically, the results generated during design are an incomplete first step; **during programming and testing, numerous changes will be made** and detailed problems will be uncovered and resolved.

The **design artifacts provide a resilient core** that scales up with elegance and robustness to meet the new problems encountered during programming.

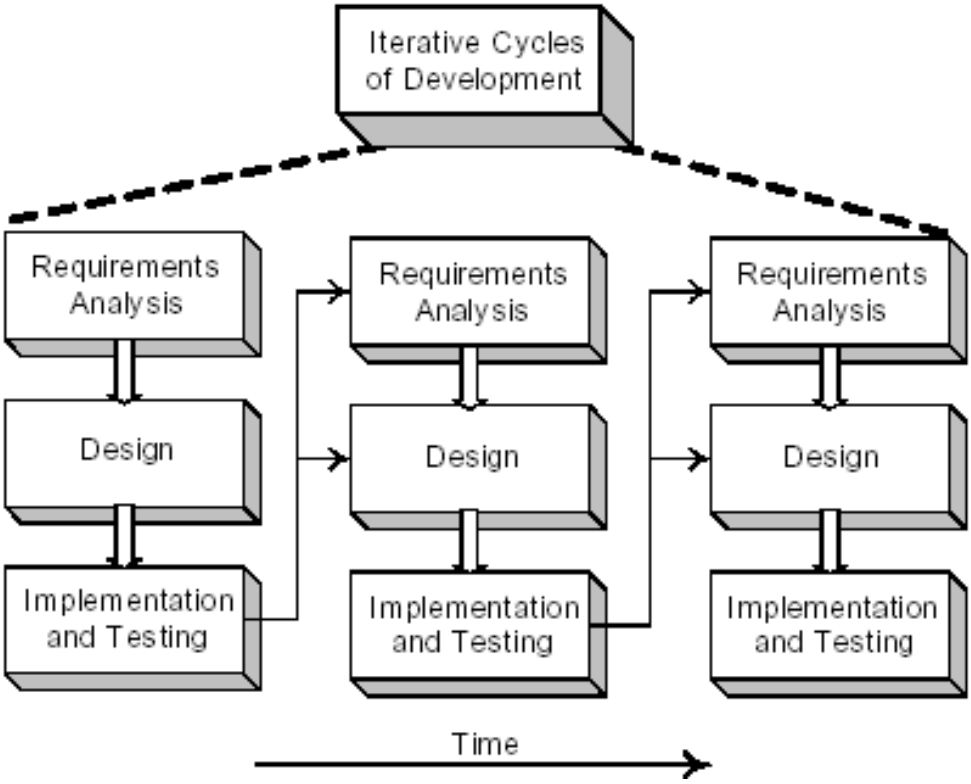
Hence, expect and plan for change and deviation from the design during programming.

Code Changes and the Iterative Process

A strength of an iterative and incremental development process is that the **results of a prior iteration can feed into the beginning of the next iteration**.

Thus, **subsequent analysis and design results are continually being refined and enhanced** from prior implementation work.

For example, when the code in iteration N deviates from the design of iteration N (which it inevitably will), the final design based on the implementation can be input to the analysis and design models of iteration N+l.



An early activity within an iteration is to synchronize the design diagrams; the earlier diagrams of iteration N will not match the final code of iteration N, and they need to be synchronized before being extended with new design results.

Code Changes, CASE Tools, and Reverse-Engineering

It is desirable for the diagrams generated during **design to be semi-automatically updated to reflect changes in the subsequent coding work**.

Ideally this should be done with a CASE tool (like Rational Rose) that can read source code and automatically generate, for example, package, class, and sequence diagrams.

This is an aspect of reverse-engineering—the activity of generating diagrams from source (or sometimes, executable) code.

