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.

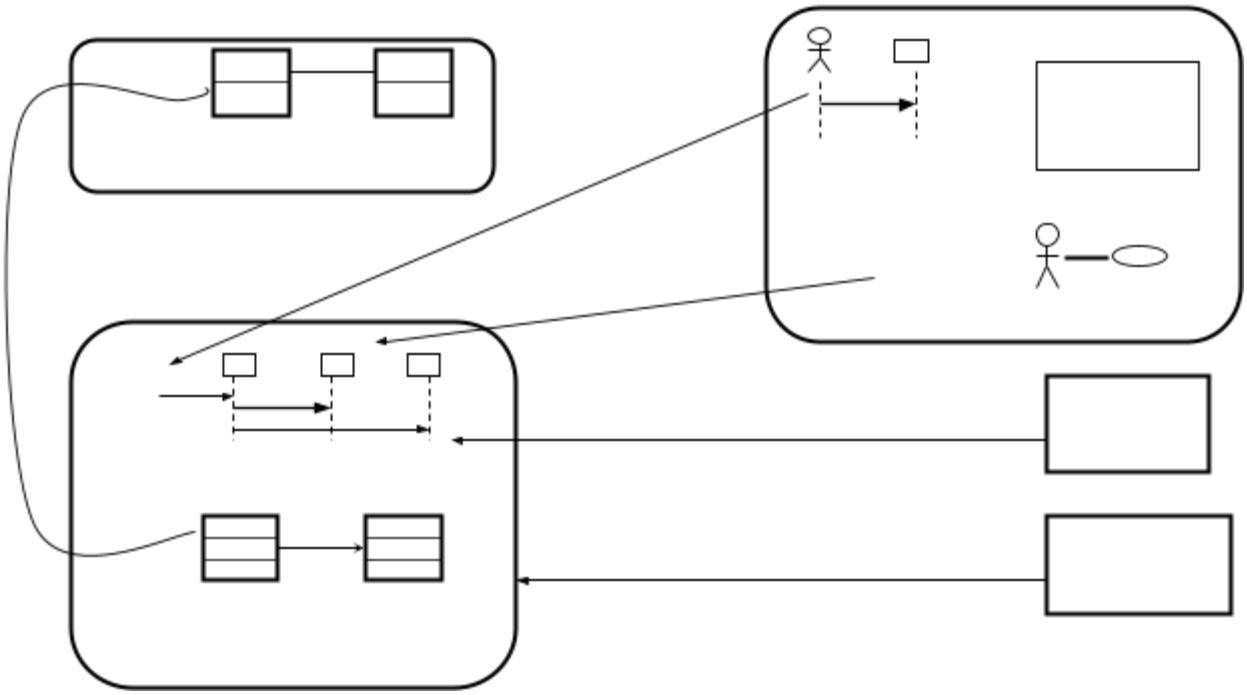
After or in parallel with drawing interaction diagrams, (design) **class diagrams** can be drawn. These summarize the definition of the software classes (and interfaces) that are to be implemented in software.

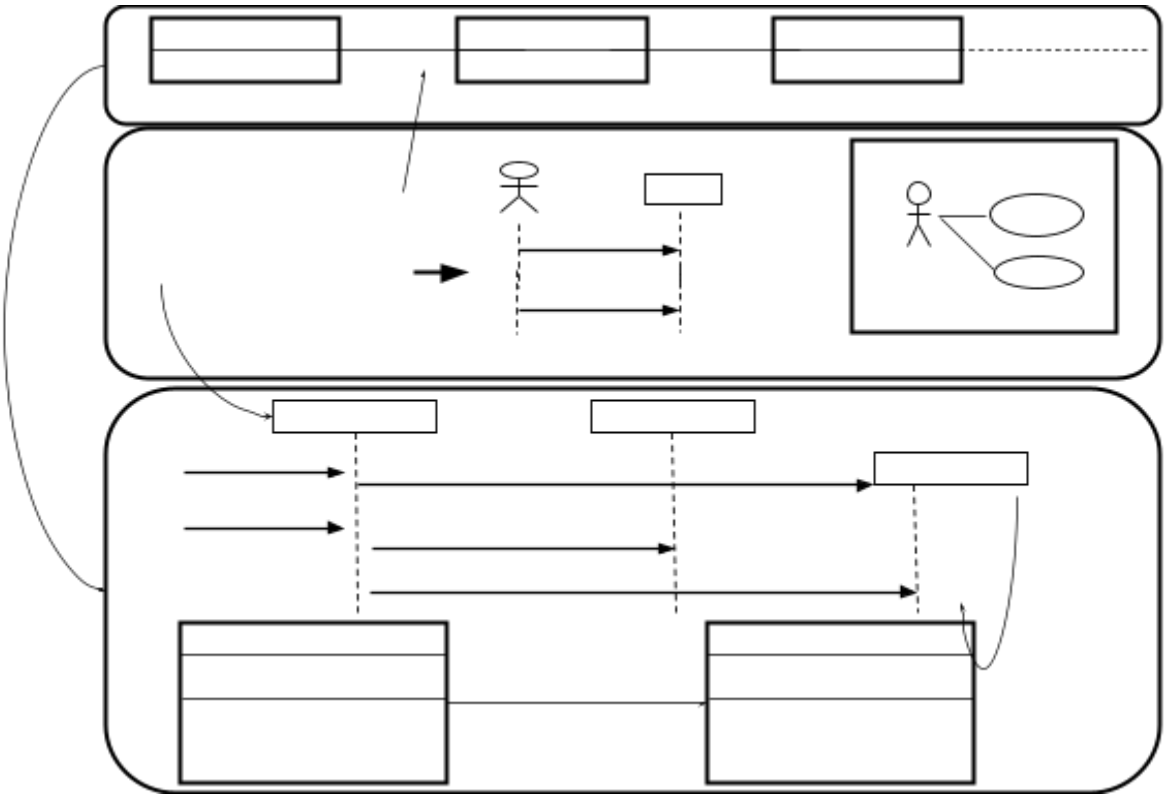
In terms of the UP, these artifacts are part of the **Design Model.**

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





During early phases requirements should focus on the **user intent only**, the UI details are avoided. The user intents are captured **using essential use cases**.

OOA (Essential Use cases) What is required?

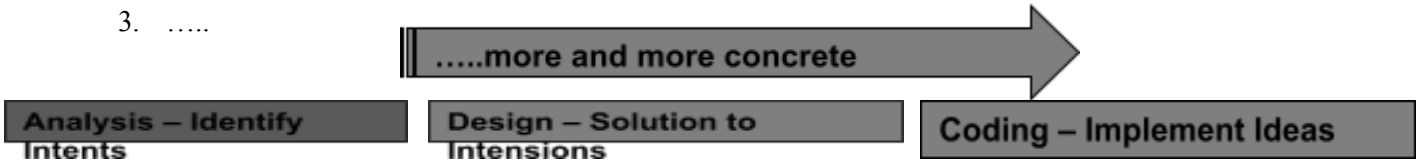
....

1. Administrator **identifies** itself
2. System **authenticates** identity
3. .....

During the design phase the **design solution to these intentions** and responsibilities that are wide open are identified.

OOD (Real/Concrete Use cases) How is it fulfilled?

1. Administrator **enters ID and Password** in dialog box as in figure 2
2. System Authenticates administrator **using Biometric readers**
3. …..



DESCRIBING CONCRETE/REAL USE CASES

Concrete/Real Use Cases **concretely describe the process** in terms of its rare current design committed to **specific input and output technologies** and so on.

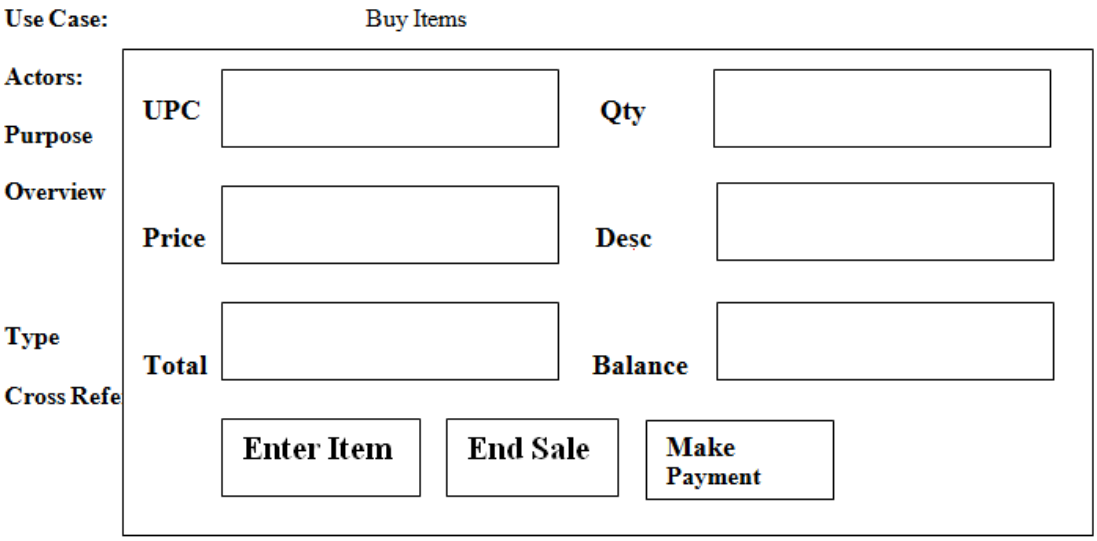
Withdraw cash example (Concrete/Real):

Actor action	**System response**
1. The customer inserts their cards.	2. Prompts for **PIN**.
3. **Enters PIN** on keypad.	4. Displays option menu
5. and so on…	

The real use cases are created **during the design phase** of a development cycle, since they **are the design artifacts**.

If a graphical user interface is involved, the real use case will **include diagrams of the windows involved,** and **discussion of the low level interaction with the interface widgets**

Concrete/Real Buy Item use case:



Real vs Essential 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

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

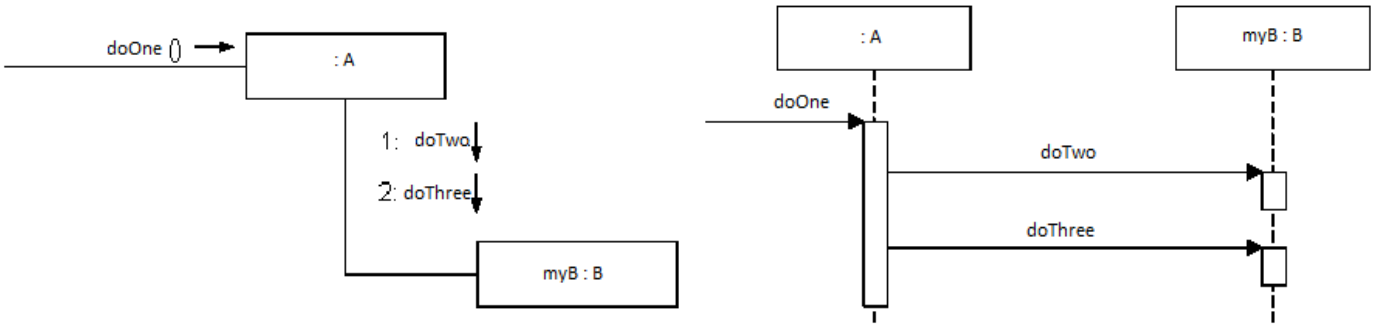
The two common types of interaction diagrams in the UML are

- 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**



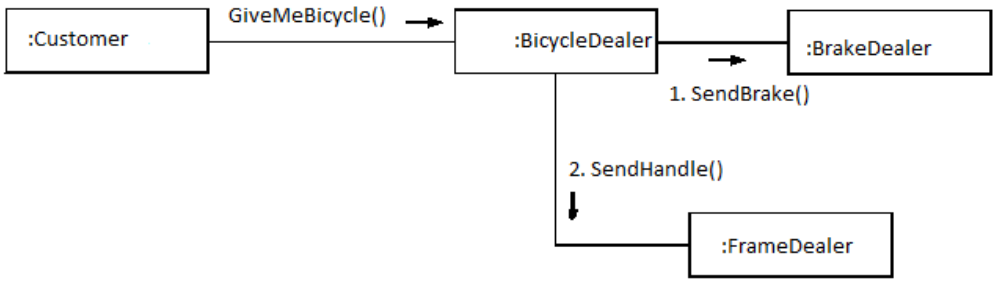
Related Code:

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

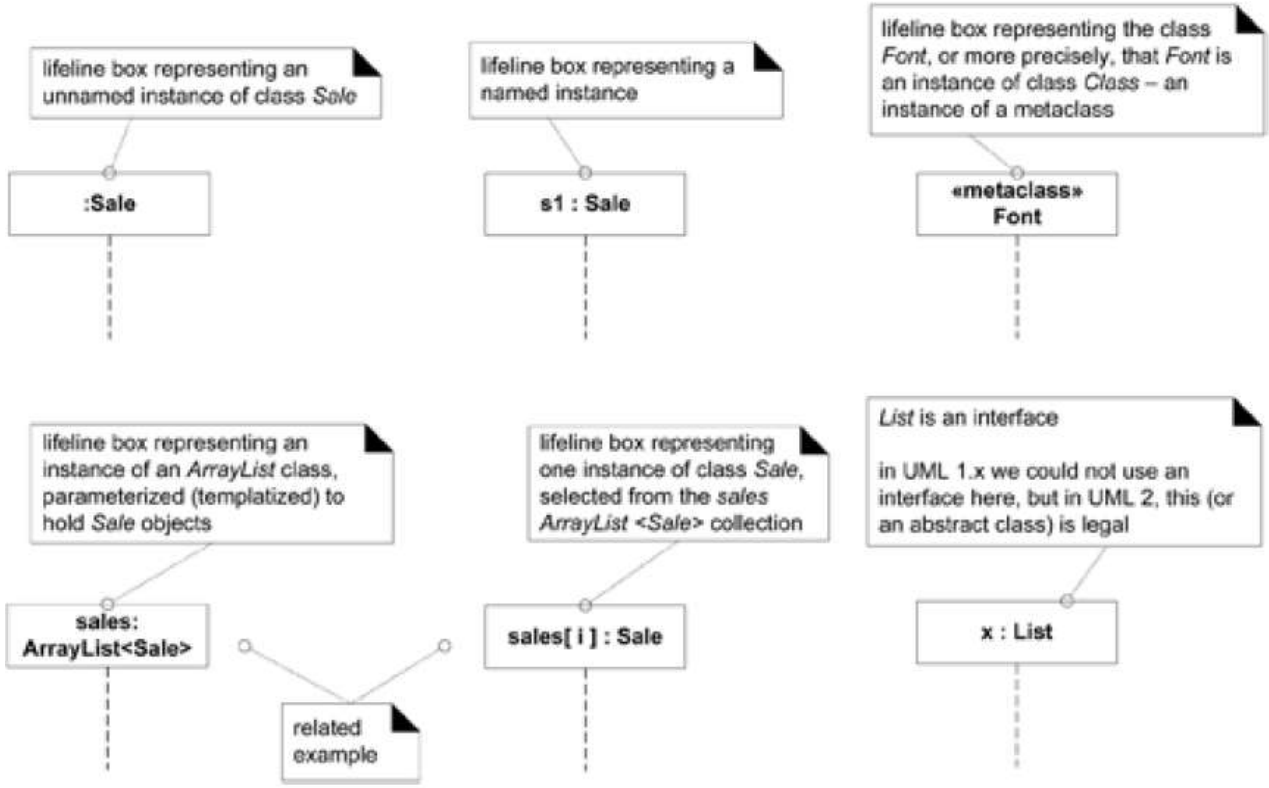
Type	Strength	Weaknesses
Sequence	Clearly shows sequence or time ordering of messages (call flow sequence) Simple but large set of detailed notation options Excellent for documentation	Forced to extend to the right when adding new objects: consumes horizontal space
Communication	Space economical-flexibility 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

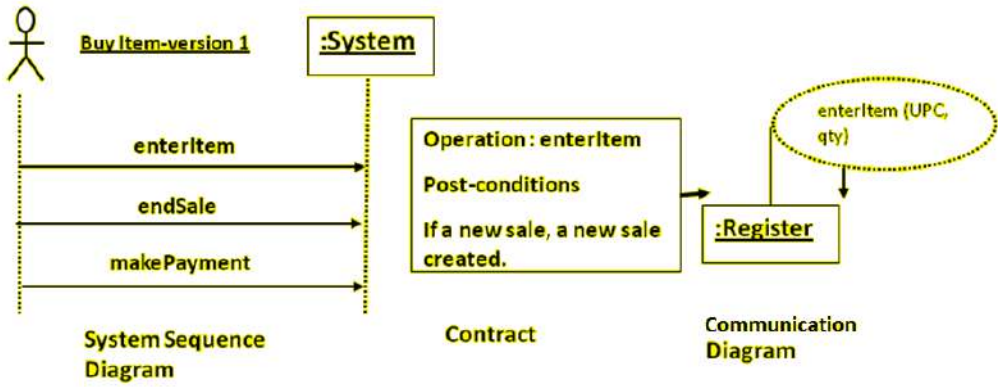
Making Communication Diagrams

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

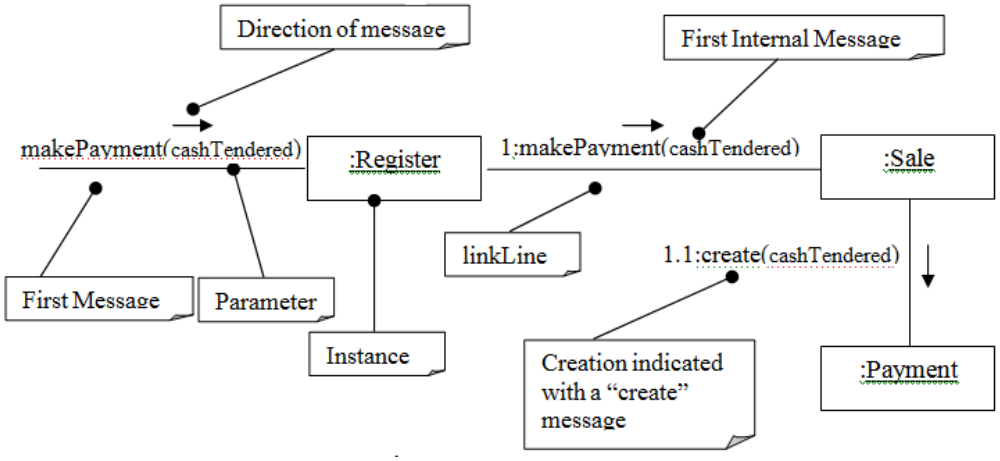
If the diagram gets complex **split it** into smaller diagrams