MAPPING DESIGNS TO CODE

Implementation in an object-oriented programming language requires writing source code for:

- class and interface definitions
- method definitions

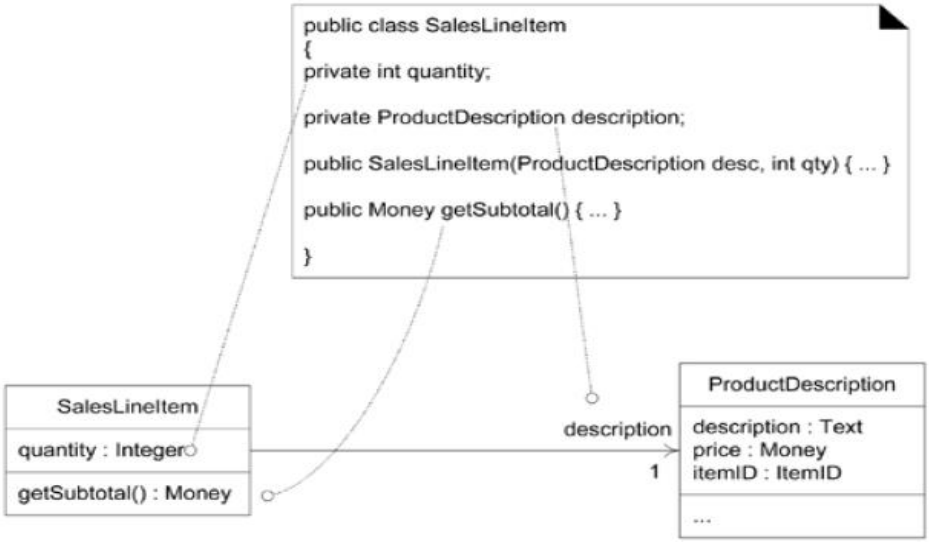
Creating Class Definitions from DCDs

DCDs depict the class or interface name, superclasses, method signatures, and simple attributes of a class.

This is sufficient to create a basic class definition in an object-oriented programming language.

Defining a Class with Methods and Simple Attributes

From the DCD, a mapping to the basic attribute definitions and method signatures for the Java definition of `SalesLineItem` is straightforward

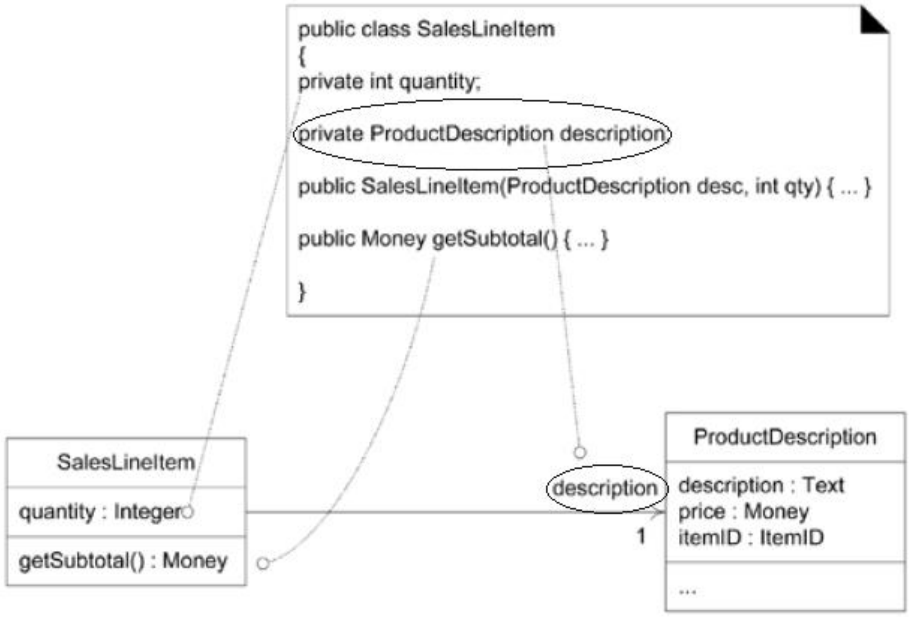


The Java Constructor public `SalesLineItem(ProductDescription desc,int qty)` is created from the `create(desc,qty)` message sent to `SalesLineItem` in the `enterItem` interaction diagram

Adding Reference Attribute

A reference attribute is an attribute that refers to another complex object, **not to a primitive type such as a String, Number**, and so on. (i.e. and object as a attribute of another object)

For example, a `SalesLineItem` has an association to a `ProductDescription`, with navigability to it. This is a reference attribute in class `SalesLineItem` that refers to a `ProductDescription` instance although we have added an instance field to the definition of `SalesLineItem` to point to a `ProductDescription`, it is **not explicitly declared as an attribute in the attribute section of the class box.**



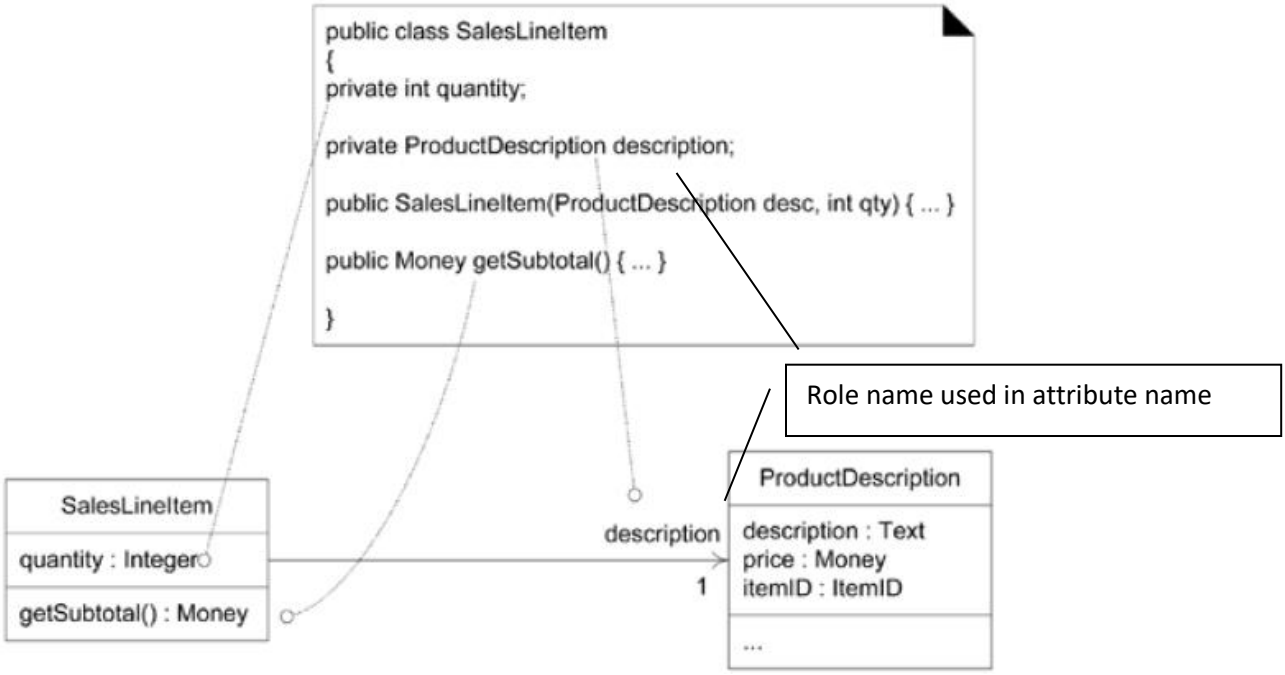
There is a **suggested attribute visibility indicated by the association and navigability (description)** —which is **explicitly defined as an attribute during the code generation phase**. In this case, an object A sees another object B, so that object A can pass message or invoke methods of that object B

Reference Attributes and Role Names

A role name is a name that identifies the role and often provides some semantic context as to the nature of the role.

If a role name is present in a class diagram, use it as the basis for the name of the reference attribute during code generation.