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

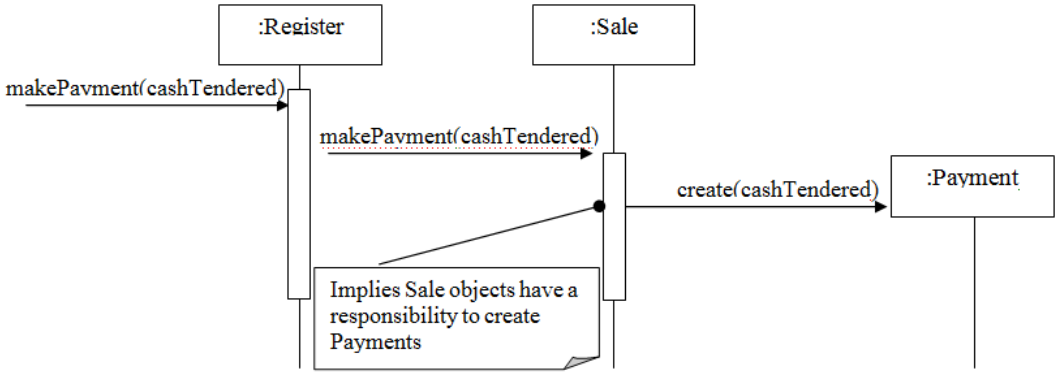


Example Communication Diagram: makePayment

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);
          --------
     }
}
```

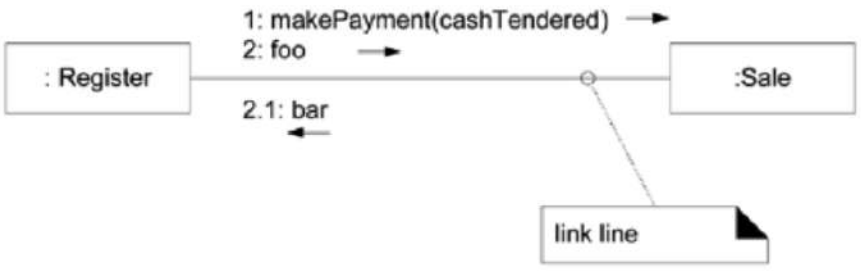
Links

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.

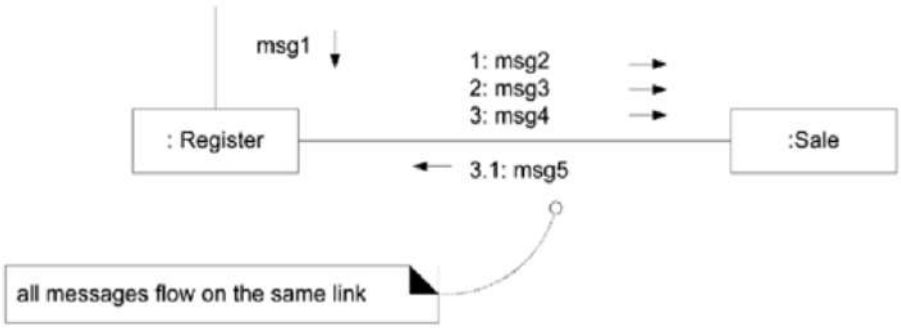


Messages

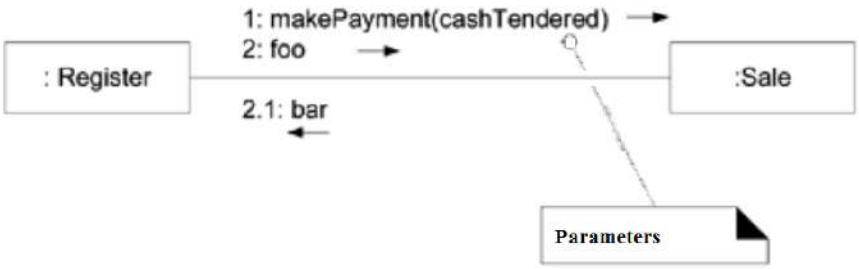
Each message between objects is represented with a **message expression** and **small arrow** indicating the direction of the message.

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.



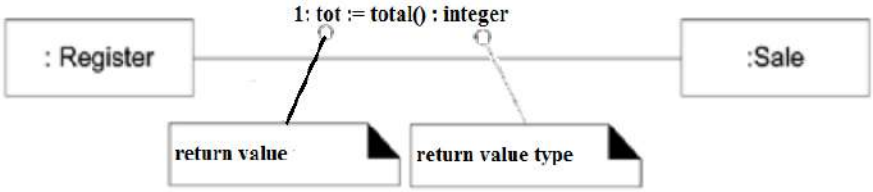
Illustrating Parameters



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

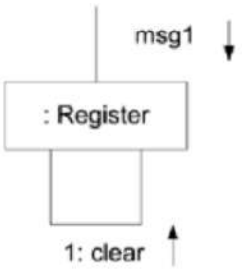


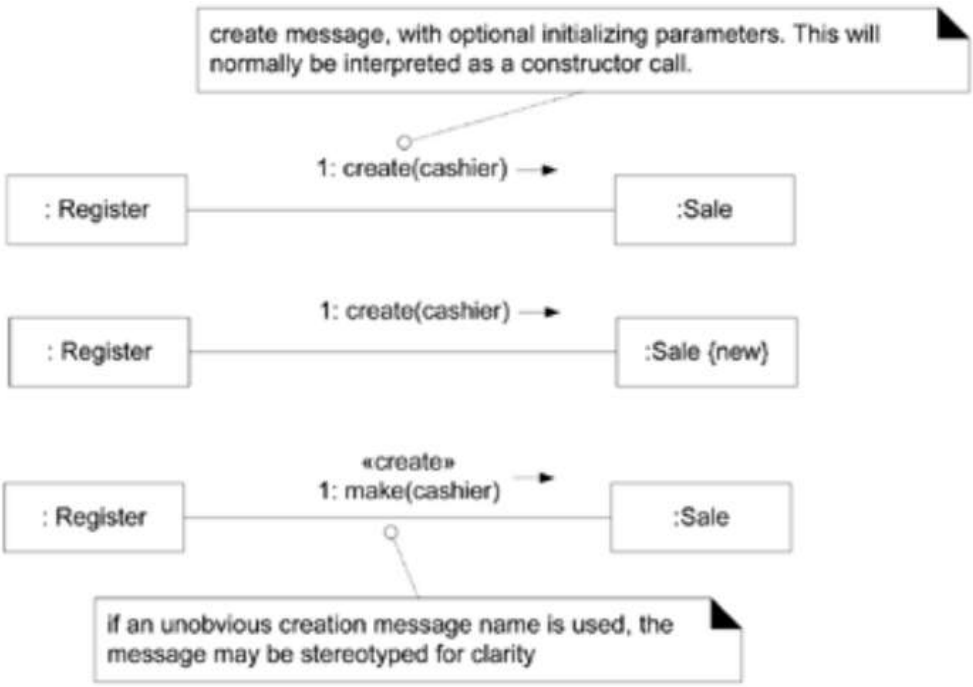
Illustration of creation of instances

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*

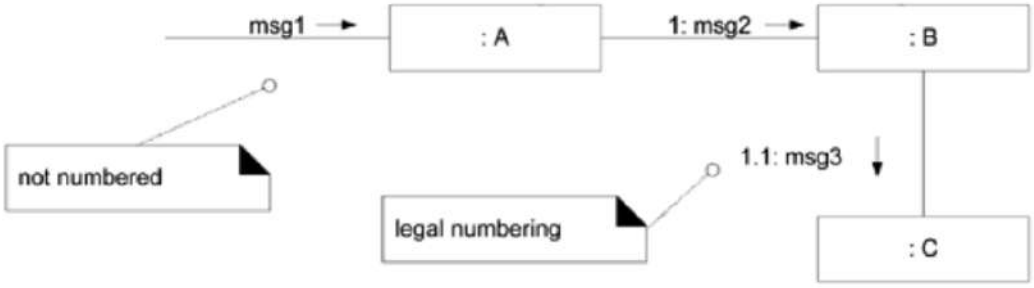


Illustrating Message Number Sequencing

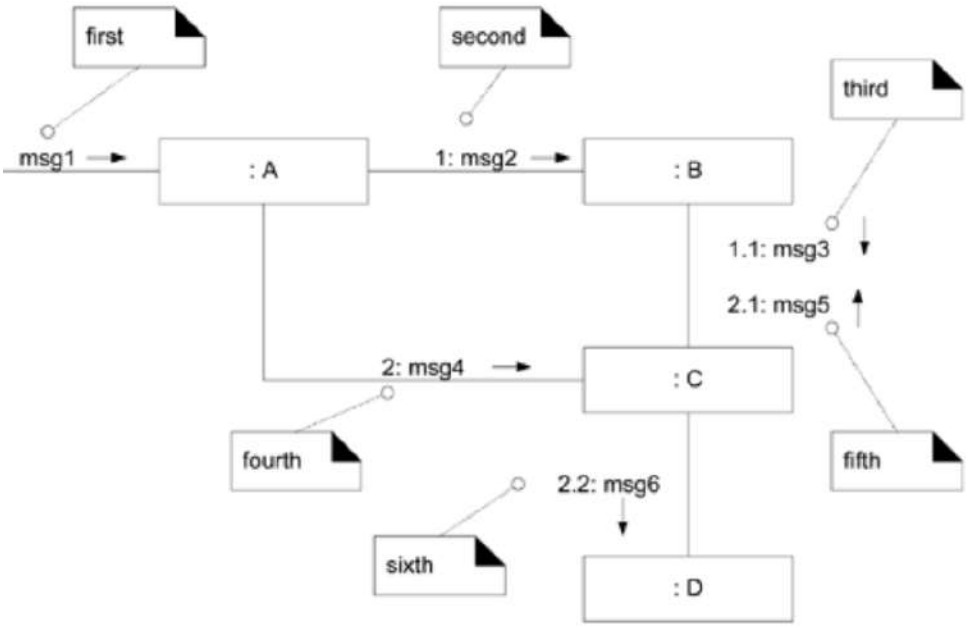
The order of message is illustrated with the sequence numbers

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.



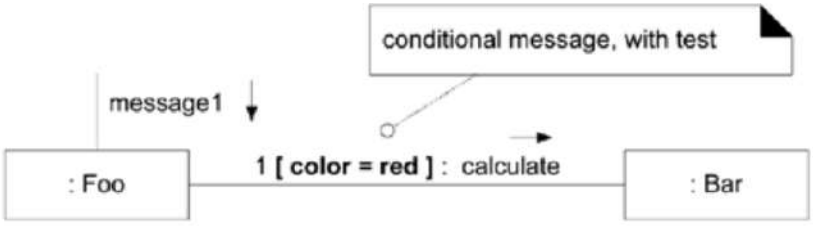
Nesting is denoted by prepending the incoming message number to the outgoing message number



Illustrating Conditional Message

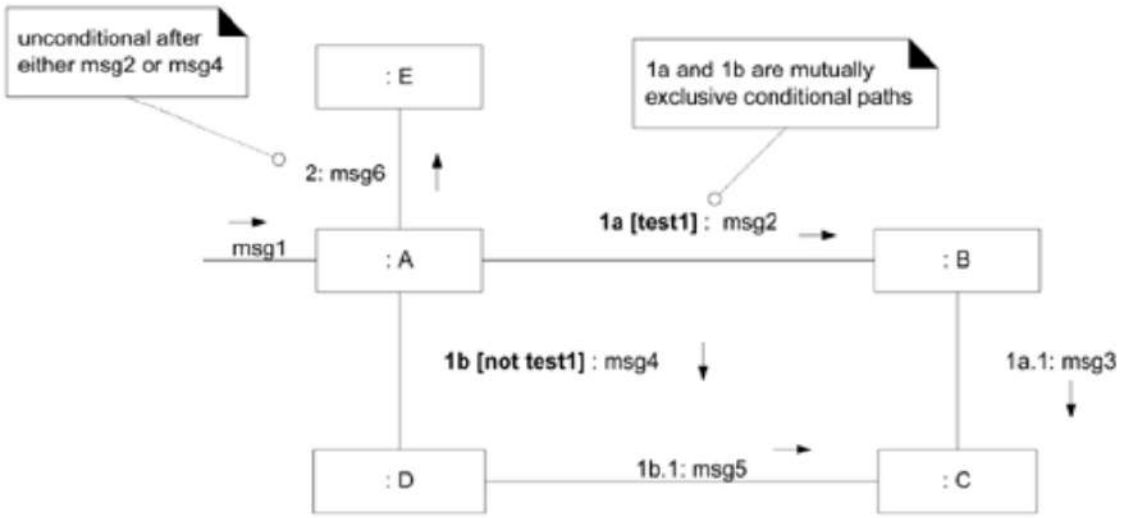
A conditional message is shown by f**ollowing a sequence number with a conditional clause in square brackets**, similar to an iteration clause.

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



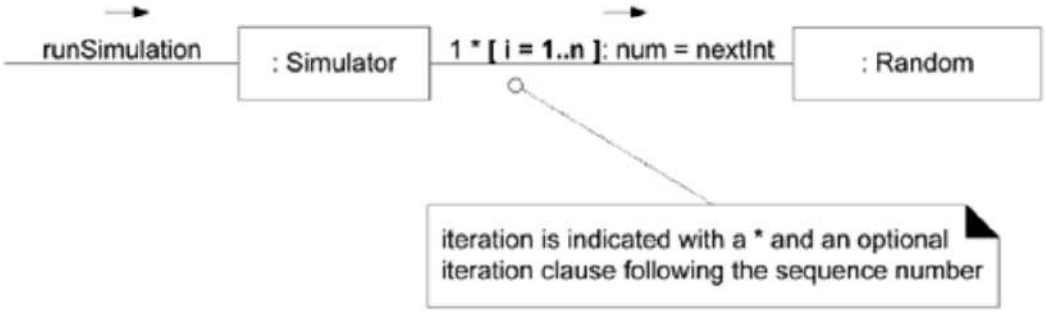
Illustrating Mutually Exclusive Conditional Paths

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



Here either 1a or 1b could execute after msg1

Illustrating iteration



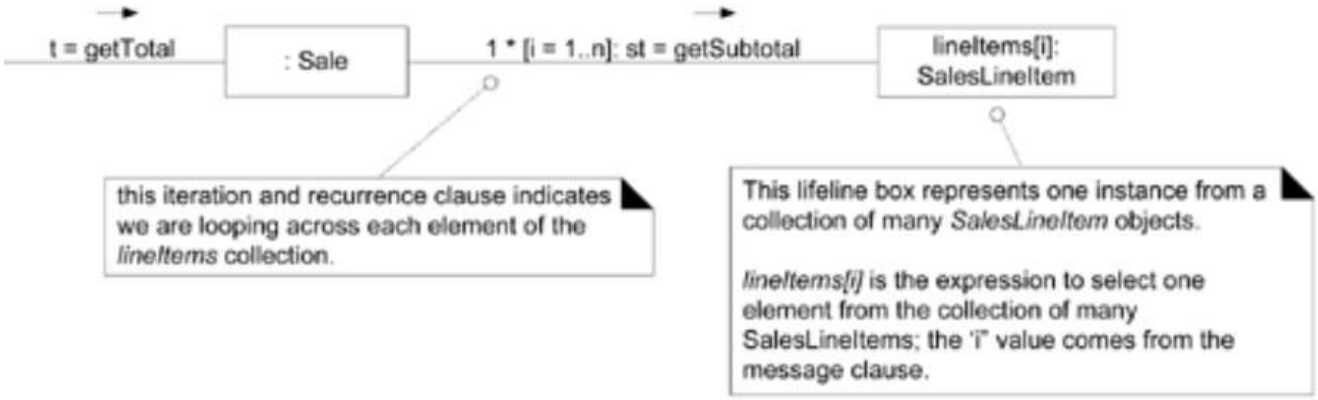
Iteration is indicated by following the sequence number with a star ('*')

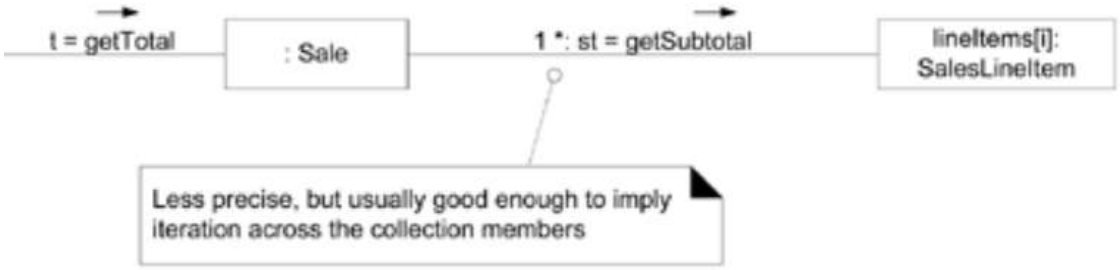
This expresses that the message is being sent repeatedly, in a loop to the receiver

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.

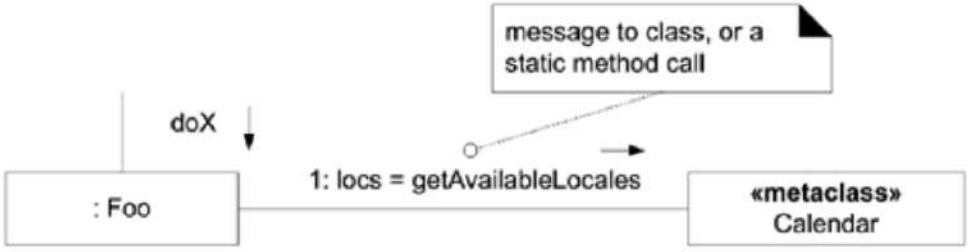
In the UML, the term **multi object** is used to denote a set of instances .a collection.

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

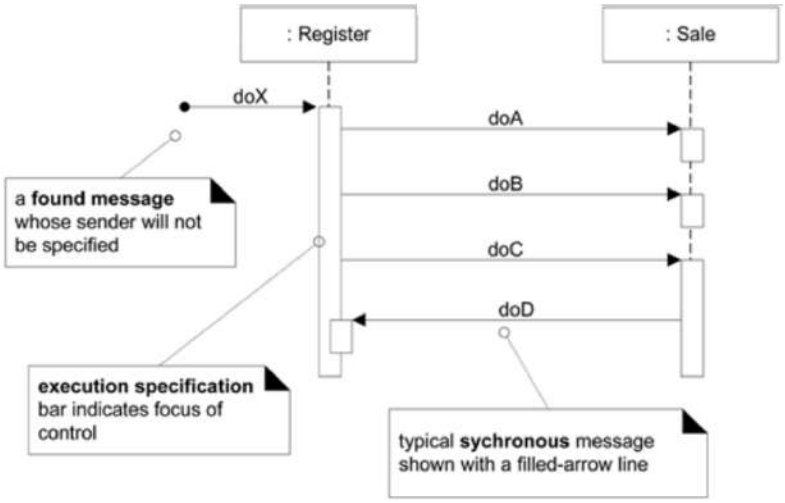
In contrast to communication diagrams, in sequence diagrams the lifeline boxes include a vertical line extending below them, these are the actual lifelines.

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

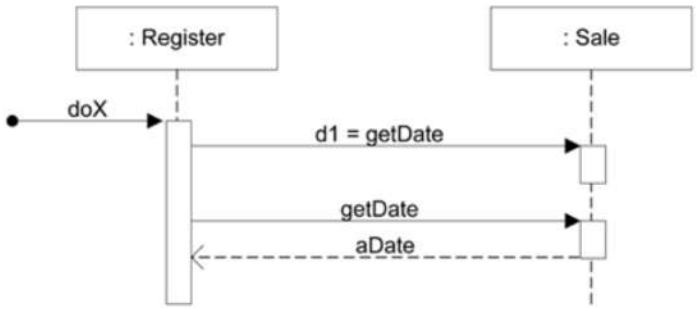
Sequence diagrams may also show the **focus of control** (informally, in a regular blocking call, the operation is on the call stack) using an **execution specification bar** (previously called an activation bar or simply an activation in UML 1). The bar is optional.

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

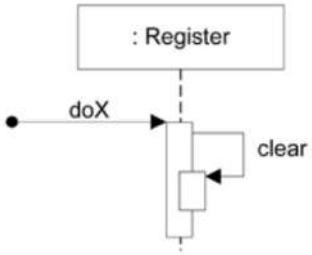
There are two ways to show the return result from a message:

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



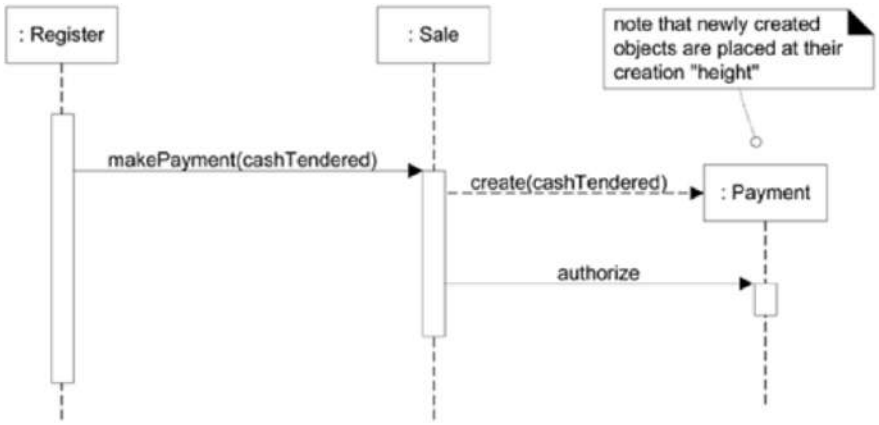
Messages to "self" or "this"

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

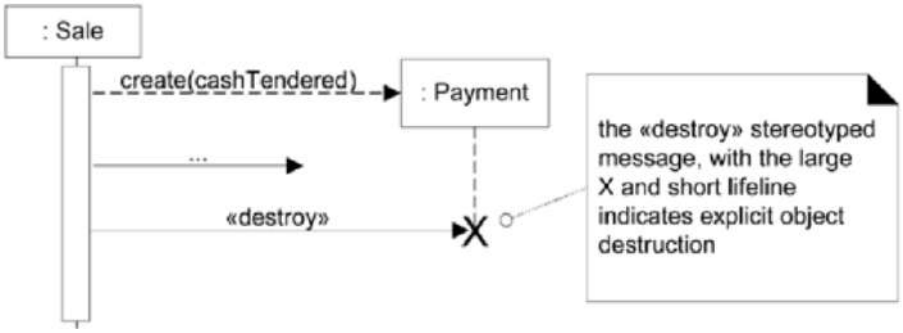


Diagram Frames in UML Sequence Diagrams

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:

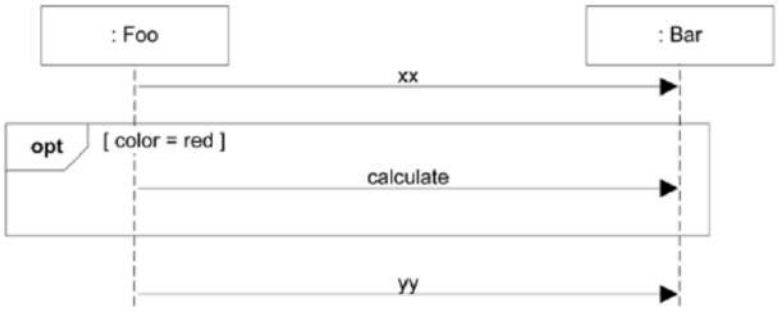
Frame Operator	Meaning
alt	Alternative fragment for mutual exclusion conditional logic expressed in the guards.
loop	Loop fragment while guard is true. Can also write *loop(n)* to indicate looping n times. There is discussion that the specification will be enhanced to define a *FOR* loop, such as *loop(i, 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

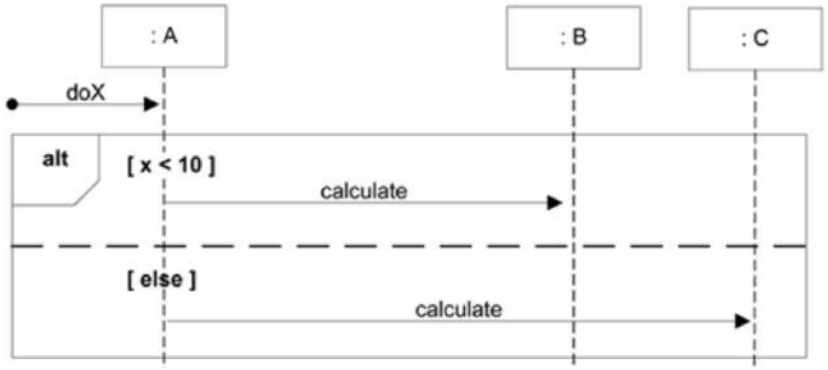
Conditional Messages

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



Mutually Exclusive Conditional Messages

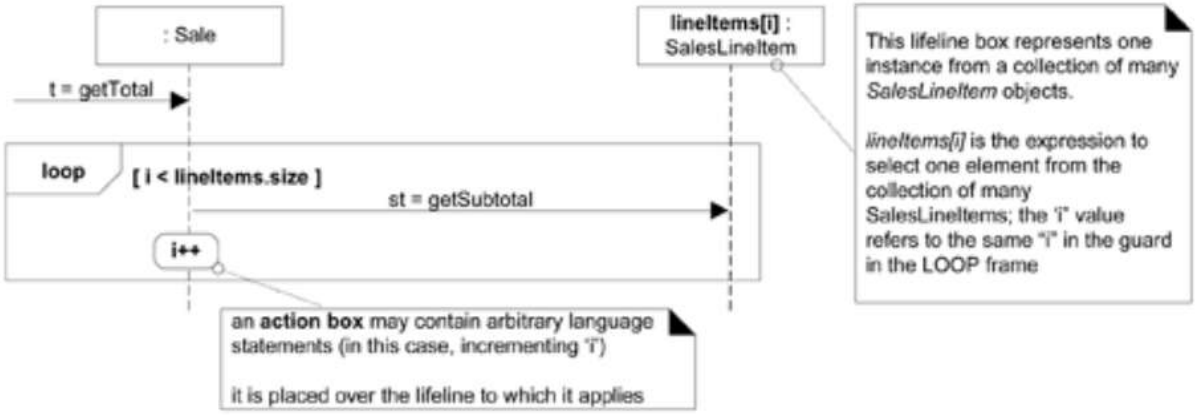
An **ALT frame** is placed around the mutually exclusive alternatives



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

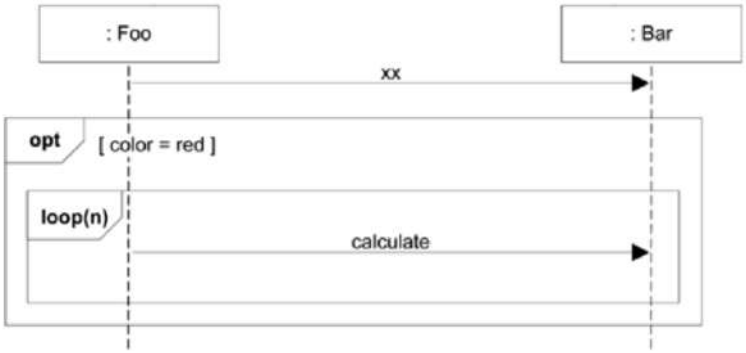
The selector expression is used to select one object from a group. Lifeline participants should represent one object, not a collection.

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;
     }
     // ...
}
```