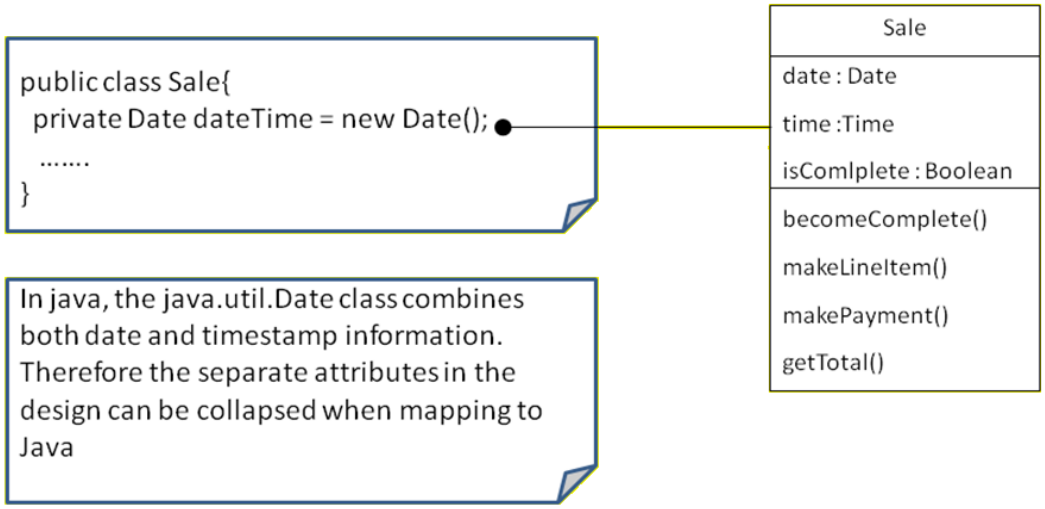
In other words, the role name in the class diagram `description` becomes the name of the reference attribute `description` of the class `SalesLineItem`



Mapping Attributes

In some cases one must consider the mapping of attributes from the design to the code in different languages.

That is , in some languages attributes can be merged into one while mapping from design to code.



For example, a `Date` class in Java combines both data and time information hence while implementing the class a single variable `dateTime` can be used instead of separate attributes date and time

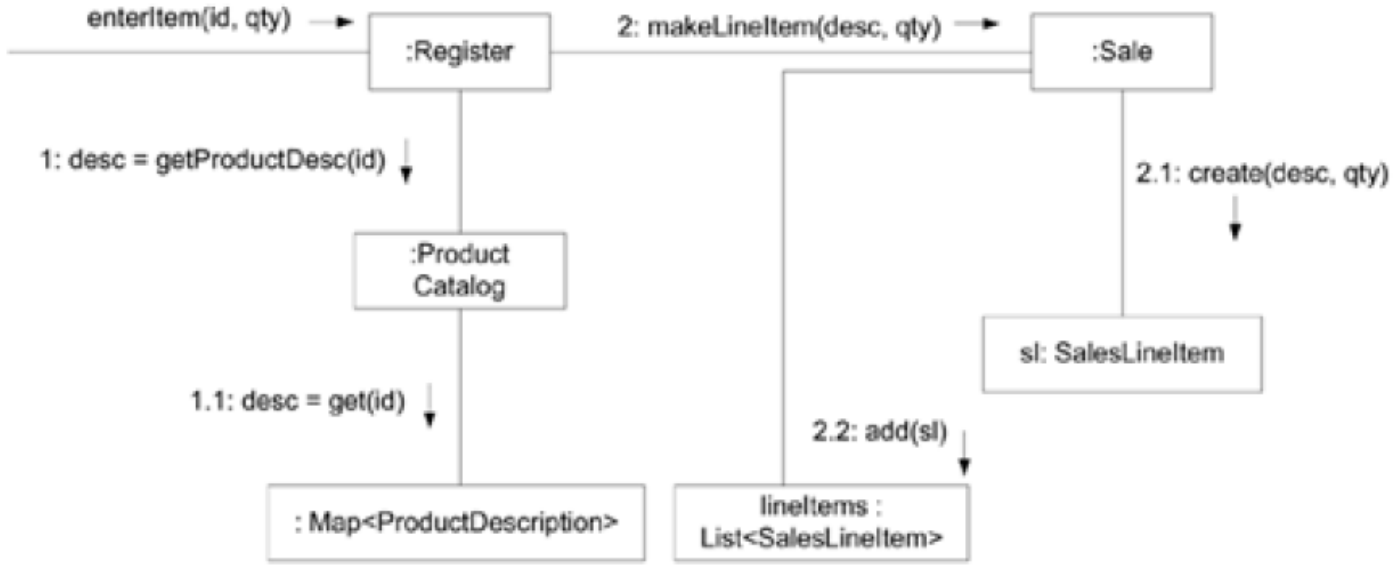
Creating Methods from Interaction Diagrams

An interaction diagram shows the messages that are sent in response to a method invocation.

The sequence of these messages translates to a series of statements in the method definition.

Consider the `enterItem` communication diagram

The `enterItem` interaction diagram illustrates the Java definition of the `enterItem` method.



The `enterItem` message is sent to a Register instance so the enterItem method is defined in the Register class

```
public void enterItem(ItemID itemID, int qty)
```

Message 1: A `getProductDescription` message is sent to the `ProductCatalog` to retrieve `ProductDescription`

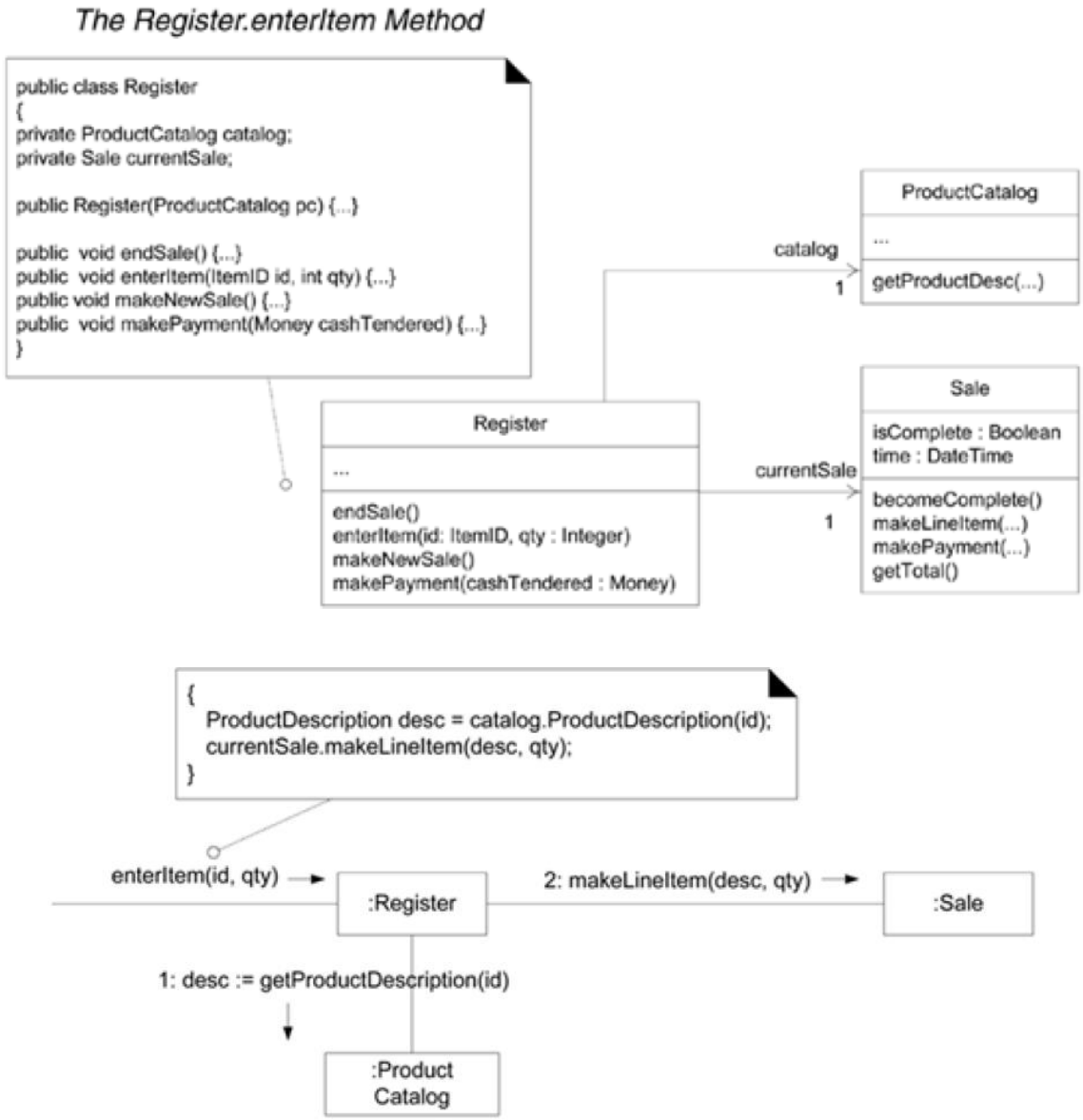
```
ProductDescription desc = catalog.getProductDescription(itemID);
```