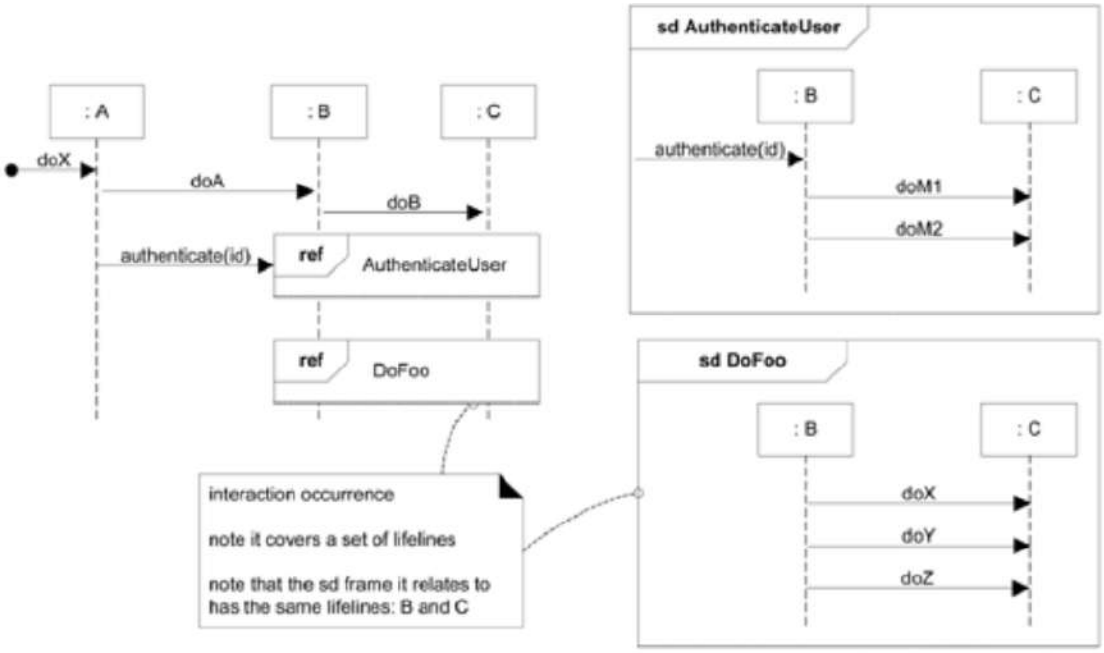
Nesting of Frames

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:

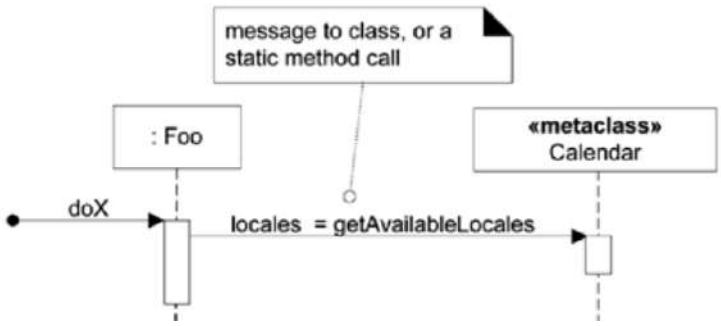
- a frame around an entire sequence diagram, labeled with the tag **sd** and a **name**, such as AuthenticateUser
- a frame tagged **ref** , called a reference , that **refers to another named sequence diagram**; it is the actual interaction occurrence

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.

Messages to Classes to Invoke Static (or Class) Methods

Class or static method calls can be shown by using a lifeline box label that indicates the receiving object is a class, or more precisely, an instance of a metaclass



In Java and Smalltalk, all classes are conceptually or literally instances of class Class ; in .NET classes are instances of class Type . The classes Class and Type are Metaclasses , which means their instances are themselves classes. A specific class, such as class Calendar , is itself an instance of class Class.

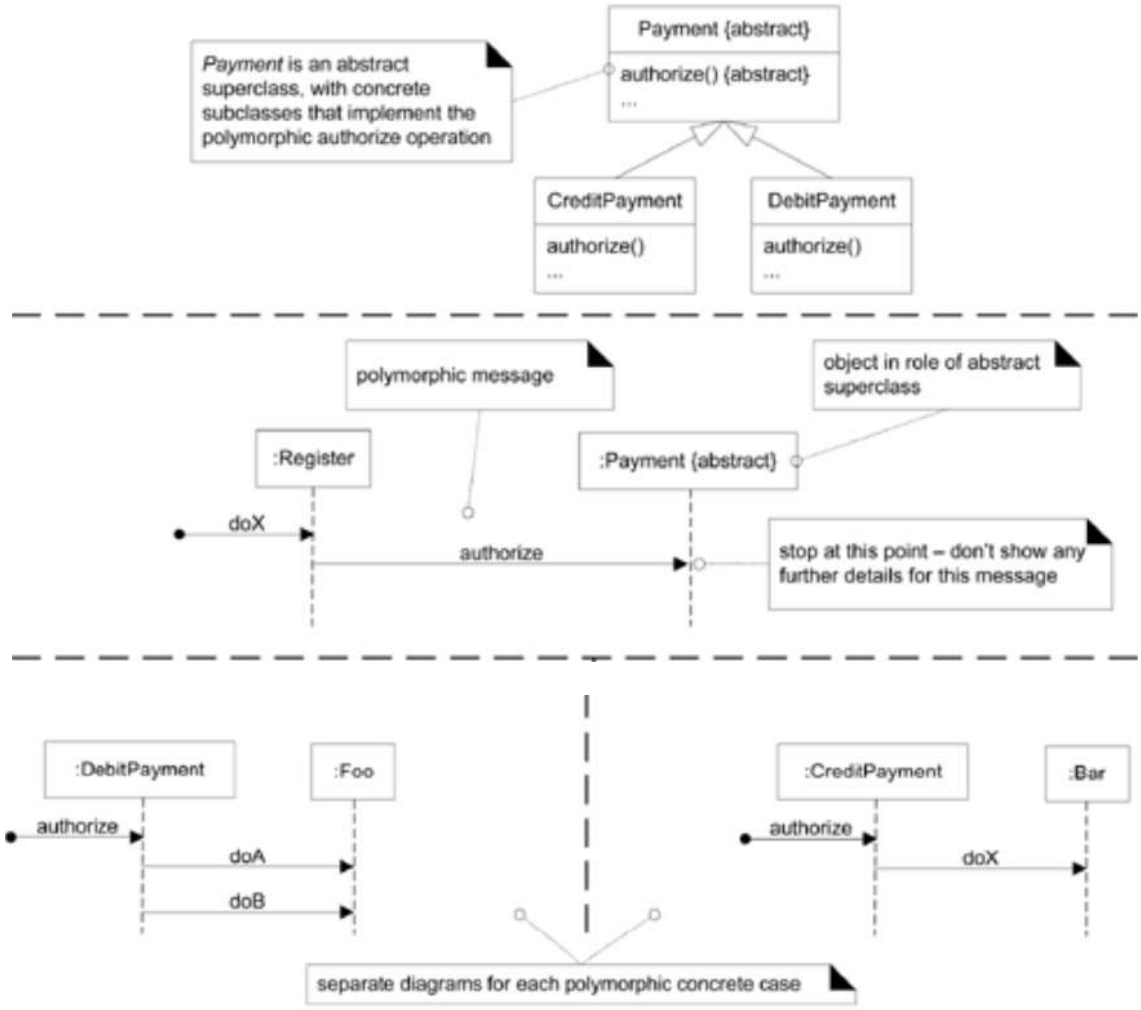
Thus, class Calendar is an instance of a metaclass! In code, a likely implementation is:

```
public class Foo
{
     public void doX()
     {
          // static method call on class Calendar
          Locale[] locales = Calendar.getAvailableLocales();
          // ...
     }
     // ...
}
```

Polymorphic Messages and Cases

Polymorphism is fundamental to OO design.

One approach is to use multiple sequence diagrams one that shows the polymorphic message to the abstract super class or interface object, and then separate sequence diagrams detailing each polymorphic case, each starting with a found polymorphic message.



Designing a Solution with Objects and Patterns

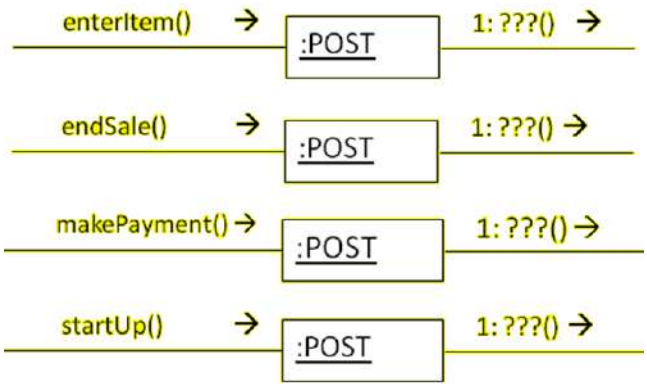
In the current scenario of POST , let us consider two use cases and their associated system events

- `Buy Items`
    - `enterItem()`
    - `endSale()`
    - `makePayment()`
- `Start Up`
    - `startup()`

At least four interaction diagrams, one for each system event, are required.

The POST can be chosen as the controller for handling the events

Therefore there will be at least four interaction diagrams, one for each system event



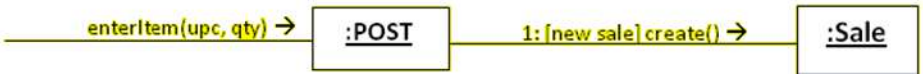
For each system event a **contract exists which elaborates on** what the operation must achieve

Using the contract responsibility and post conditions, and **use cases** as a starting point, **design a system of interacting objects** to fulfill the task

Operation contracts complements the use case text to clarify what the software objects must achieve in a system operation and the post conditions describe the detailed achievements

Contract for `enterItem`

Name:	**enterItems (upc: number, quanity: integer)**
Responsibility:	**Enter (record) sale of an item and add it to the sale. Display the item description and price.**
Type	**System, System Functions: R1.1, R1.3, R1.9**
Cross References:	**Use Cases: Buy Items**
Exceptions:	**If UPC is not valid, indicate that it was an error.**
Pre-conditions:	
Post-condition:	**•a new sale, a sale was created (instance)** **•a new sale, the new sale was associated with the POST (association formed)** **•A SaleLineItem was created (instance creation)** **•The SaleLineItem was associated with the Sale. (association formed)**



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.**

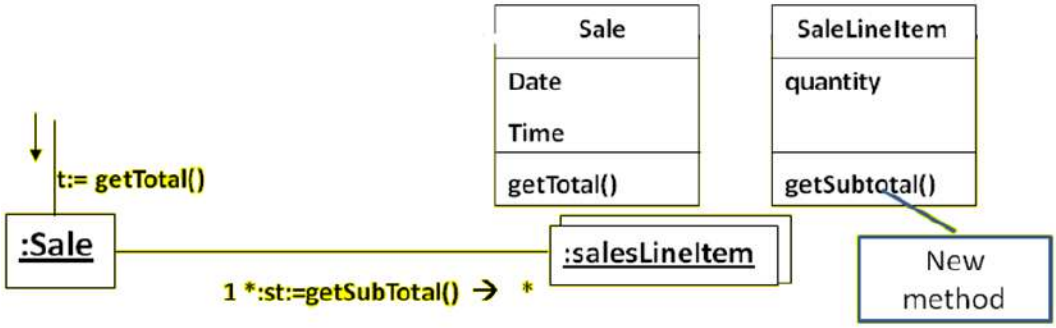
Responsibility to "provide access to relational databases" involves dozens of classes and hundreds of methods, packaged in a subsystem.

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.



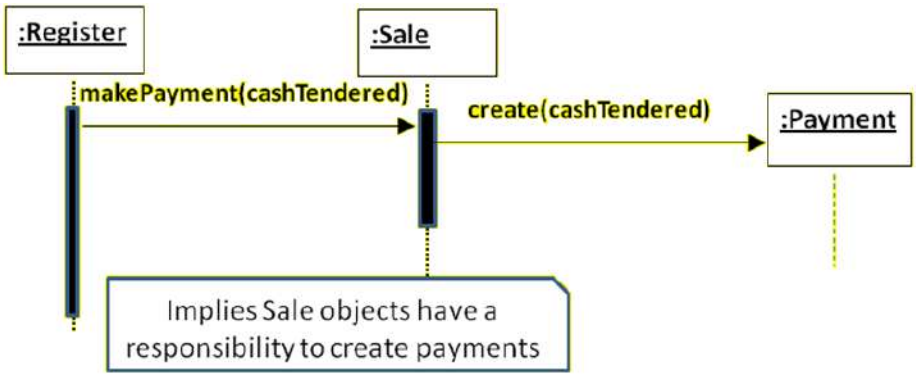
Responsibilities and Interaction Diagrams

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

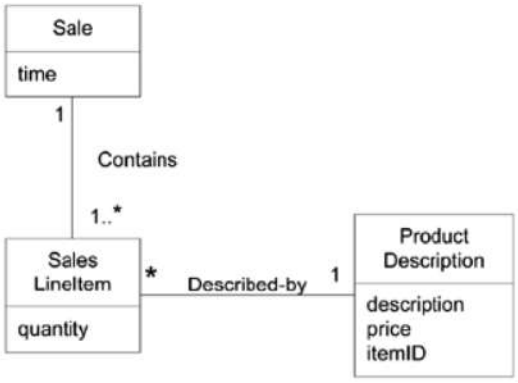
This pattern guides us to assign a responsibility to the information expert (class) that **has the *information*** necessary to fulfill the responsibility.

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.

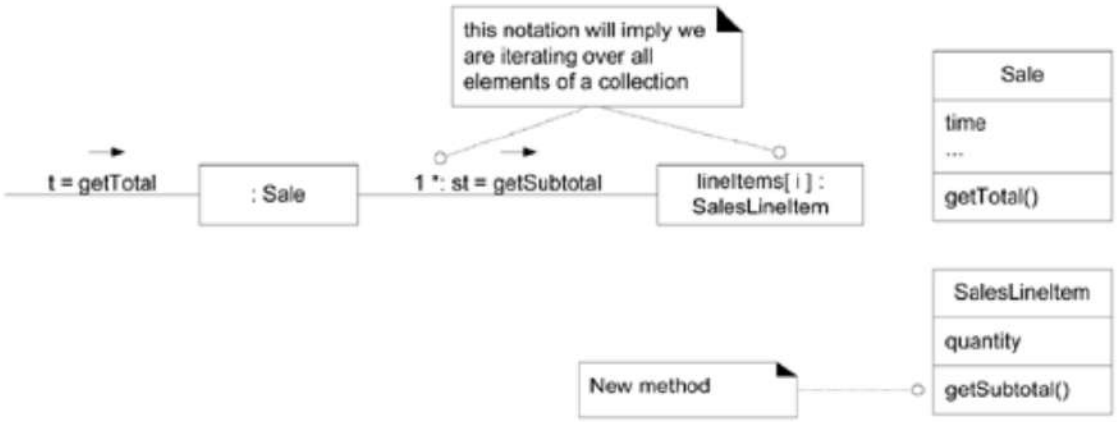
Only `Sales` instance in conceptual model knows the information about the grand total by having the all the information about the `SalesLineItem` in the Sale and the sum of their subtotals. Hence sale in information expert



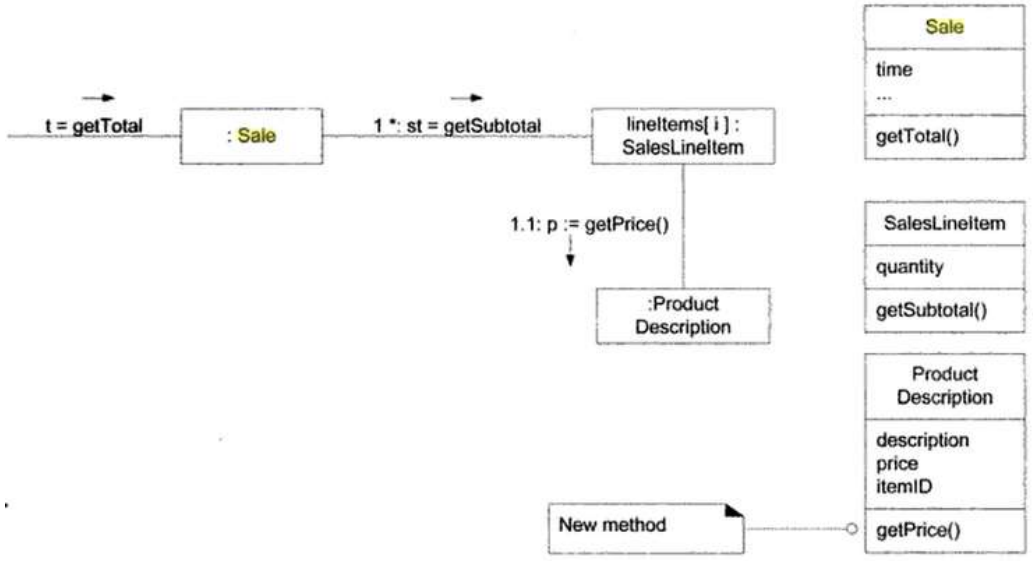
In terms of an interaction diagram, this means that the `Sale` needs to send `get-Subtotal` messages to each of the `SalesLineItems` and sum the results

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*

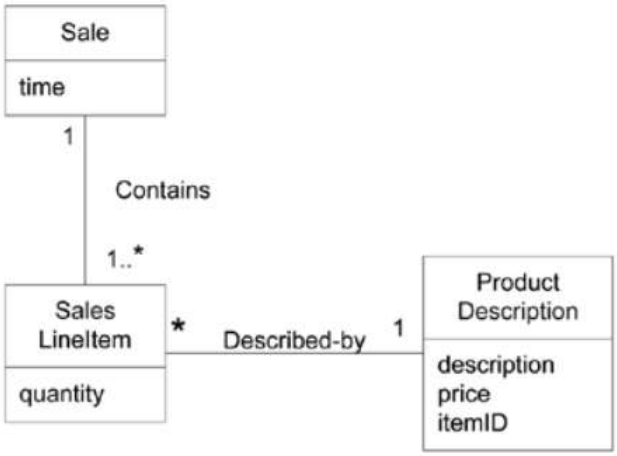
Recorded are all very common relationships between classes in a class diagram.

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

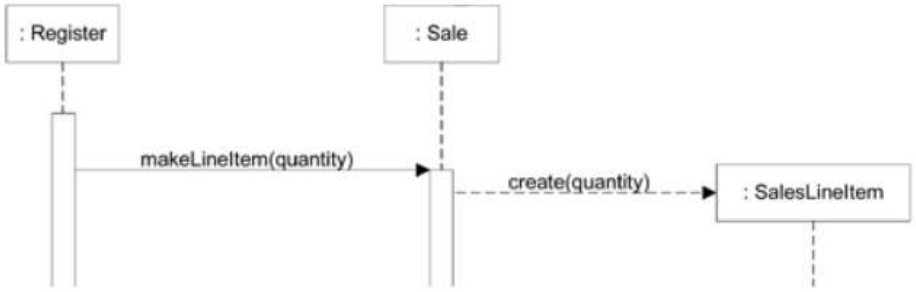
Who should be responsible for creation of a new class?

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**

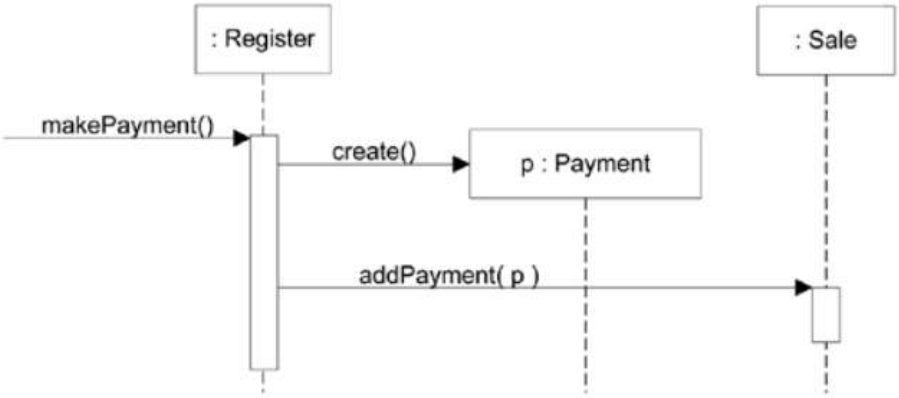
Low cohesion classes often represent a very "large grain" of abstraction, or have taken on responsibilities that **should have been delegated to other objects**

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.

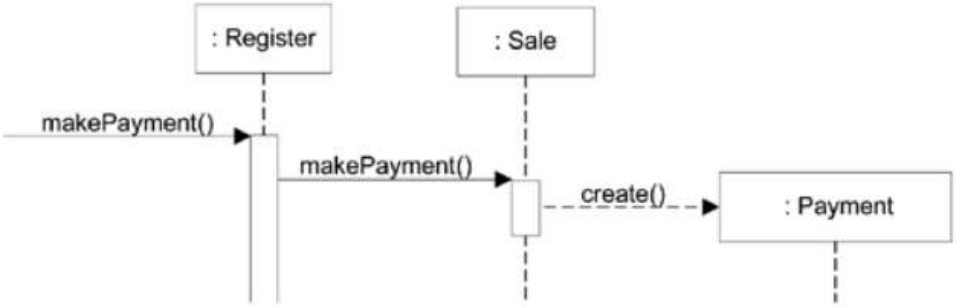
Here the `Register` is taking on part of the responsibility fulfilling the `makePayment` function.



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

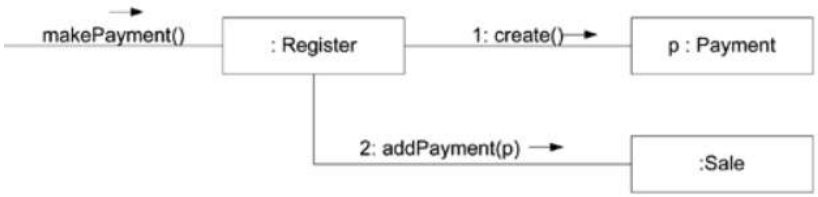
Consider the following partial class diagram from a NextGen case study:

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`.