Message 2: A `makeLineItem` message is sent to the `Sale`

```
currentSale.makeLineItem (desc,qty);
```

So it takes the form:

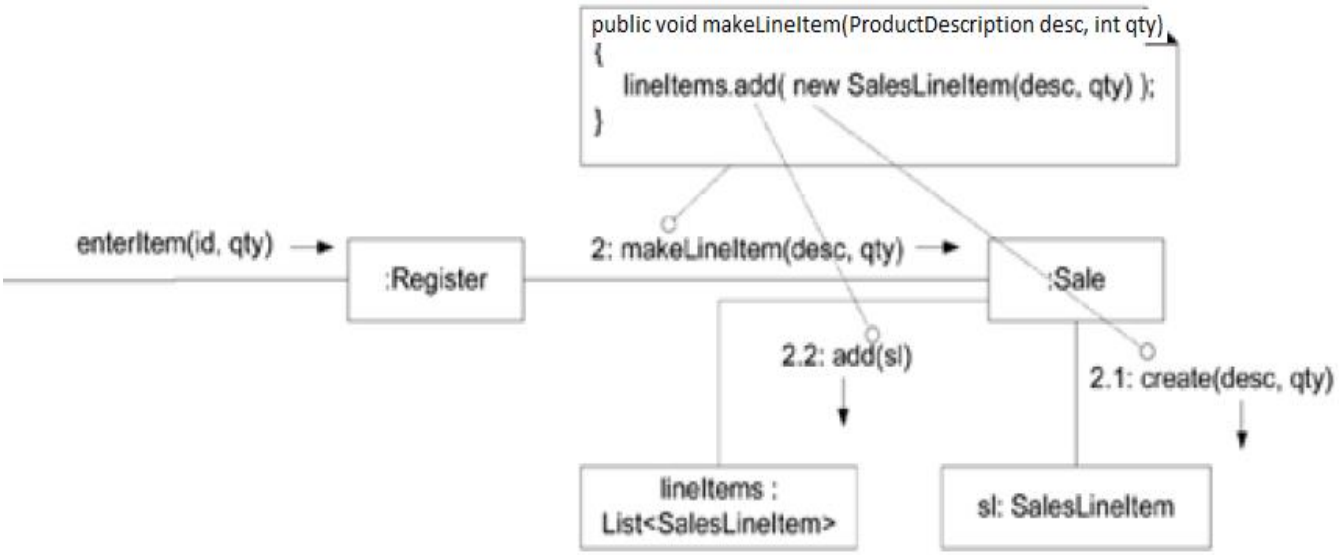
```
public void enterItem(ItemID itemID, int qty){
   ProductDescription desc = catalog.getProductDescription(itemID);
   currentSale.makeLineItem (desc,qty);
}
```



Defining the Sale—`makeLineItem` Method (`sale.makeLineItem`)

As a final example, the `makeLineItem` method of class `Sale` can also be written by inspecting the `enterItem` communication diagram.

Here, in response to the method invocation (`makeLineItem(----)`), the **sale object creates a `saleLineObject`** and **adds that item**, which is done in a single method invocation **`makeItems.add (new SalesLineItem(spec,qty))`**



EXCEPTIONS IN THE UML

In the UML, an Exception is a specialization of a Signal, which is the specification of an

asynchronous communication between objects. This means that in interaction diagrams,

Exceptions are illustrated as asynchronous messages.

The UML has default syntax for operations but it doesn't include an official solution to show exceptions thrown by an operation

Three solutions:

1. The UML allows the operation syntax to be any other language, such as Java, in addition, some UML CASE tools allow display of operations explicitly in Java syntax :

```
object get(key, class) throws DBUnavailableException, FatalException
```

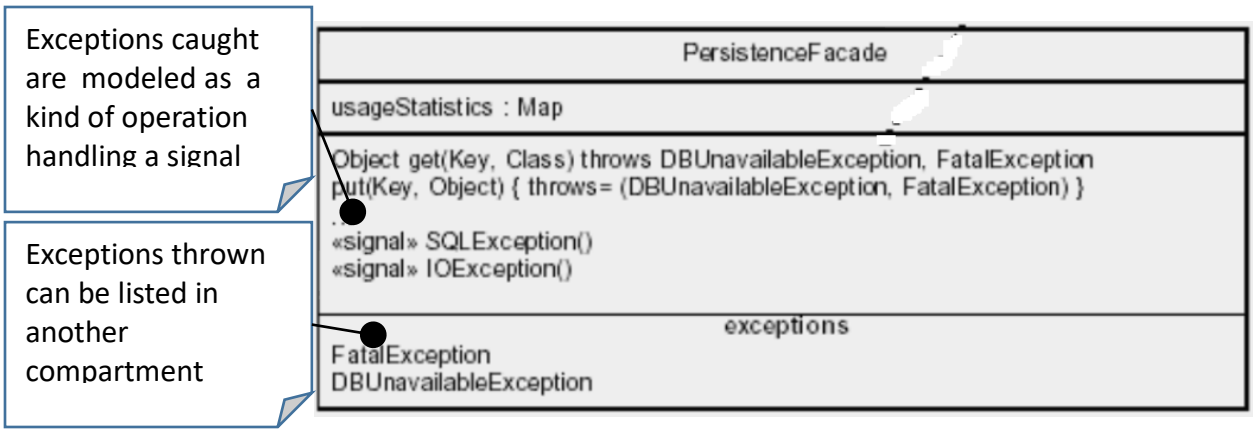
2. The default syntax allows the last element to be a "property string" this is a list of arbitrary property + value pairs such as