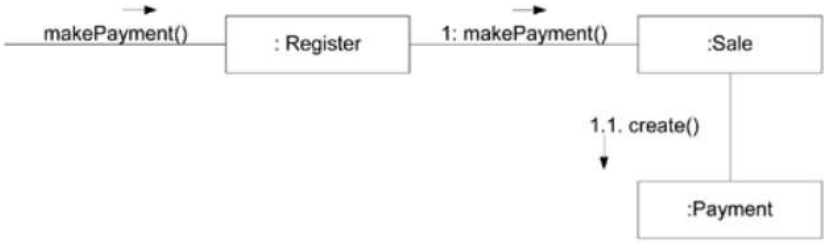
What class should be responsible for this?

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).

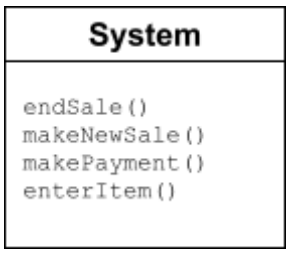
Who should be responsible for handling an input system event?

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

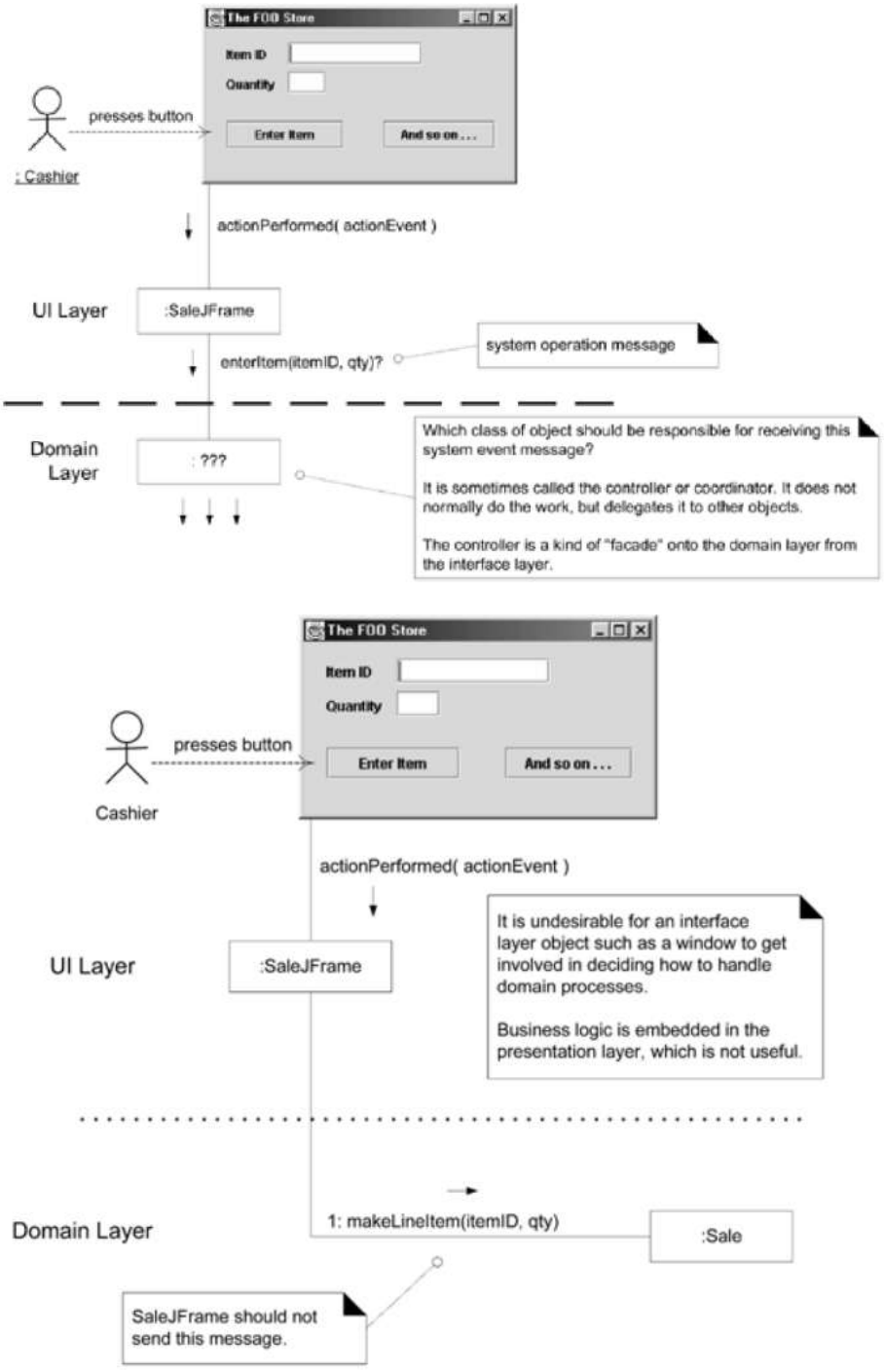
Choosing the Controller Class

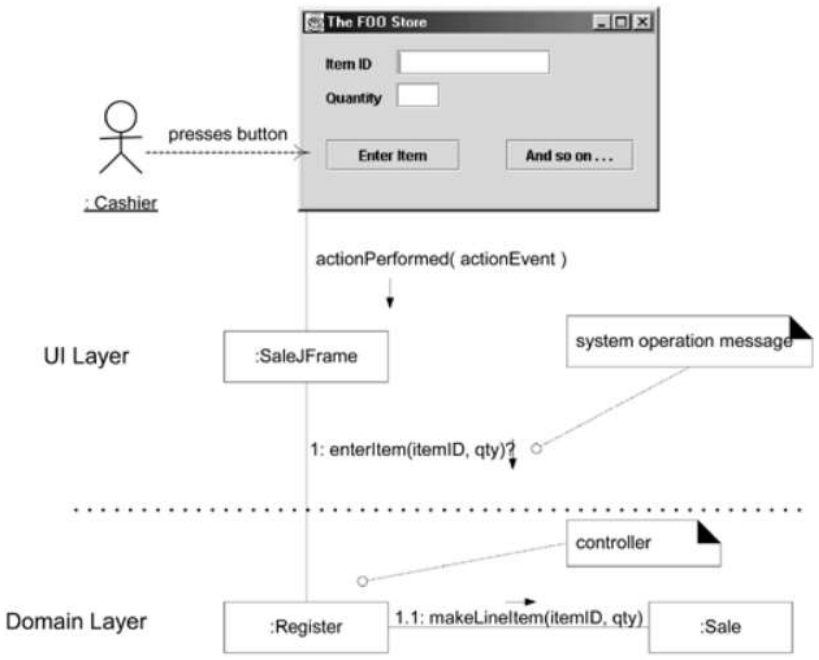
Our first choice involves choosing the controller for the system operation message `enterItem`

Choices: -

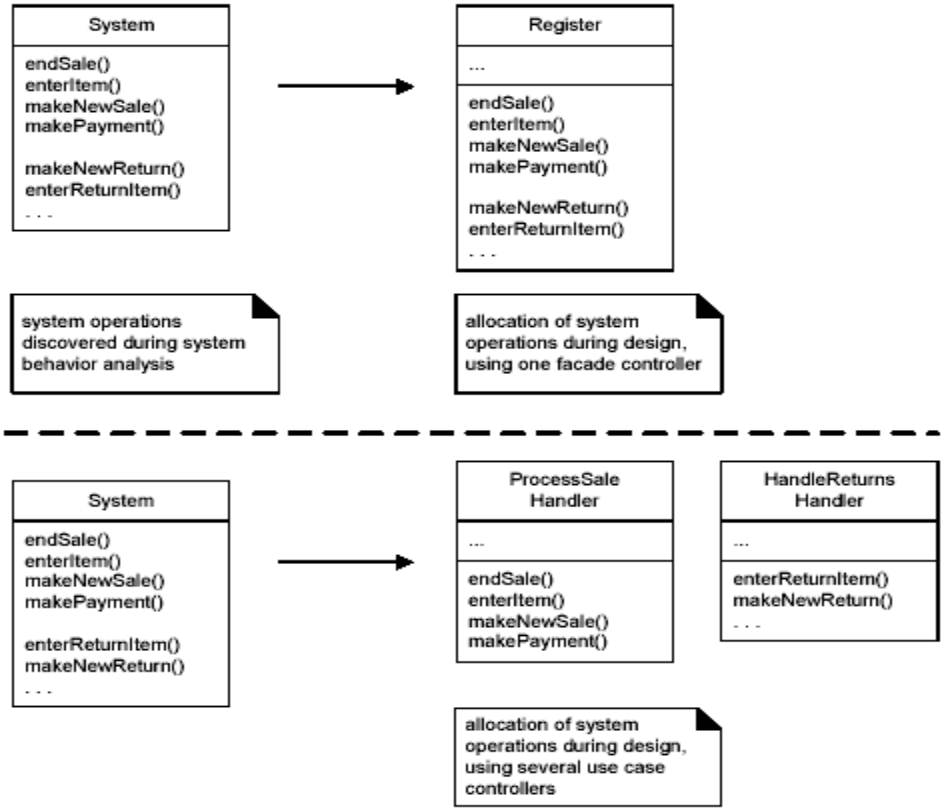
represents the overall "system," device, or `Register, POSTSystem` subsystem represents a receiver or handler of all system `ProcessSaleHandler`, events of a use case scenario `ProcessSaleSestsion`

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





Allocation of system operations



Design Model: Usecase Realizations with GRASP Patterns

A use-case realization describes how a particular use case is realized within the design model, **in terms of collaborating objects**

A designer can describe the design of one or more scenarios of a use case , also known as use case realization or scenario realization

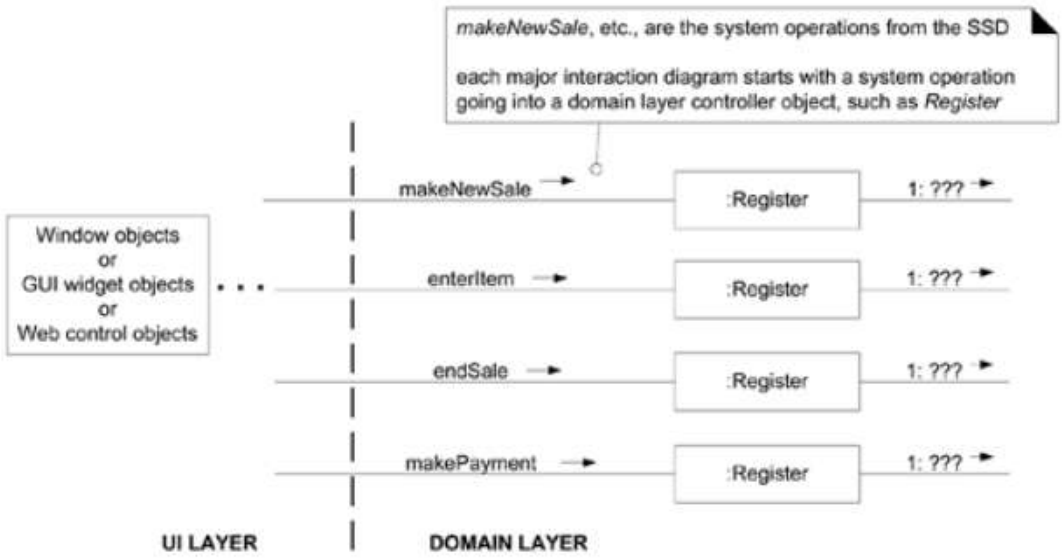
It reflects the connection between the requirements expressed as use cases and the object design that satisfies the requirements

Object design patterns can be applied during use case realization design work

Use case suggests system operations shown in System Sequence Diagrams

The system operations become the starting messages entering the controllers for domain layer interaction diagrams

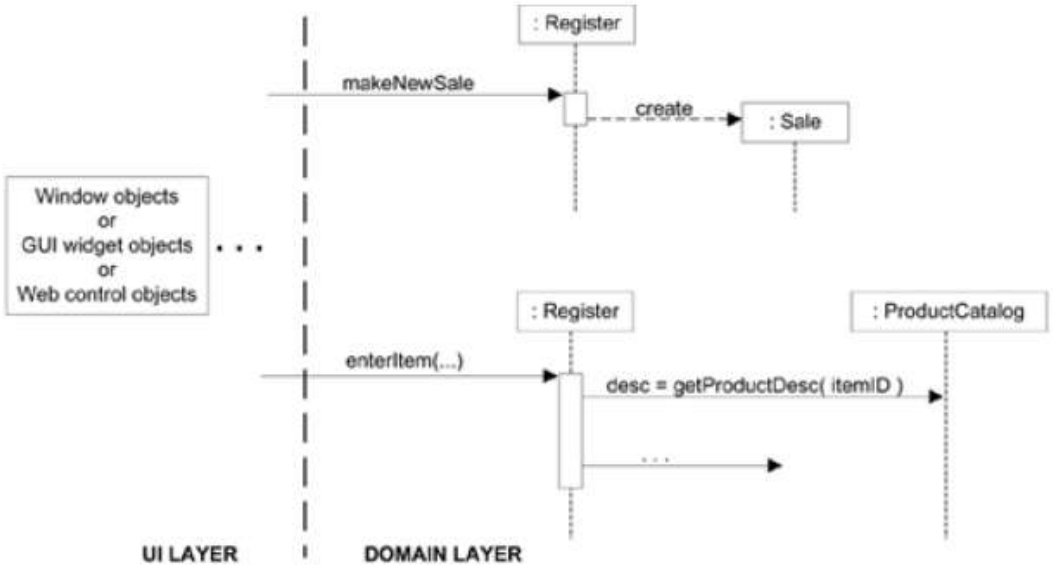
The domain layer interaction diagrams show the interaction of objects to fulfill the required tasks- use case realization



The system operations on the SSD of ProcessSale use are

- `makeNewSale`
- `enterItem`
- `endSale`
- `makePayment`

The handling of each of these system operation messages can be shown by drawing communication diagrams for each system operation **depicting the realization of use cases**



Operation contracts and Use case Realizations

Use case realizations are drawn from use case texts or from one's domain knowledge

Contracts may be added for complex system operations that add more analysis details

While designing use case realization or message interactions for satisfying the requirements, **post condition state changes** specified in operation contracts are also taken into consideration

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). - ...

For this `enterItem` system operation a partial interaction diagram can be drawn that shows the state change of `SaleLineItem` instance creation

Use Case Realization Examples:

Designing `makeNewSale`

The `makeNewSale` system operation occurs when a cashier requests to start a new sale, after a customer has arrived with things to buy. The use case may have been sufficient to decide what was necessary, but for this case study we wrote contracts for all the system events, for explanation and completeness.

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 the Register (association formed). - Attributes of s were initialized.

Choosing the Controller Class

The first choice involves **handling the responsibility for the system operation message** `enterItem`.

Based on the Controller pattern a controller class represents:

The overall system, robot object, a specialized device, a major subsystem

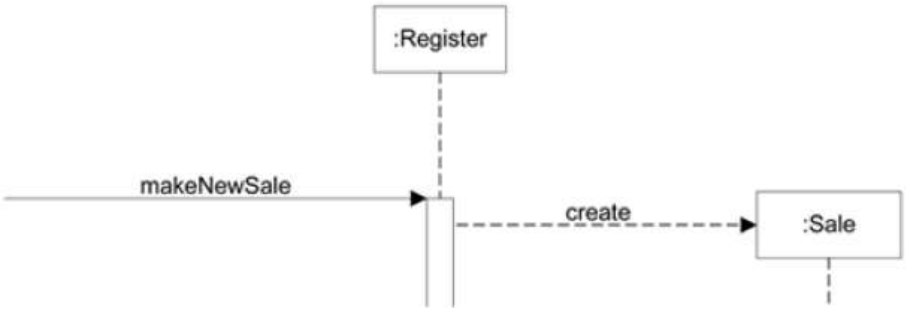
- Store a kind of root object as we think of other domain objects within the `Store`
- `Register`, a specialized device that the software runs on also called Point Of Sale Terminal (POST)
- `POSYSTEM` a name suggesting the overall system

A receiver or handler of all system events of a use case scenario

- `ProcessSaleHandler` constructed from the pattern <use-case-name> "Handler" or "Session" `ProcessSaleSession`
- Choosing a **device-object façade controller** like `Register` is satisfactory in there are only a **few system operations** and the façade controller is **not taking too many responsibilities**

Choosing a **use case controller** is useful when there are **many system operations** and those **responsibilities need to be distributed** to keep the control class lightweight and cohesive

- So by applying the Controller pattern, the software object in the design model called `Register` can be selected as there are few system operations and the interaction diagram begins by sending the system operation `makeNewSale` to the `Register` software object.



Creating new Sale by GRASP creator pattern

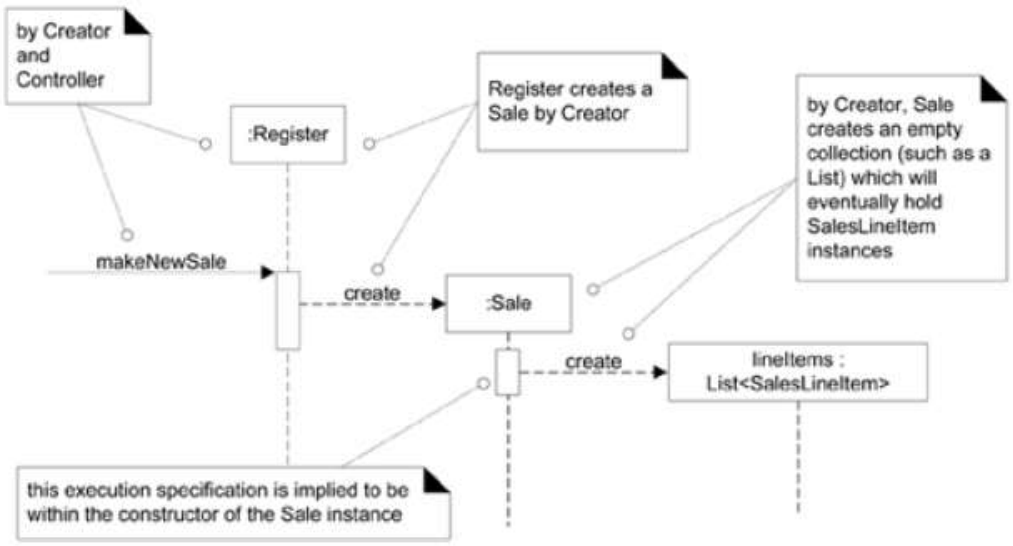
The post condition of the operation contract suggests that a new `Sale` instance must be created and GRASP creator pattern suggest assigning the responsibility of creating a class to one that **aggregates**, **contains** or **records** the object to be created

From the conceptual model, `Register`/`POST` **may be considered of as recording** `Sale`. Thus `POST` is a reasonable candidate for recording Sale.

In addition, when the `Sale` is created, it must contain an empty collection (a list) to record all the future `SalesLineItem` instance that will be added.

This **collection will be maintained by the sales** instance, which implies by Creator is good candidate for creating it.

Hence `Register`/`POST` **creates** `Sale`, `Sale` **creates empty collection** represented by a multi object in the communication diagram.



Designing `enterItem`

The `enterItem` system operation occurs when a cashier enters the `itemID` and (optionally) the quantity of something to be purchased.

Contract CO3: `enterItem`

Operation:	enterItem(itemID : ItemID, quantity : integer)
Cross References:	Use Cases: Process Sale
Preconditions:	There is an underway sale.
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).

Interaction diagram has to be drawn to satisfy the post conditions using the GRASP pattern

Choosing the Controller Class

Our first choice involves handling the responsibility for the system operation message `enterItem`. Based on the Controller GRASP pattern, as for `makeNewSale`, we will continue to use `Register` as a controller.

Creating a new `SalesLineItem`

The `enterItem` contract post condition indicates the creation, association and initialization of a `SalesLineItem` object.

Analysis of domain model reveals that Sale contains `SalesLineItem` objects

So software `Sale` can contain software `SalesLineItem` objects hence by Creator pattern software `Sale` is an appropriate candidate for creating `SalesLineItem`

Sales contains `SalesLineItem` objects, hence it is appropriate for creating line items.

The new `SalesLineItem` needs a quantity, when created, therefore the POST must pass it along to the `Sale`, which must pass it along as a parameter in the create message. (a constructor call with a parameter)

Therefore by creator a `makeLineItem` message is sent to a sale in order for it to create a `SalesLineItem`.

Sale creates a `SalesLineItem`, and then stores the new instance in its permanent collection

The parameters to `makeLineItem` message include the quantity so that `SalesLineItem` can record it and the `ProductDescription` that matches the `itemID`

Finding a Product Description

The `SalesLineItem` needs to be associated with the product specification that matches the incoming `itemID`. This implies it is necessary to retrieve a `ProductDescription` based on an `itemID` match.

The GRASP pattern suggests that the object that has the information required to fulfill the responsibility should do it.

Who should be responsible for knowing `ProductDescription` based on an `itemID` match?

We find from class diagram that the `ProductCatalogue` **logically contains all the** `ProductDescriptions`, and thus by Information Expert pattern, `ProductCatalogue` is a good candidate for this look up responsibility.

Visibility to `ProductCatalogue`

Who sends the specification method message to the `ProductCatalogue` to ask for a `ProductDescription`?

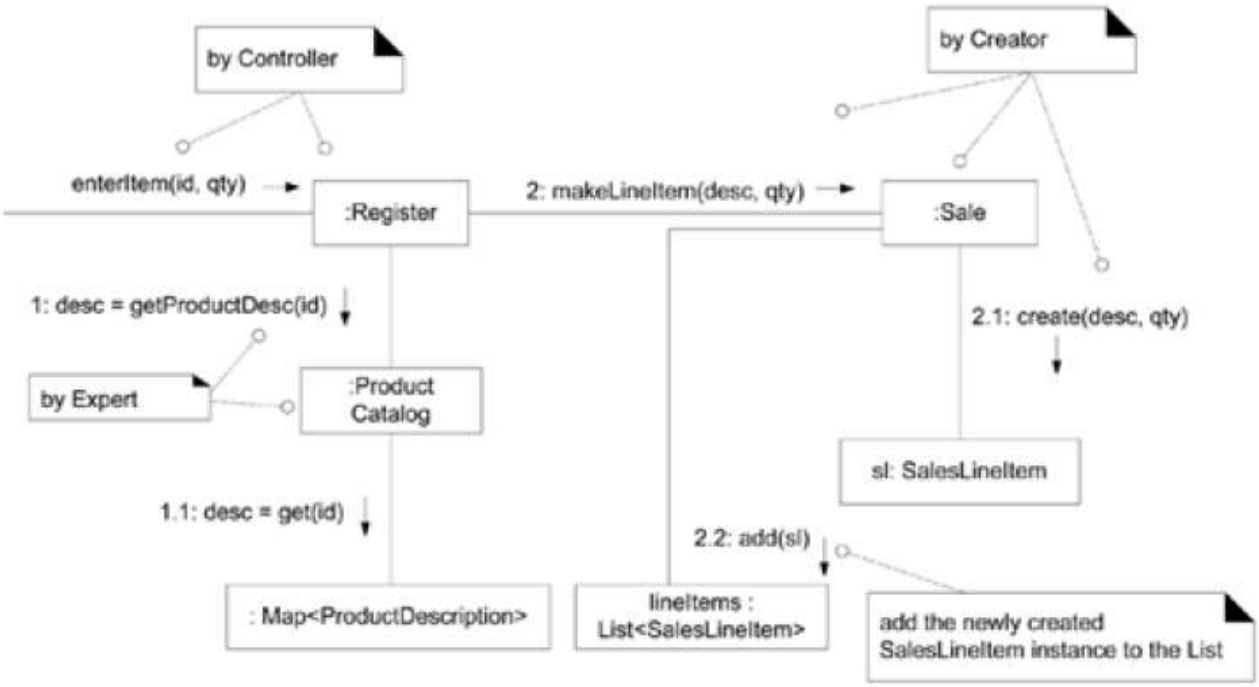
We find that there is a permanent connection between the POST object and the `ProductCatalogue` object as were created during the initial `Startup` use case

Hence it is possible for the POST to send the specification message to `ProductCatalogue`

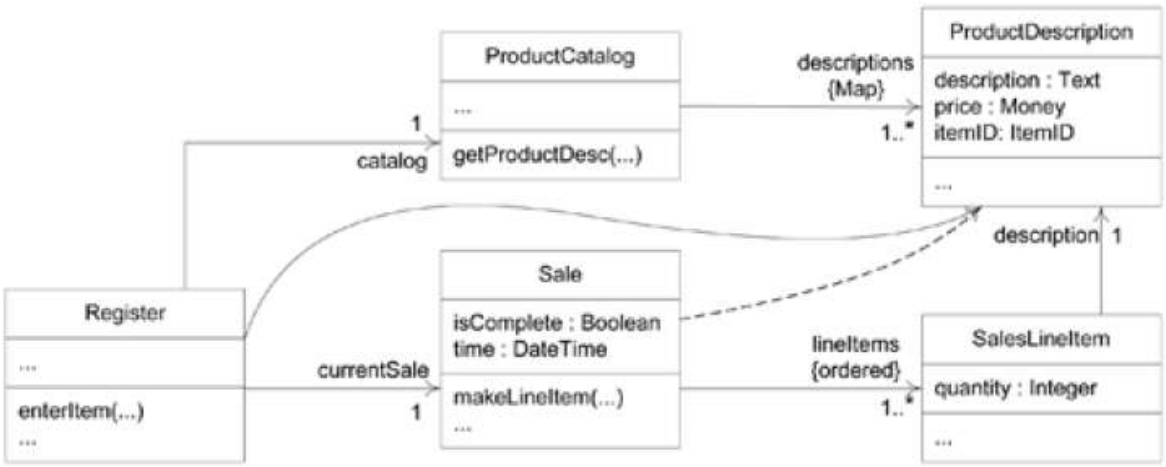
"**Visibility** it is the ability of one object to see or have reference to another object"

Since **Register/POST** has a permanent connection or reference to the `ProductCatalogue` it has visibility to it and hence can send messages such as `getProductDescription`

The `enterItem` communication diagram: Dynamic View



The `enterItem` communication diagram: Static View



Designing `endSale`

The `endSale` system operation occurs when a cashier presses a button indicating the end of entering items into a `Sale`.

Contract CO2: `endSale`

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

Choosing the Controller Class

Our first choice involves handling the responsibility for the system operation message `endSale`. Based on the Controller GRASP pattern, as for `enterItem`, we will continue to use `Register` as a controller.

Setting the `Sale.isComplete` Attribute

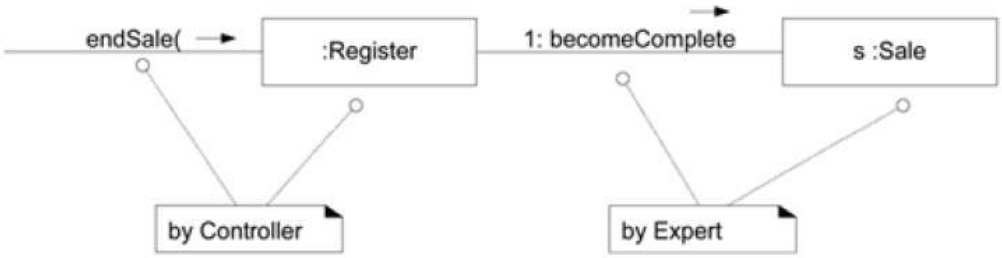
The contract postconditions state:

• `Sale.isComplete` became true (attribute modification).

As always, Expert should be the first pattern considered unless it is a controller or creation problem (which it is not).

Who should be responsible for setting the `isComplete` attribute of the `Sale` to true?

By Expert, it should be the Sale itself, since it owns and maintains the `isComplete` attribute. Thus the `Register` will send a `becomeComplete` message to `Sale` to set it to true.



Calculating Sale Total

Process Sale use Case:

1. Customer Arrives
2. Cashier tells System to create a new Sale
3. Cashier enters item identifier
4. System records sale line item and …

   Cashier repeats 3-4 until indicates done
5. System presents total with tax calculated

How to assign the responsibility of knowing the total?

1. State the responsibility:
   a. Who should be responsible of knowing the sale total?
2. Summarize the information required:
   a. The sale total is the sum of subtotals of all sale line items
   b. `Sales line item subtotal := line item quantity * product description price`
3. List information required to fulfill the responsibility and the classes that know about this information

Information required for `Sale` Total	**Information Expert**
`ProductDescription.price`	`ProductDescription`
`SalesLineItem.quantity`	`SalesLineItem`
All the `SalesLineItems` in current sale	`Sale`

Analysis of the reasoning process:

Who should be responsible for calculating the `Sale` total?

By information expert, it should be the `Sale` itself as it knows about all `SalesLineItem` instances whose subtotals must be summed to calculate the `Sale` total.

`Sale` should have the responsibility of knowing the total implemented as the method `getTotal`.

Now for `Sale` to calculate `Total` it needs the subtotal for each `SalesLineItem`.

Who should be responsible for calculating the `SalesLineItem` subtotal?

By information expert, it should be the `SalesLineItem` itself as it knows about the quantity and the `ProductDescription`.

`SalesLineItem` should have the responsibility of knowing the `subtotal` implemented as the method `getSubTotal`.

Now for `SalesLineItem` to calculate `subTotal` it needs the price of `ProductDescription`.

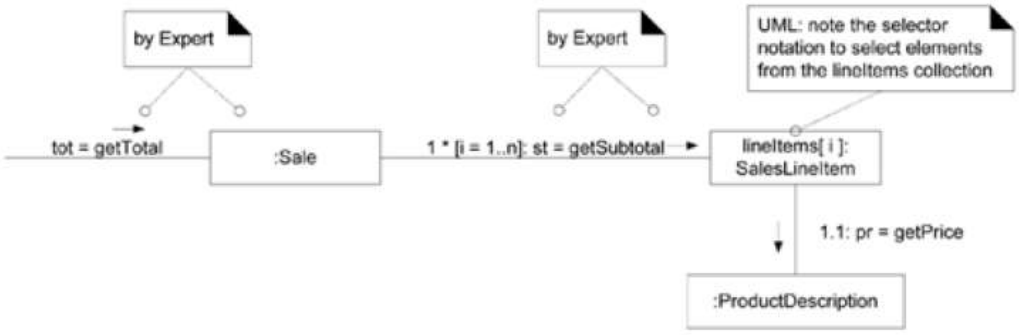
Who should be responsible for providing `ProductDescription` price?

By information expert, it should be the `ProductDescription` itself as it encapsulates the `price` as an attribute.

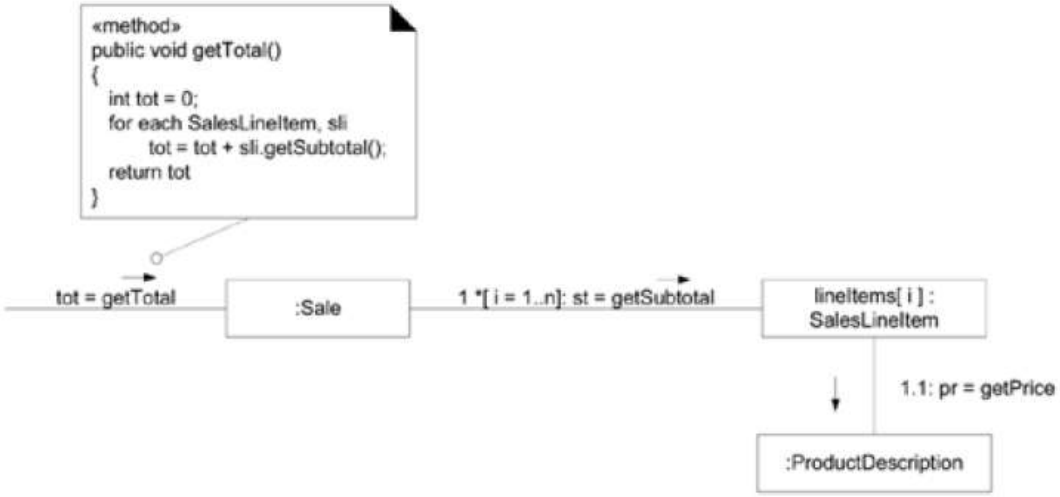
`ProductDescription` should have the responsibility of knowing its `price` implemented as the method `getPrice`.

Interaction Diagram for this Scenario

When `Sale` receives a `getTotal` message (which is not a system operation message like `makeNewSale` or `enterItem`),it sends a `getSubTotal` message to all related `SalesLineItem` instances. The `SaleLineItem` sends `getPrice` message to its associated `ProductDescriptions` (items bought by the customer)



Showing a method in a note symbol in UML 2.0



Designing `makePayment`

The `makePayment` system operation occurs when a cashier enters amount of cash tendered for payment

Contract CO4: `makePayment`

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

In case of the contract post condition

- A `payment` instance `p` was created (instance creation)

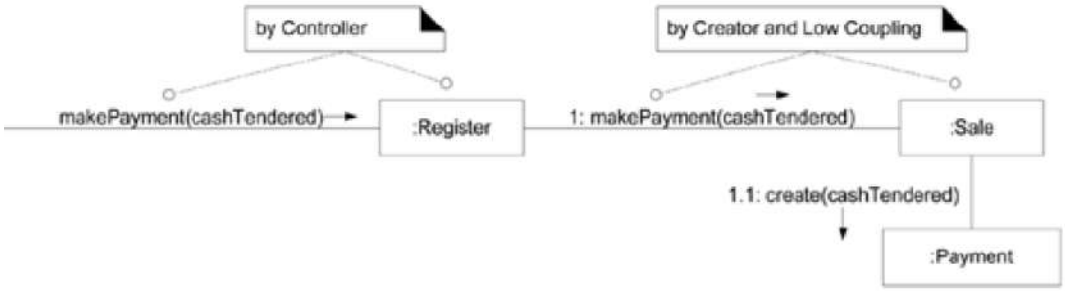
There are two classes `Register` and `Sale` for fulfilling the obligation of creating the `payment` instance

In real domain a "`Register`" records account information, making it an eligible candidate but `Sale` software also closely uses a `Payment`

When there are alternative design choices, consider Cohesion and Coupling implications of the alternatives and choose an alternative with good cohesion, coupling and stability in the presence of future changes

Why choose `Sale` instead of `Register` for creating `Payment` Instance?

- Choosing `Sale` to create `Payment`, the work of `Register` becomes lighter
- The `register` does not need to know about the existence of `Payment` as it can be recorded indirectly via `Sale` leading to lower coupling in `Register`



This diagram satisfies the post conditions of the contract:

- `Payment` has been created
- `Payment` has been associated with `Sale`
- `Payment's` amount tendered has been set

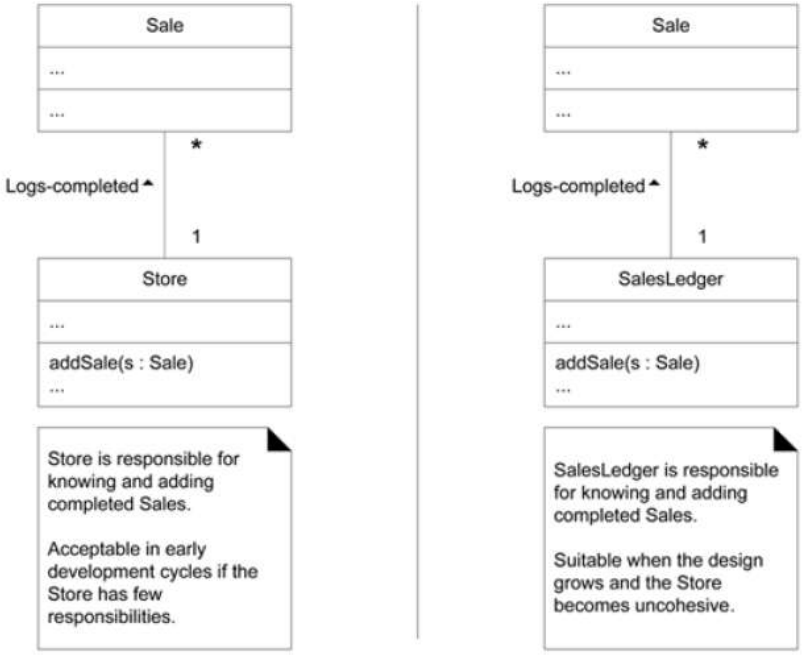
Logging a Sale

According to the requirements, the `sale` should be placed in a historical log after it is complete

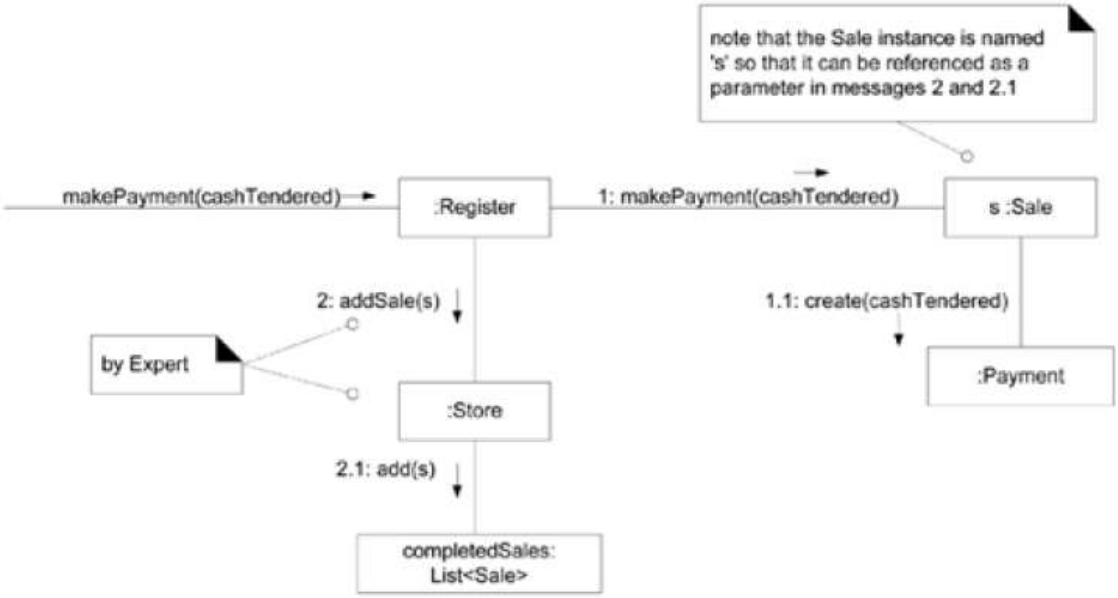
Who is responsible for knowing all the logged sales and doing the logging?

`Store` should be responsible for knowing all the logged sales since they are strongly related to its finances

`SalesLedger` might be an alternative based on classic accounting concepts but it will become suitable as design grows and `Store` becomes incohesive



The post conditions of the contract also indicate relating `Sale` to `Store`



Calculating `Balance`

The Process `Sale` use case implies that the balance due from a payment be printed on a receipt and displayed somehow

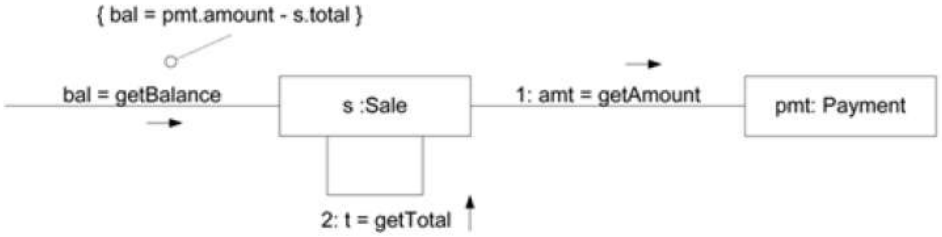
Who is responsible for knowing the `balance`?

To calculate the `balance`, the sale total and `payment cash tendered` are needed

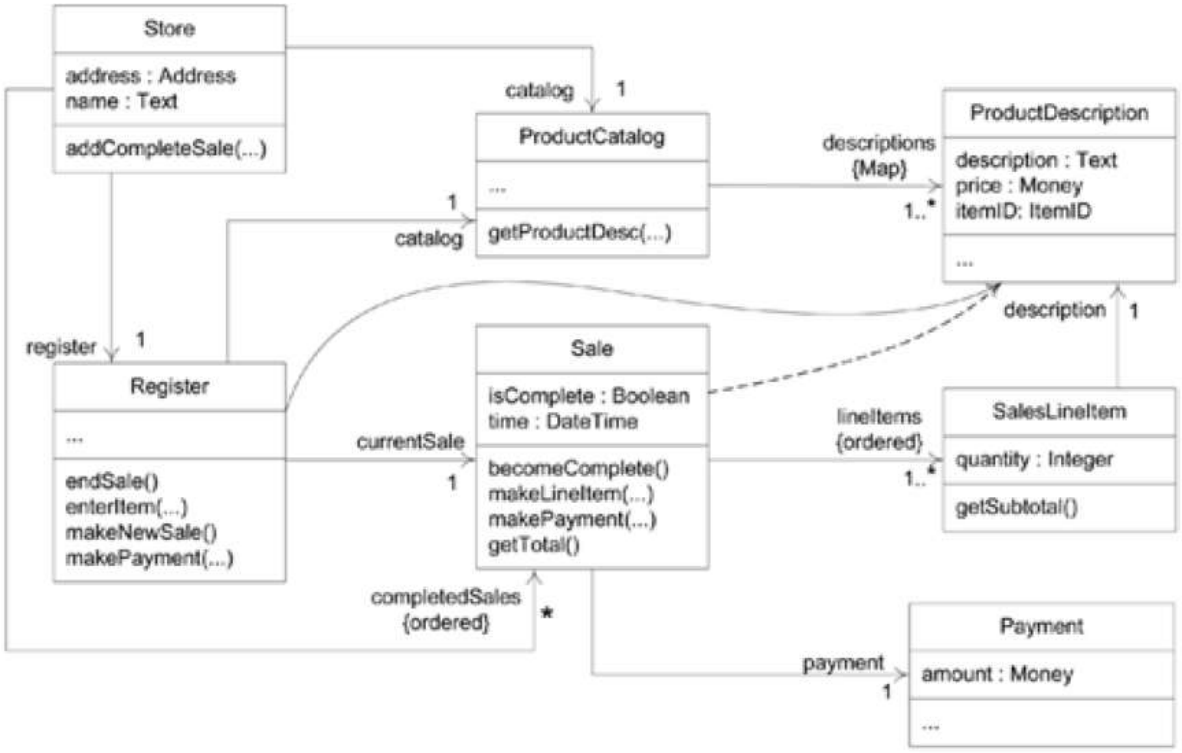
So `Sale` and `Payment` are potential candidates for knowing this information

If `Payment` is chosen to know the balance then it needs visibility to `Sale` to know about the total hence this approach would increase the overall coupling in the design

If `Sale` is chosen then it also needs visibility to `Payment` to know about the cash tendered but `Sale` already has visibility to `Payment` as its creator, so this approach does not increase the overall coupling and is a preferable design.