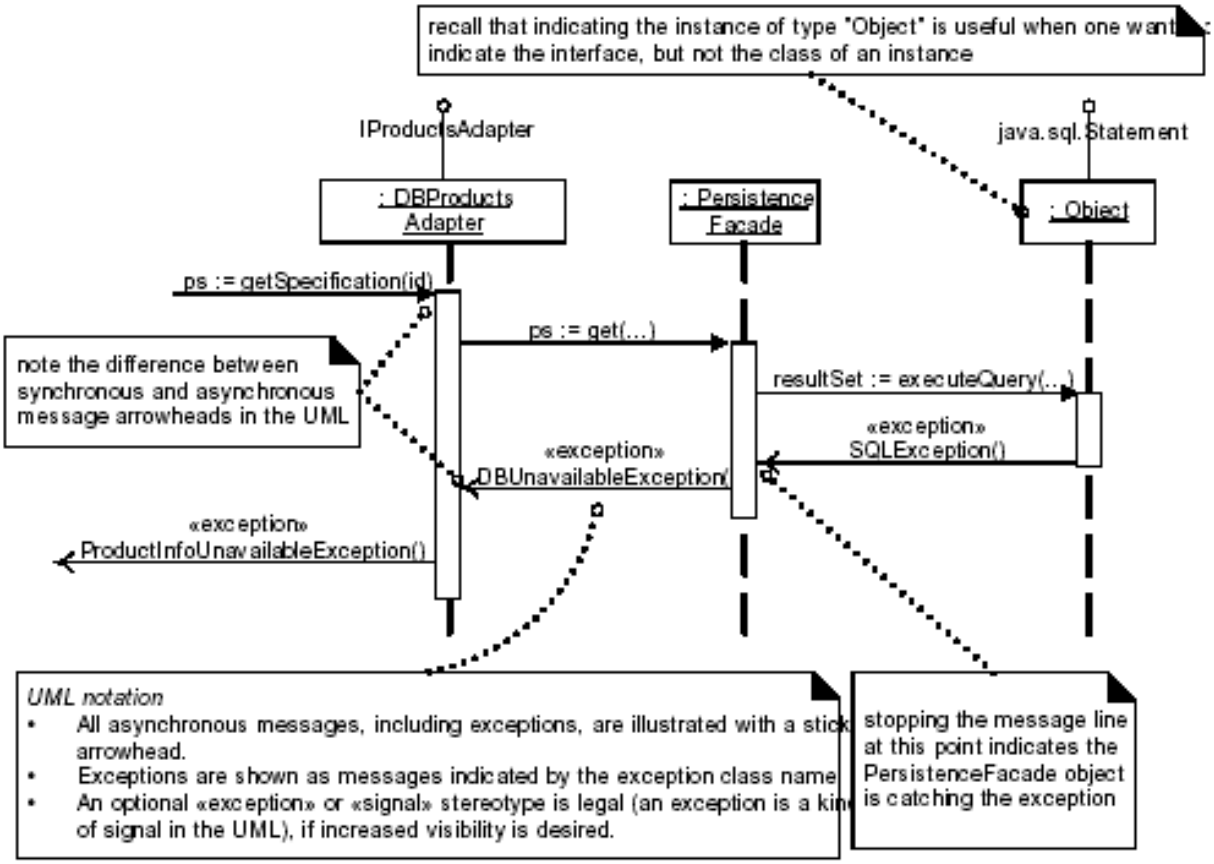
```
{ author = Craig, kids =(Hannah, Haley) }
```

thus

```
put(Object id){throws = (DBUnavailableException, FatalException)}
```

3. Some UML CASE tools allow one to specify (in a special dialbox) the exceptions that an operation throws


Exceptions caught are modeled as a kind of operation handling a signal
Exceptions thrown can be listed in another compartment
PersistenceFacade
usageStatistics : Map
Object get(Key, Class) throws DBUnavailableException, FatalException
put(Key, Object) { throws= (DBUnavailableException, FatalException) }
«signal» SQLException()
«signal» IOException()
exceptions
FatalException
DBUnavailableException


recall that indicating the instance of type "Object" is useful when one want
indicate the interface, but not the class of an instance
IProductsAdapter
java.sql.Statement
: DBProducts Adapter
: Persistence Facade
: Object
ps := getSpecification(id)
ps := get(...)
resultSet := executeQuery(...)
note the difference between synchronous and asynchronous message arrowheads in the UML
«exception» SQLException()
«exception» DBUnavailableException()
«exception» ProductInfoUnavailableException()
UML notation
All asynchronous messages, including exceptions, are illustrated with a stick arrowhead.
Exceptions are shown as messages indicated by the exception class name.
An optional «exception» or «signal» stereotype is legal (an exception is a kind of signal in the UML), if increased visibility is desired.
stopping the message line at this point indicates the PersistenceFacade object is catching the exception