A more complete Design Class Diagram

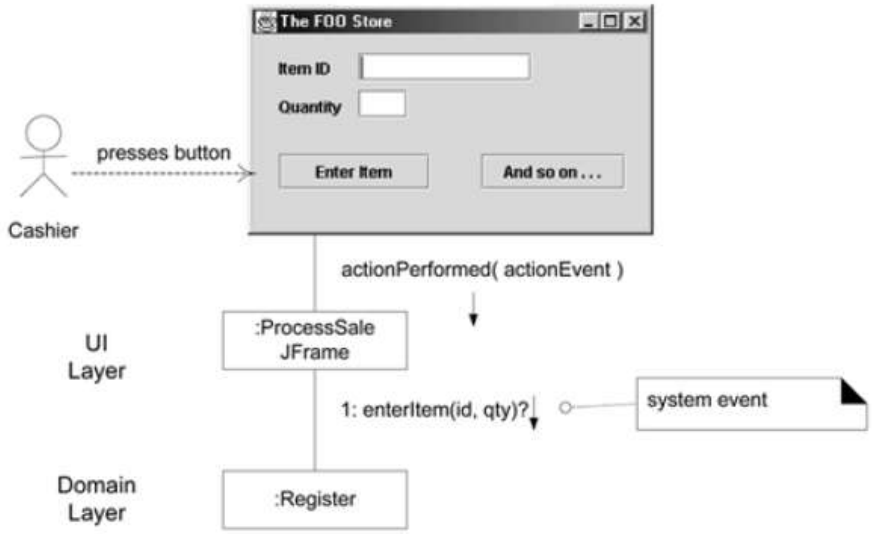


Connecting UI Layer to Domain Layer

Designs by which objects in the UI Layer obtain visibility to the objects in the Domain Layer:

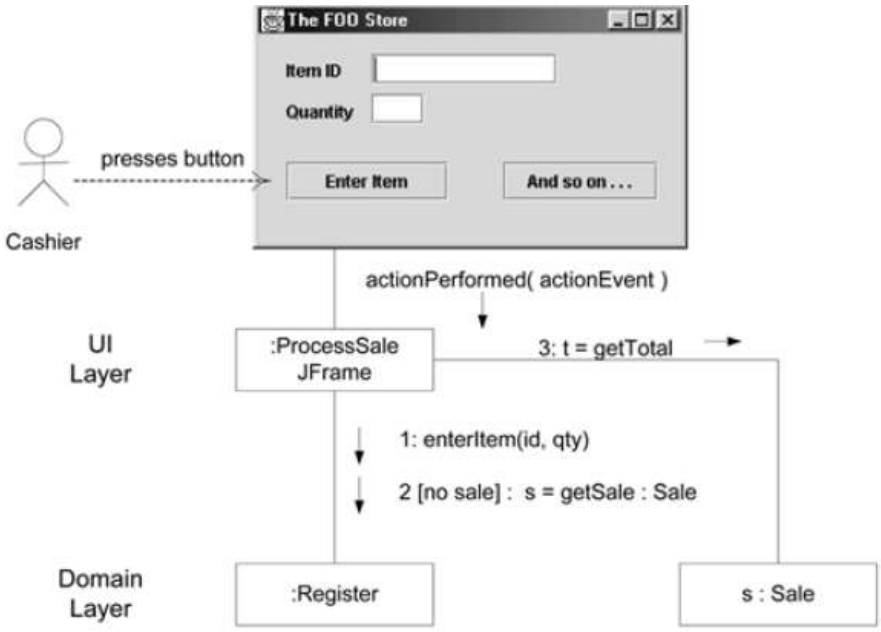
- An initializer object called from the application starting method (e.g. Java Main method) creates both a UI and a Domain object and passes the Domain object to the UI
- A UI object retrieves the Domain object

Once the UI object has reference to the `Register` object, it can forward system event messages like `enterItem`, `endSale` message to it



If the window is required to show running total after each entry of the `enterItem` message, the design choices are:

- Add a `getTotal` method to `Register` . the UI sends `getTotal` message to the `Register` which then delegates to the `Sale`. This approach maintains lower coupling from the UI to the Domain Layer
- A UI has a reference to the current `Sale` object, when it requires the `total` or information regarding `sale` , it sends message directly to `Sale`. This does increase coupling from the UI to Domain Layer



Initialization and Startup Use Case

These are **concerned with initialization and starting up** of the application when it is launched

Choosing the initial domain object

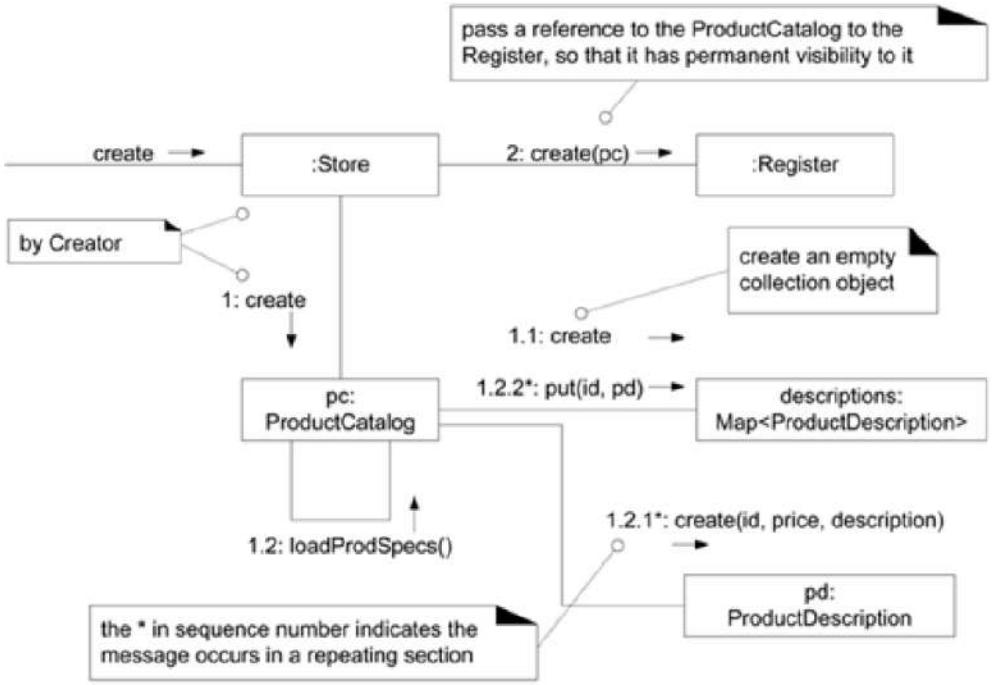
What should the **class of initial domain object** be?

The class **at the root of containment or aggregation hierarchy** of domain objects should be chosen as the initial domain object. Like `Register` or `Store` **which contains almost all other objects**

`Store.create` **Design**

The initialization work include:

- Create `Store, Register, ProductCatalog, ProductDescriptions`
- Associate `ProductCatalog` with `ProductDescriptions`
- Associate `Store` with `ProductCatalog`
- Associate `Store` with `Register`
- Associate `Register` with `ProductCatalog`

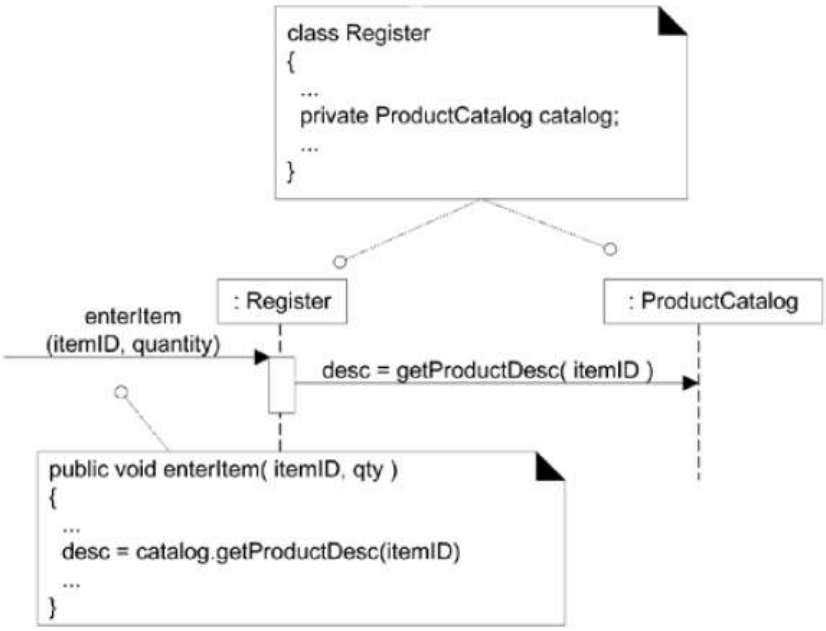


Determining Visibility: Visibility between Objects

For a Sender Object to send message to a receiver object, the sender must be visible to the receiver or sender must have some kind of reference or pointer to the receiver object

For example, the `getProductDescription` message sent from a `Register` to a `ProductCatalog` implies that the `ProductCatalog` instance is visible to the `Register` instance

A reference to the `ProductCatalog` instance is **maintained as an attribute of the** `Register`**, giving** `Register` **a visibility to the** `ProductCatalog`



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?

Q. When is visibility necessary?

A. 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

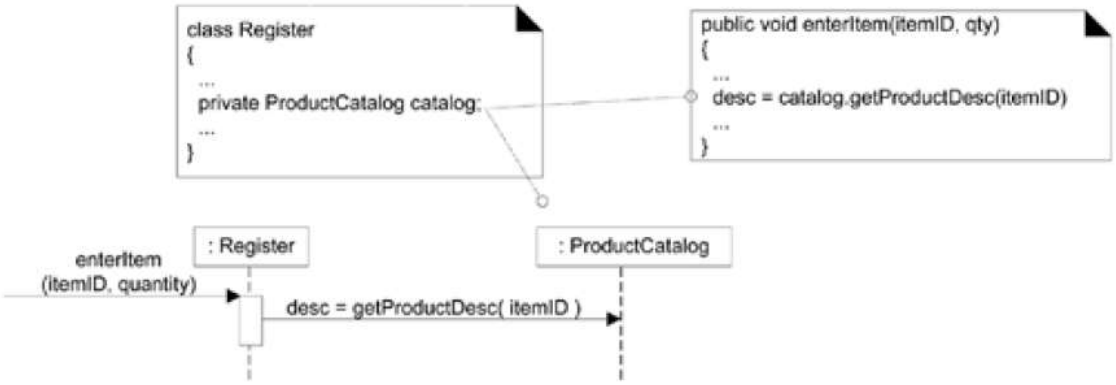
It exist from A to B when B is an attribute of A.

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;
  …
```

}

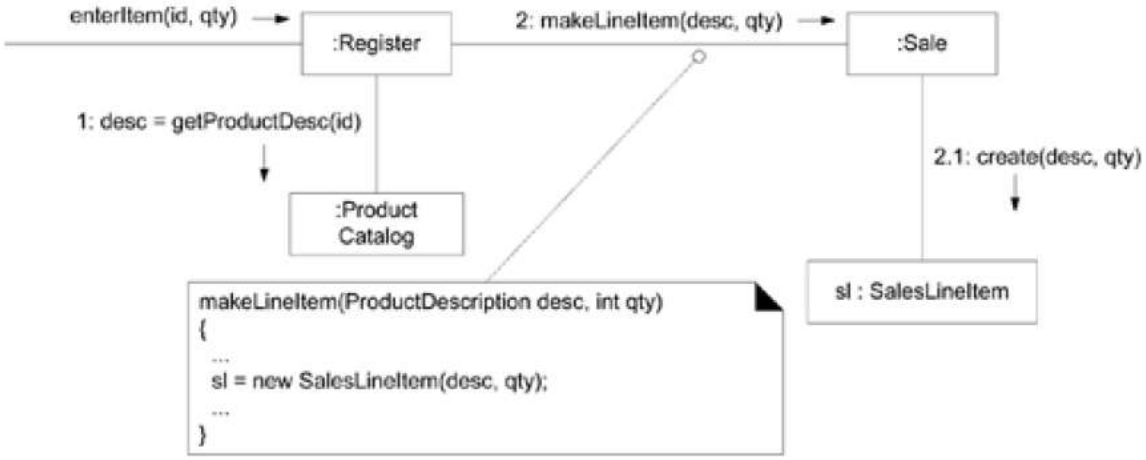


This visibility allows the `Register` to send the `getProductDescription` message to a `ProductCatalog`

ii. Parameter Visibility

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

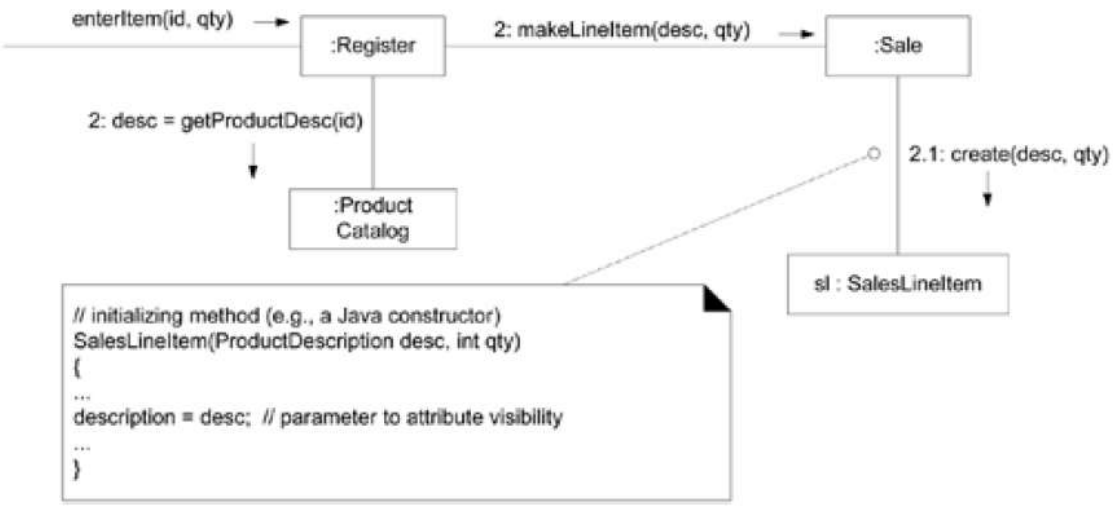
It is temporary as it persists only within the scope of the method.



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



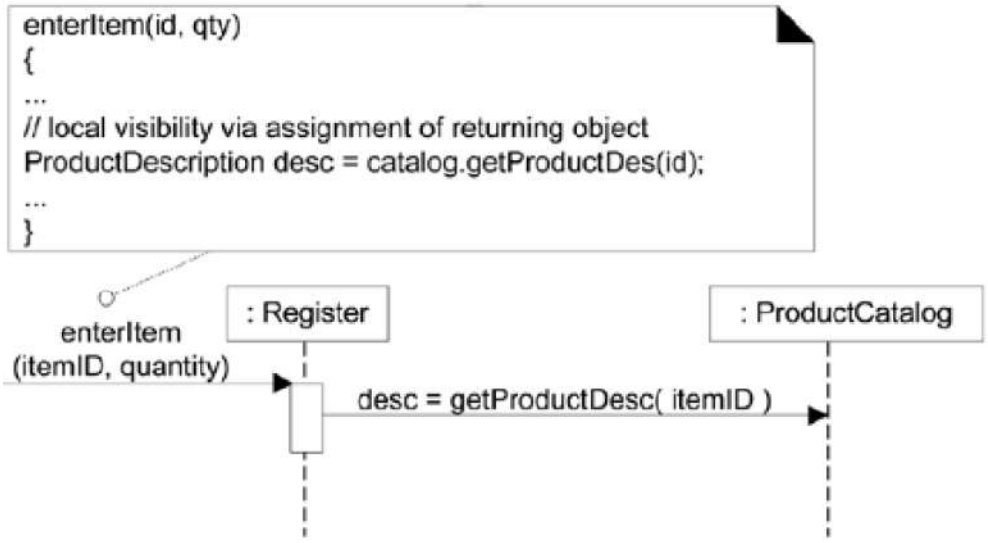
iii. Locally Declared Visibility

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)

B. Creating Design Class Diagrams

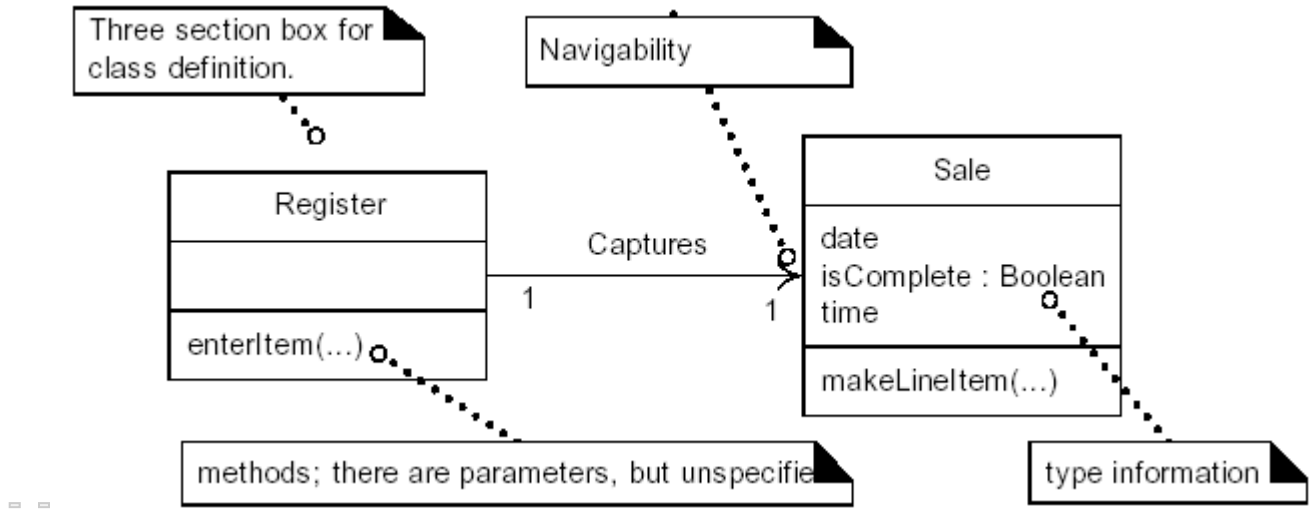
Activities and Dependencies

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

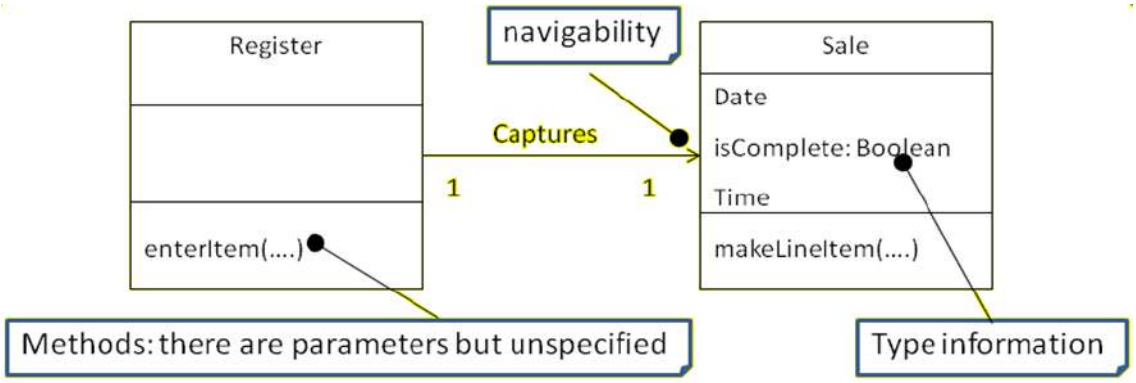
UML has no explicit notation for DCDs

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

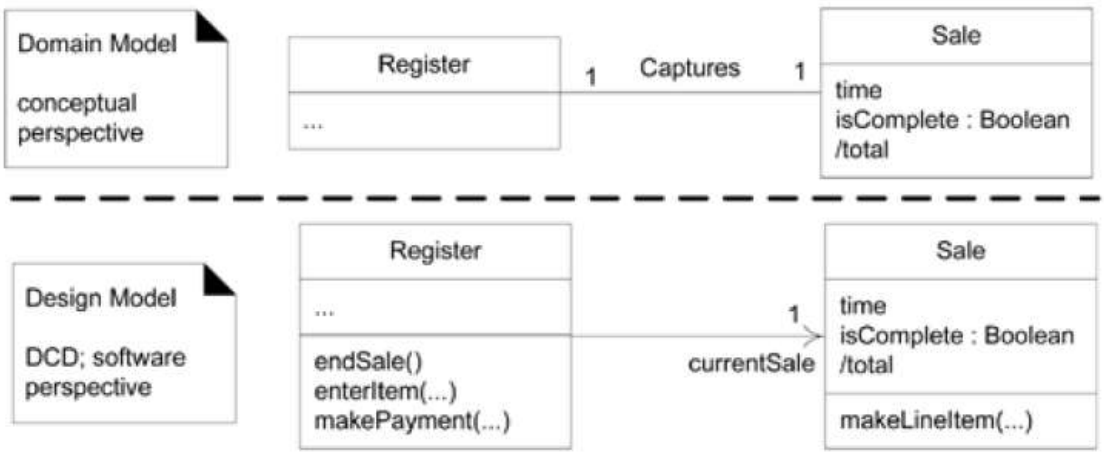
Navigability Dependencies



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.

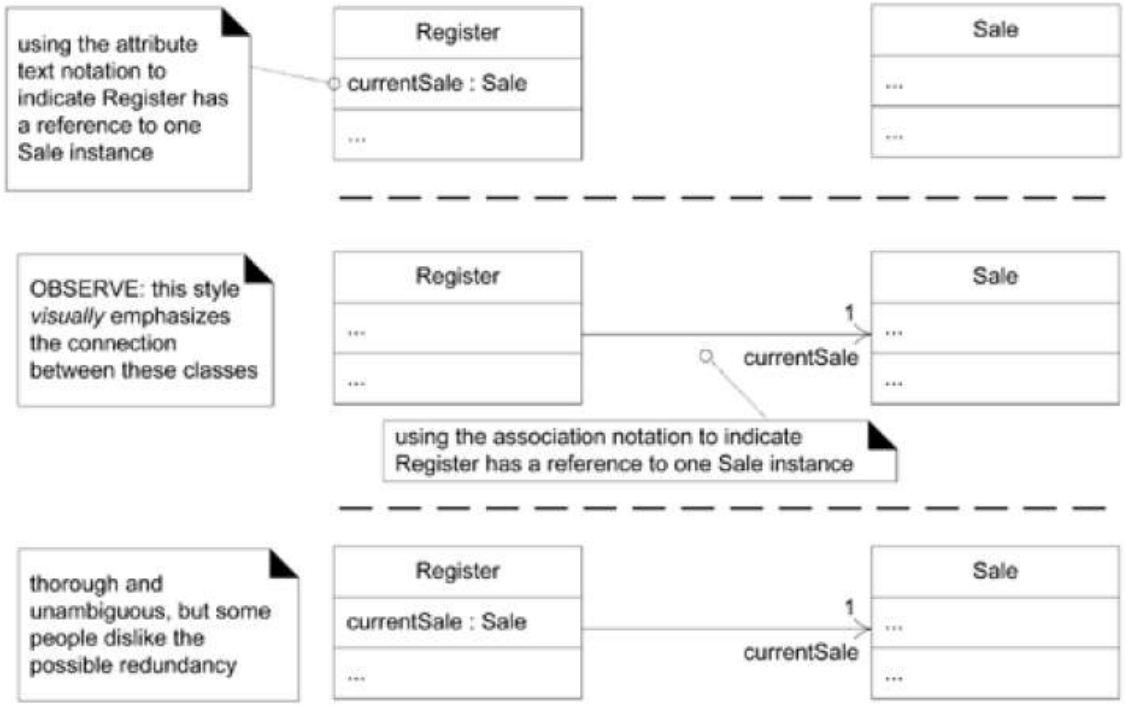


UML uses Class Diagram for Design Class Diagram

Showing UML Attributes

Attributes of a classifier are show in several ways:

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

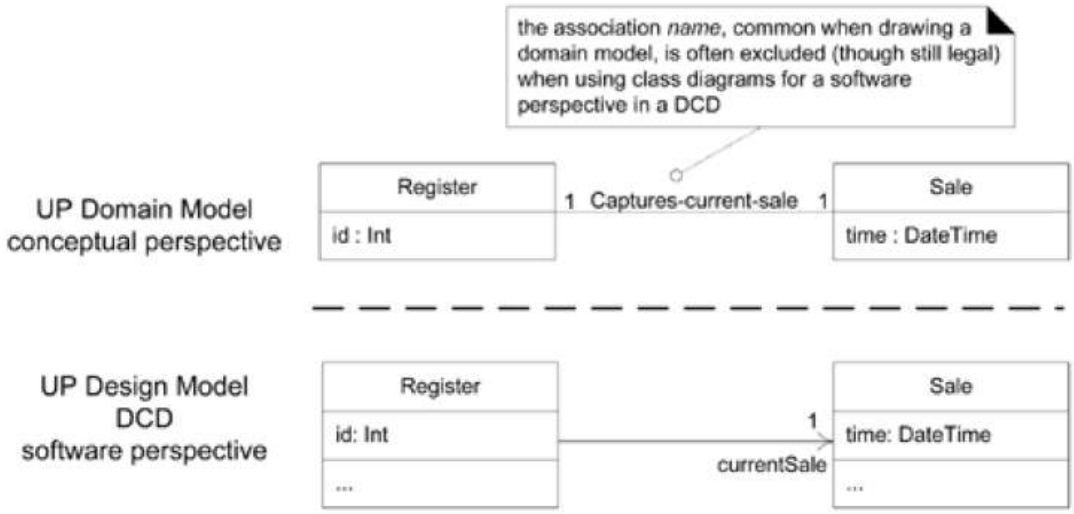


The full format of text attribute notation:

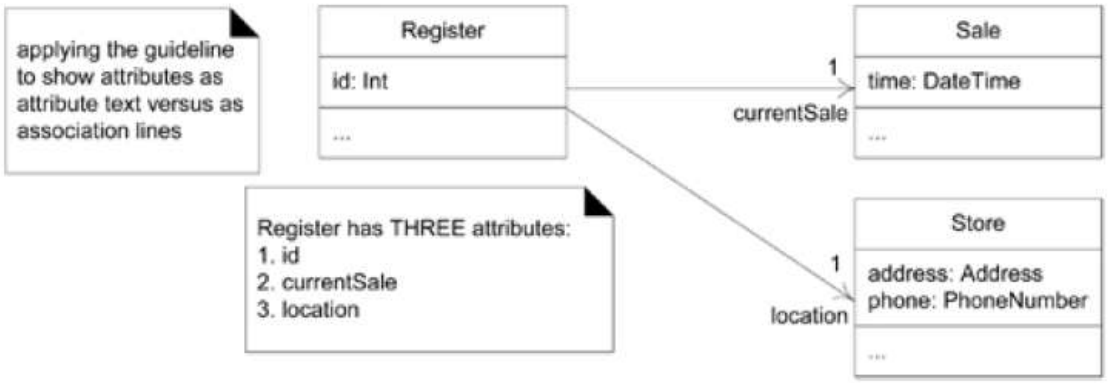
```
visibility name : type multiplicity = default {property-string}
```

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

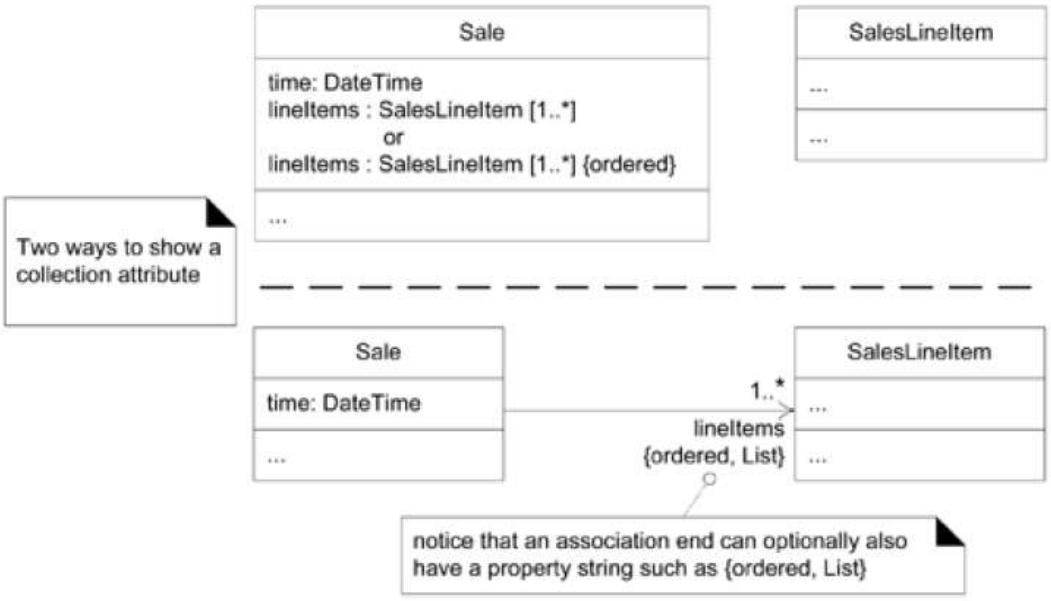
{unique} implies a set of unique elements

Showing Collection Attributes with Attribute Text and Association Lines

A Sale software object holding a list of many SalesLineItem objects

```
public class Sale{
      private List<SalesLineItem> lineItems = New ArrayList<SalesLineItem>();
      ------------
}
```

Can be depicted as



Note Symbols:

- A UML note symbol is displayed as a dog eared rectangle with a dashed line to the annotated element and it may represent:
- A UML note or comment
- A UML constraint encased in braces {….}
- A method body, implementation of an UML operation

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>>

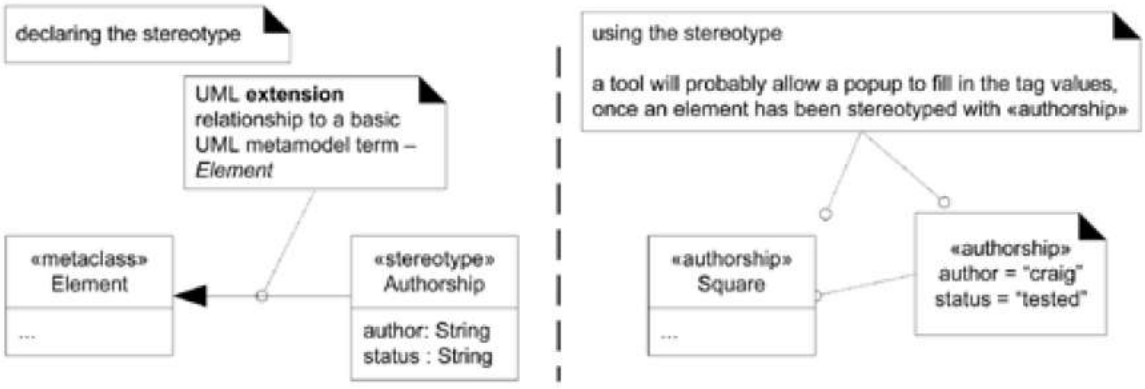
A stereotype represents a refinement of an existing modeling concept and is defined within a UML profile

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).

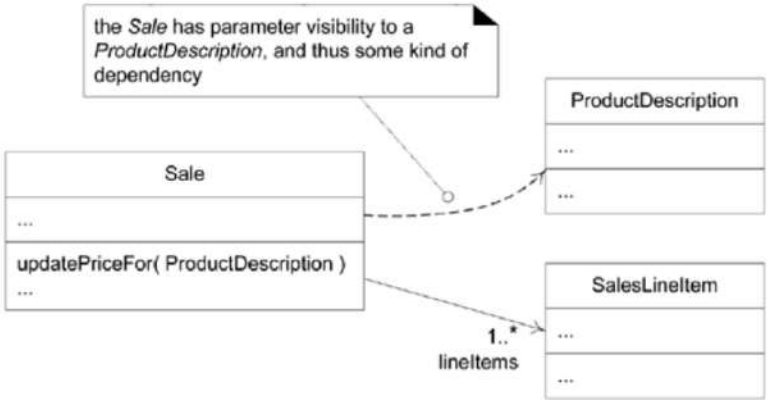
In class diagrams use the dependency line **to depict global, parameter variable, local variable, and static-method** (when a call is made to a static method of another class) **dependency** between objects.

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.

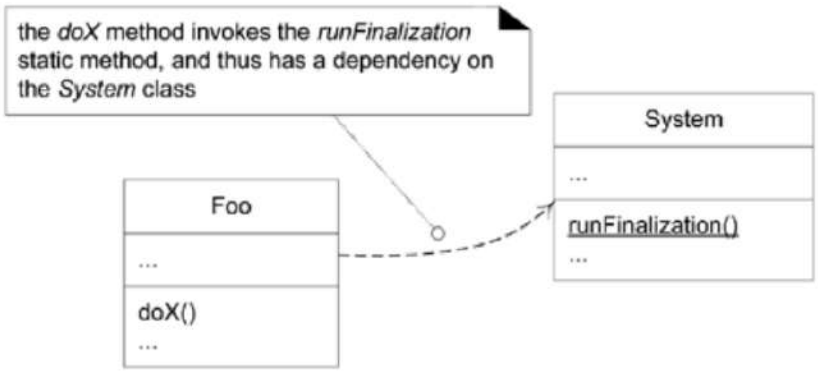
If the ProductDescription class is changed, the **Sale class could be affected**.



Another example: A doX method in the Foo class:

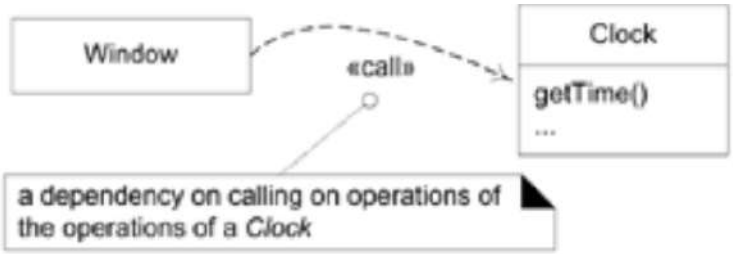
```
public class Foo
{
     public void doX()
     {
          System.runFinalization();
          //...
     }
// ...
}
```

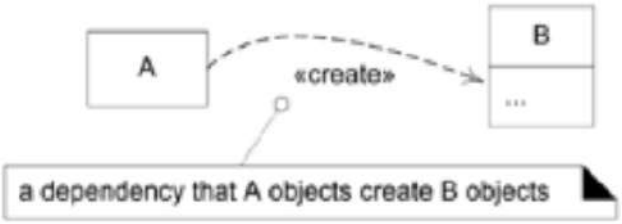
The doX method **invokes a static method on the System class**. Therefore, the Foo object has a **static-method dependency** on the System class.



Dependency Labels

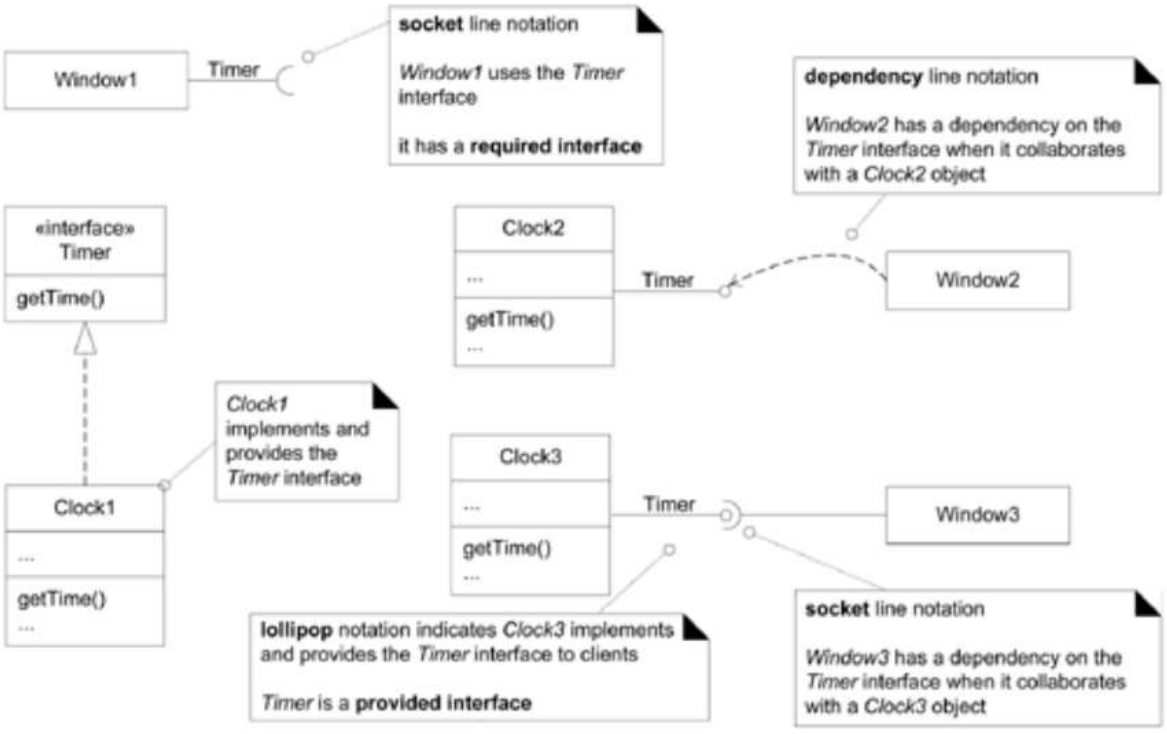
To show the type of dependencythe dependency line can be labeled with keywords or stereotypes.





Interfaces

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



The socket notation is new to UML 2. It's useful to indicate "Class X requires (uses) interface Y" without drawing a line pointing to interface Y.

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.

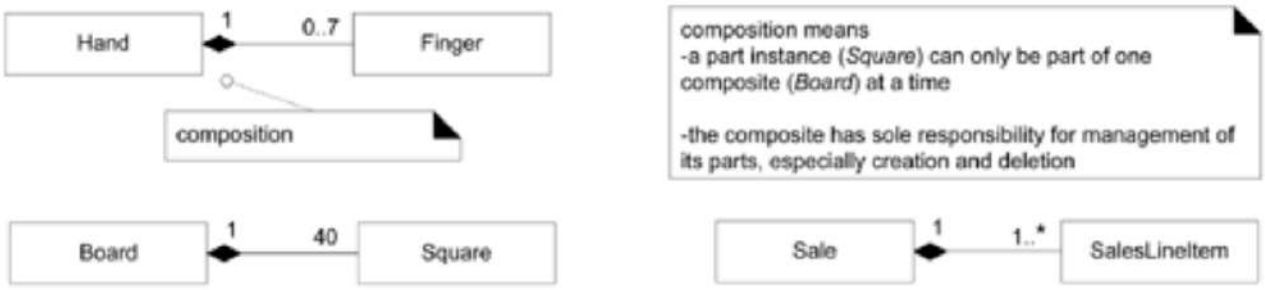
Related to this constraint is that **if the composite is destroyed**, its parts must either be destroyed, or attached to another composite

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"

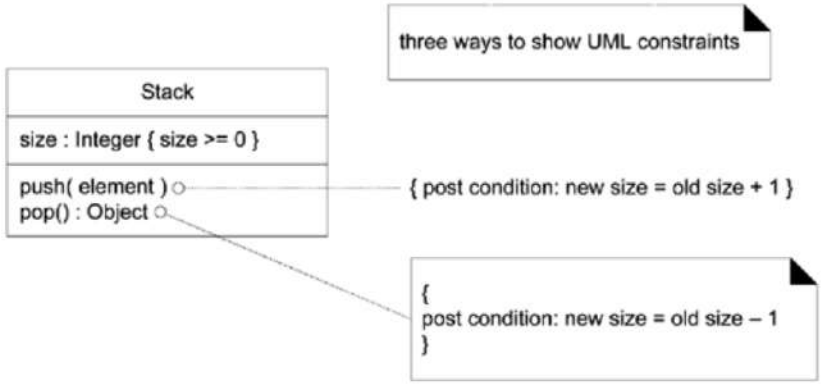


Constraints

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)

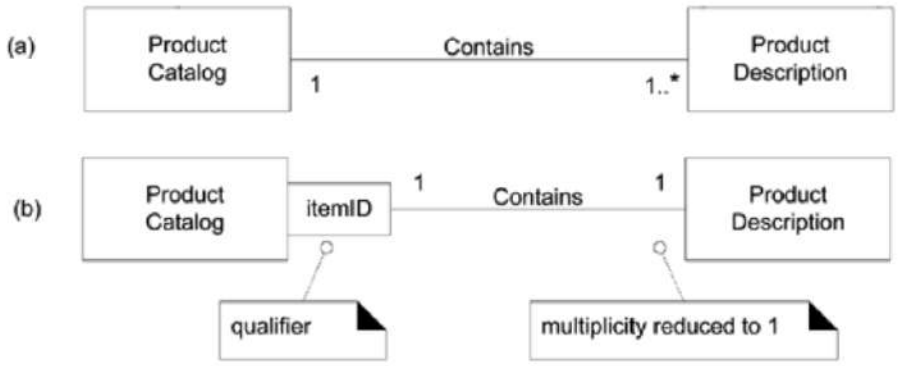


Qualified Association

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

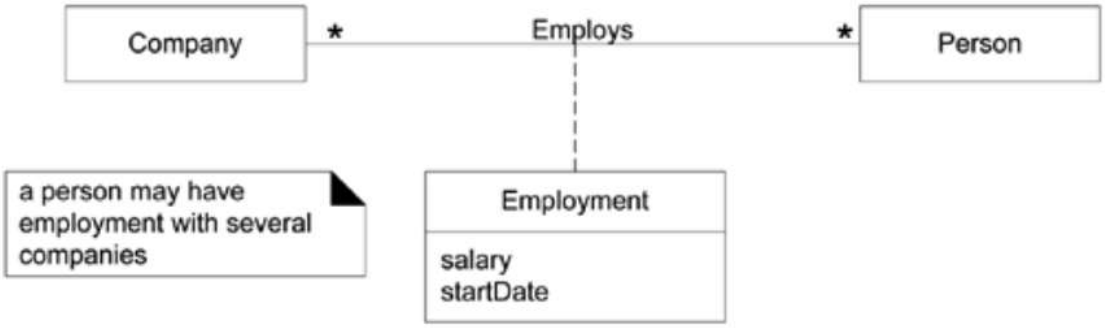


There's one subtle point about qualified associations: the change in multiplicity.

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.

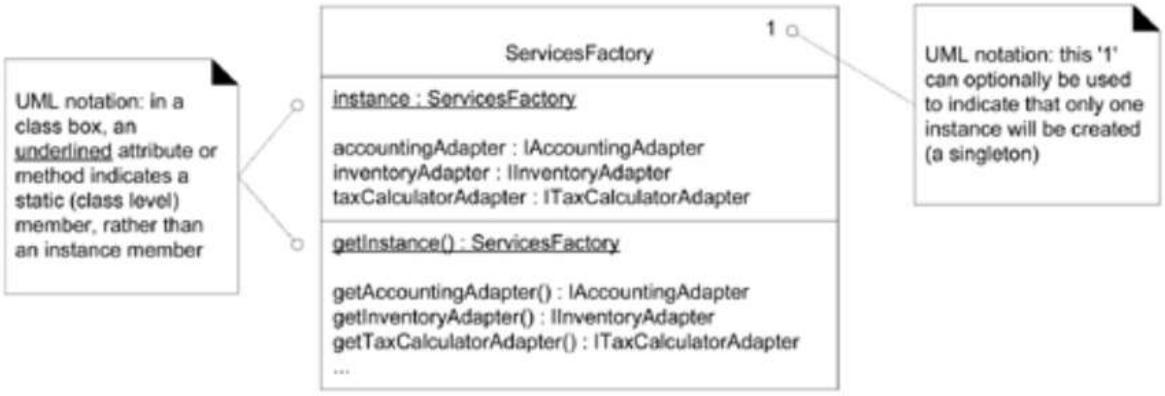
Association Class

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

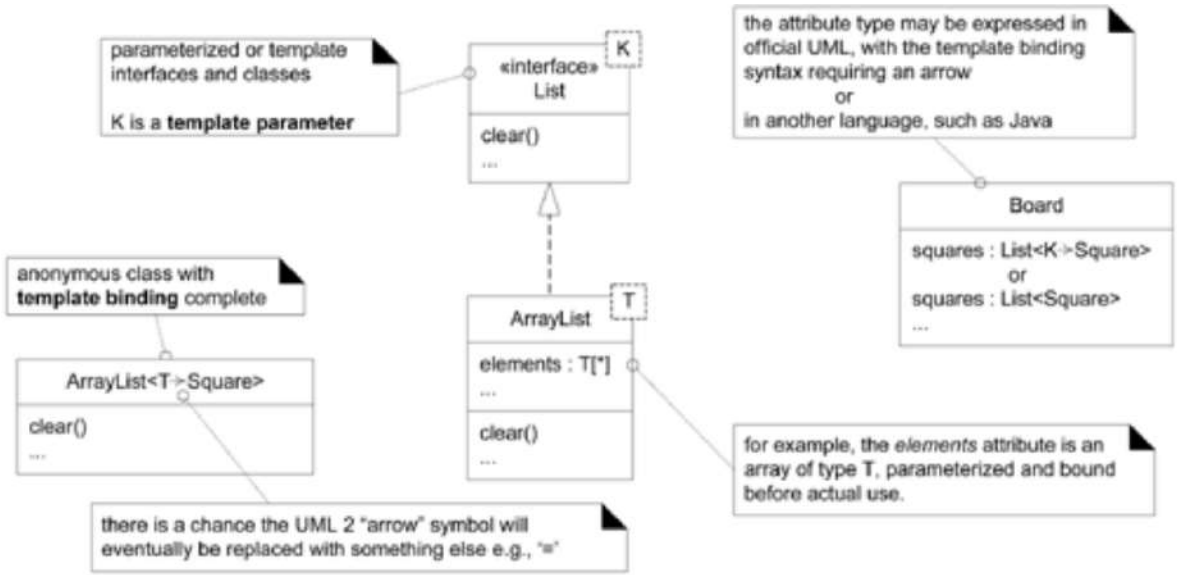
An implication of the Singleton pattern is that **there is only One instance of a class instantiated** never two (a "singleton" instance) In a UML diagram, such a class **can be marked with a '1' in the upper right corner** of the name compartment.



Template Classes and Interfaces

Many languages (Java, C++, ...) support templatized types , also known as templates, parameterized types , and generics. They are most commonly used for the element type of collection classes, such as the elements of lists and maps. For example, in Java, suppose that a Board software object holds a List (an interface for a kind of collection) of many Squares . And, the concrete class that implements the List interface is an ArrayList

```
public class Board
{
     private List<Square> squares = new ArrayList<Square>();
// ...
}
```

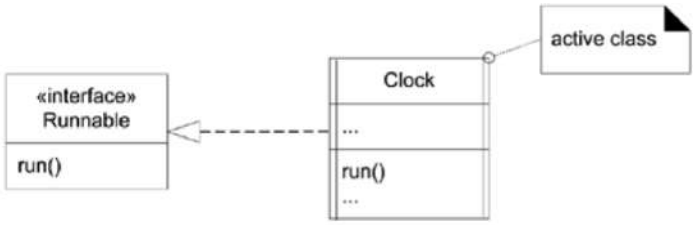


User-Defined Compartments

In addition to common predefined compartments class compartments such as name, attributes, and operations, **user-defined compartments can be added** to a class box.

Active Class

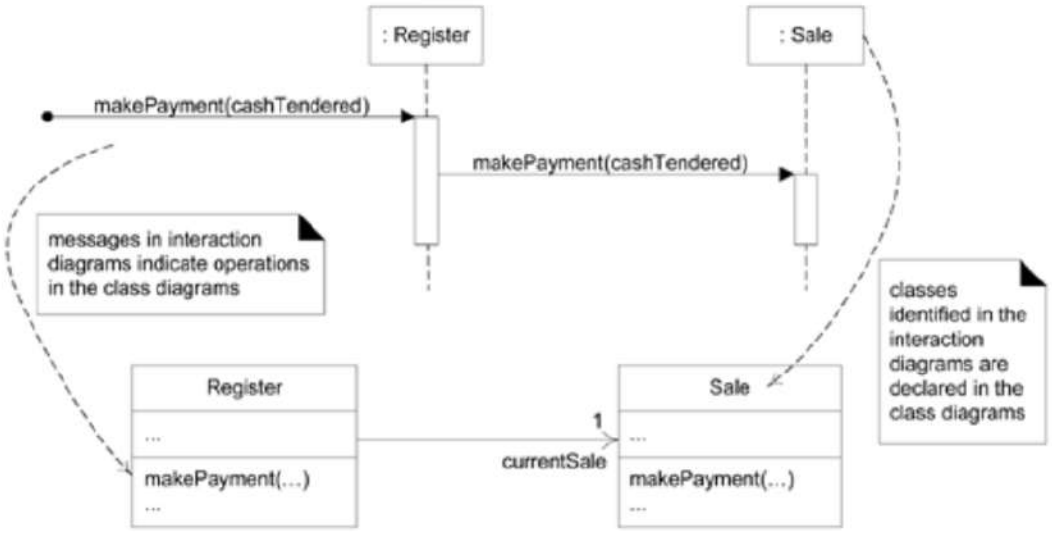
An active object **runs on and controls its own thread of execution**. Not surprisingly, the class of an active object is an active class. In the UML, it may be **shown with double vertical lines on the left and right sides** of the class box



Relationship Between Interaction and Class Diagrams

When interaction diagrams are drawn, a **set of classes and their methods emerge** from the creative design process of dynamic object modeling.

Like if we started with the makePayment sequence diagram, we see that a **Register and Sale class definition in a class diagram can be obviously derived**.



Thus, from interaction diagrams the definitions of class diagrams can be generated.

This **suggests a linear ordering** of drawing interaction diagrams before class diagrams, **but in practice**, especially when following the agile modeling practice of models in parallel, these complementary dynamic and static views **are drawn concurrently**.

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

Developing a Domain Class Diagram: the POS DCD

1) Identify **software classes:**

These can be found by scanning all the interaction diagrams and listing the classes mentioned.

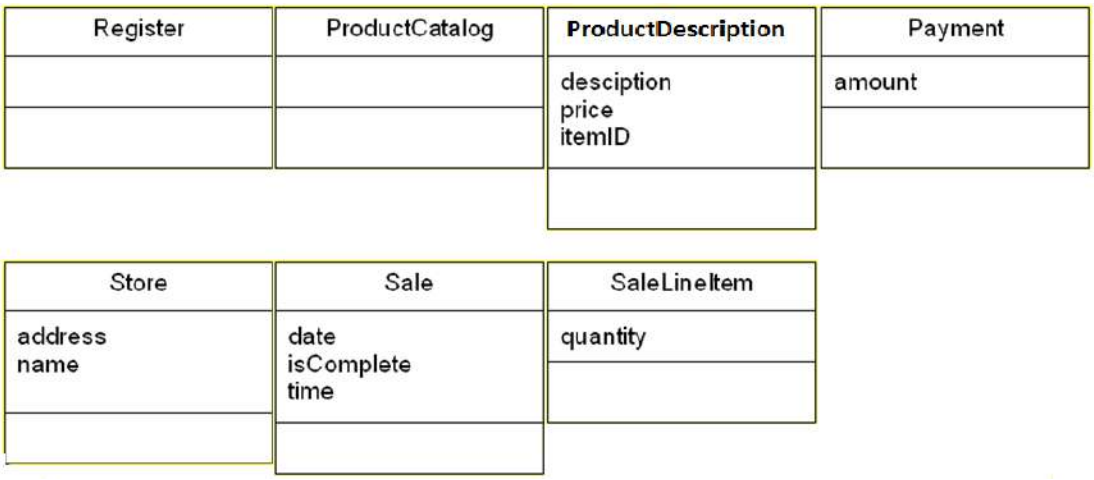
For the POS application, these are:

```
Register                Sale
ProductCatalog          ProductDescription
Store                   SalesLineItem
Payment
```

2) Begin drawing a **Class Diagram**

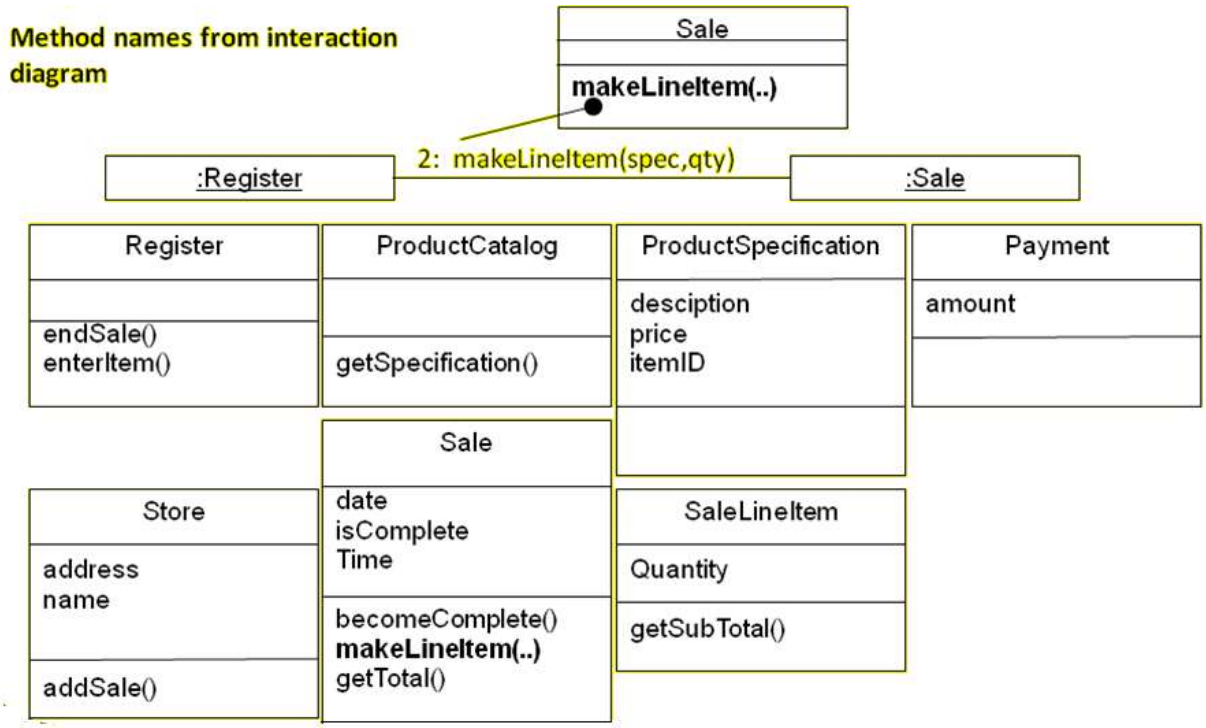
3) Include the **attributes from the domain model**

Draw a Class Diagram by including attributes identified in previous conceptual model



4) Add **method names**

This can be done by analyzing the interaction diagram (communication diagram)



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.

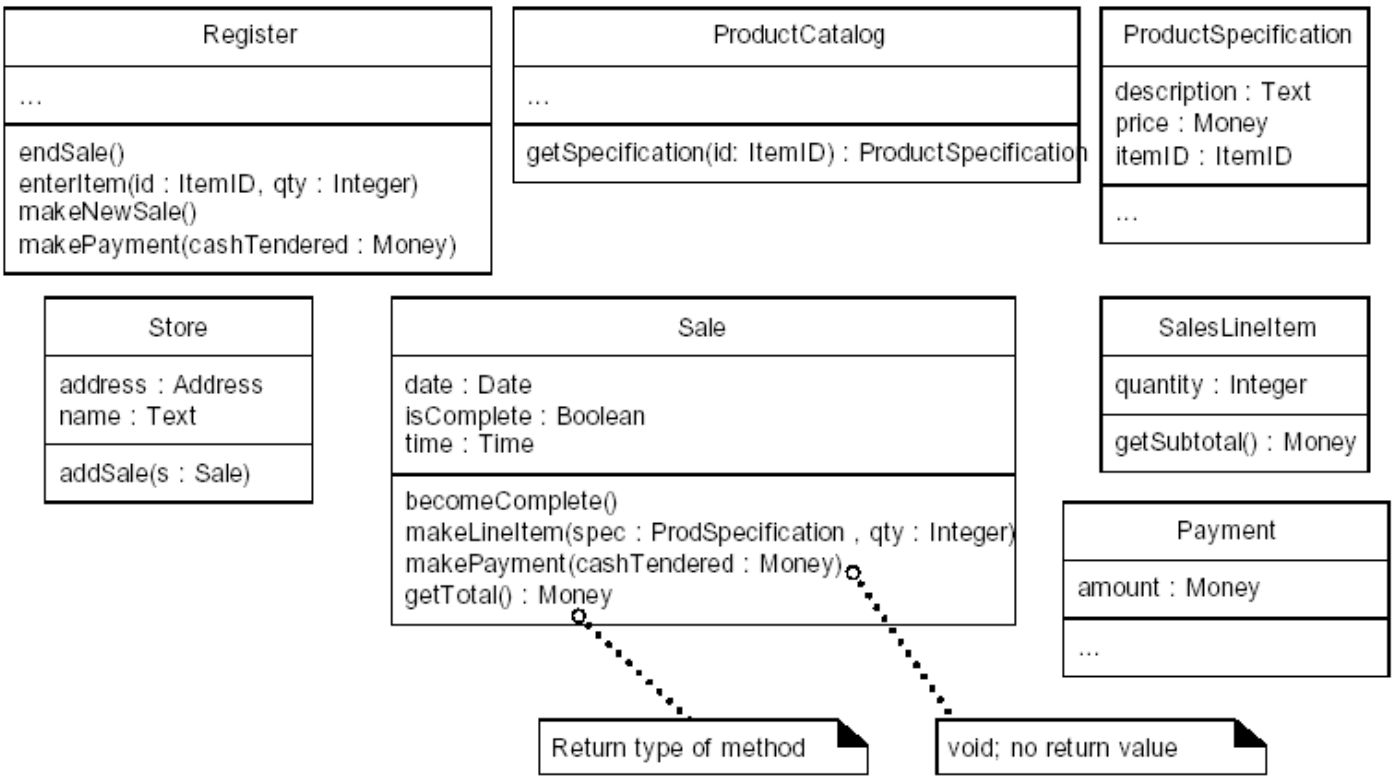
Adding More Type- Information

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.

Adding Type Information



5) Add associations and navigability

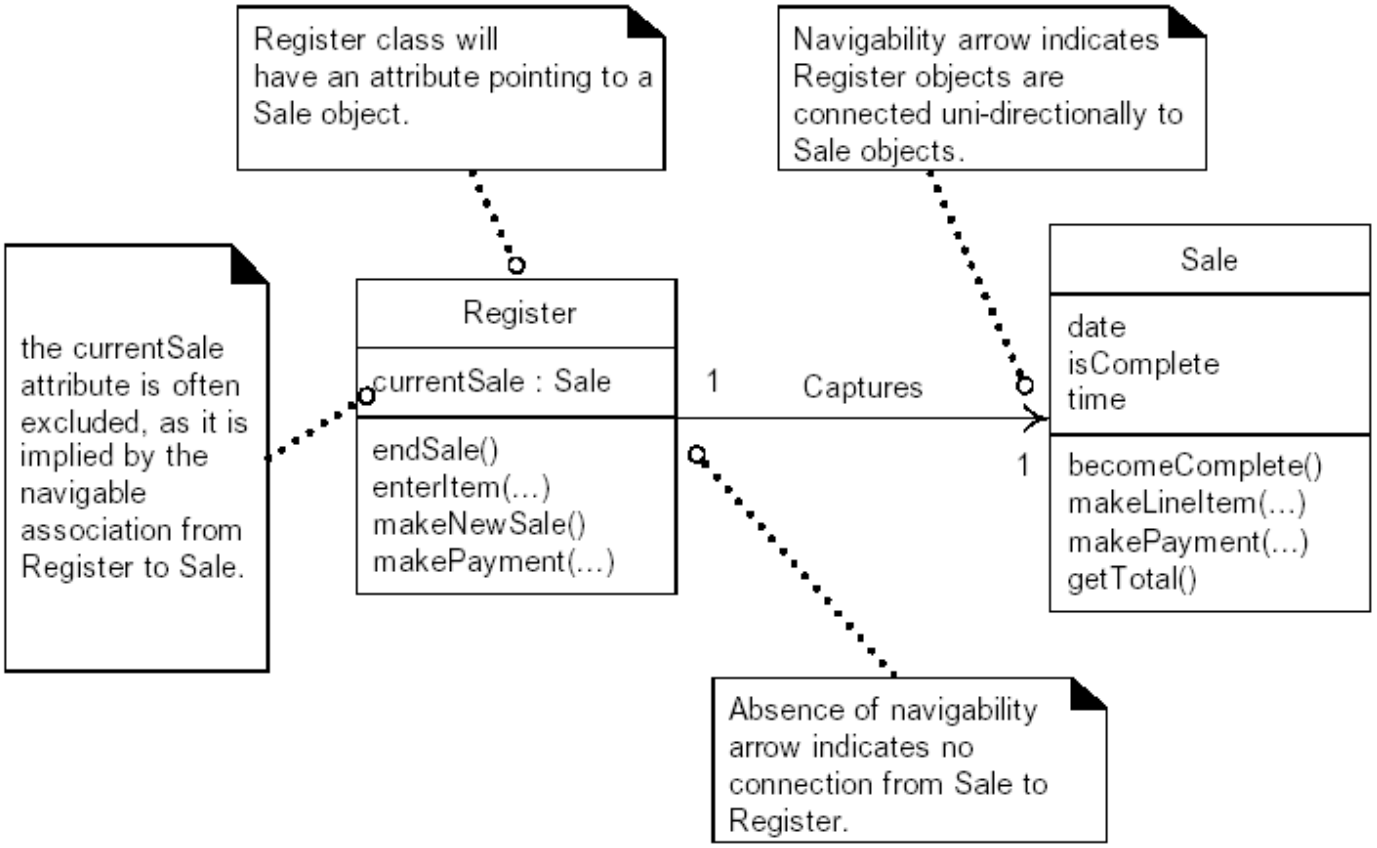
Each end of an association is called a role, and in the DCDs the role may be decorated with a navigability arrow.

Navigability is a property of the role that indicates that it is possible to navigate uni-directionally across the association from objects of the source to target class.

Navigability implies visibility—**usually attribute visibility**

The usual interpretation of an association with navigability arrow is attribute visibility from the source to target class.

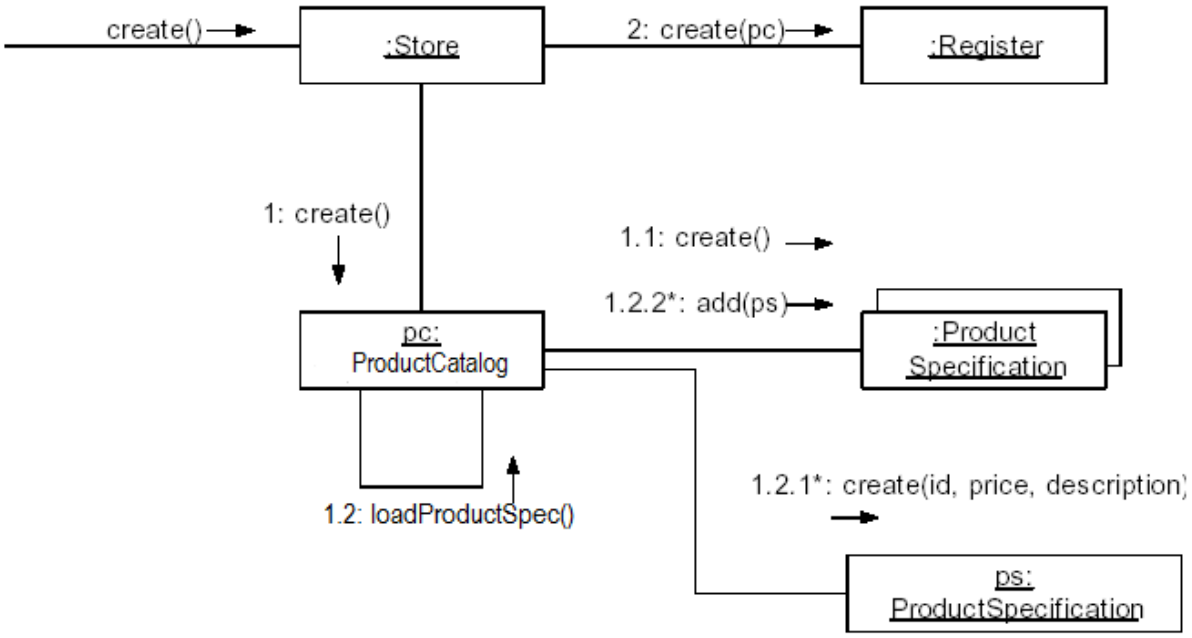
For instance POST class will define an attribute that references a Sale instance.



The usual interpretation of an association with navigability arrow is attribute visibility from the source to target class.

For instance POST class will define an attribute that references a Sale instance.

Common situations suggesting a need to define an association with navigability adornment from A to B



A sends a message to B

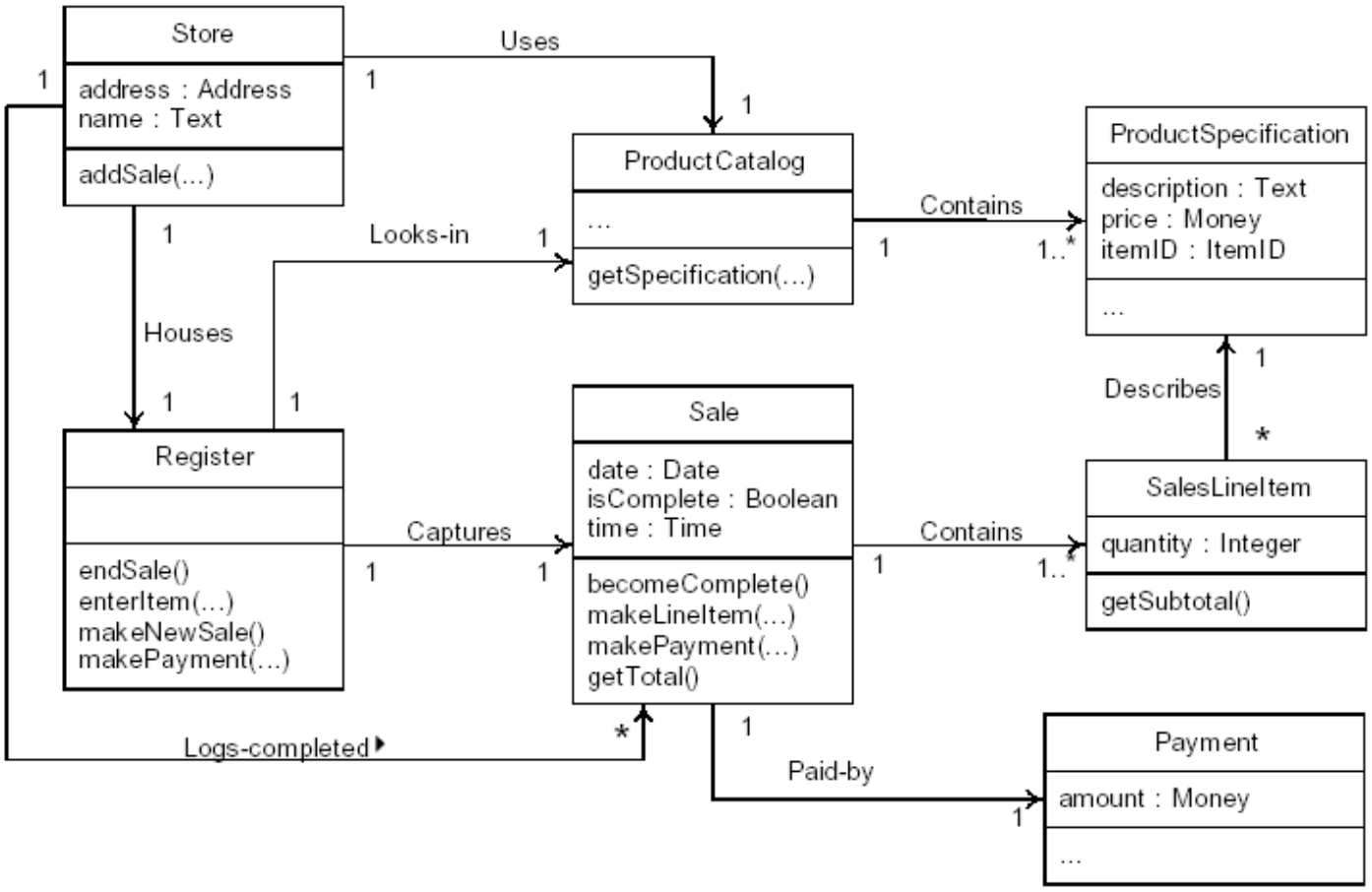
A creates an instance of B

A needs to maintain a connection to B

From the communication diagram we see that the Store should have an ongoing connection to the POST and `ProductCatalogue` instance that is created.

Similarly the `ProductCataloge` needs an ongoing to the collection of `ProductDescriptions` it created

Thus the creator of another object very typically requires an ongoing connection to it.

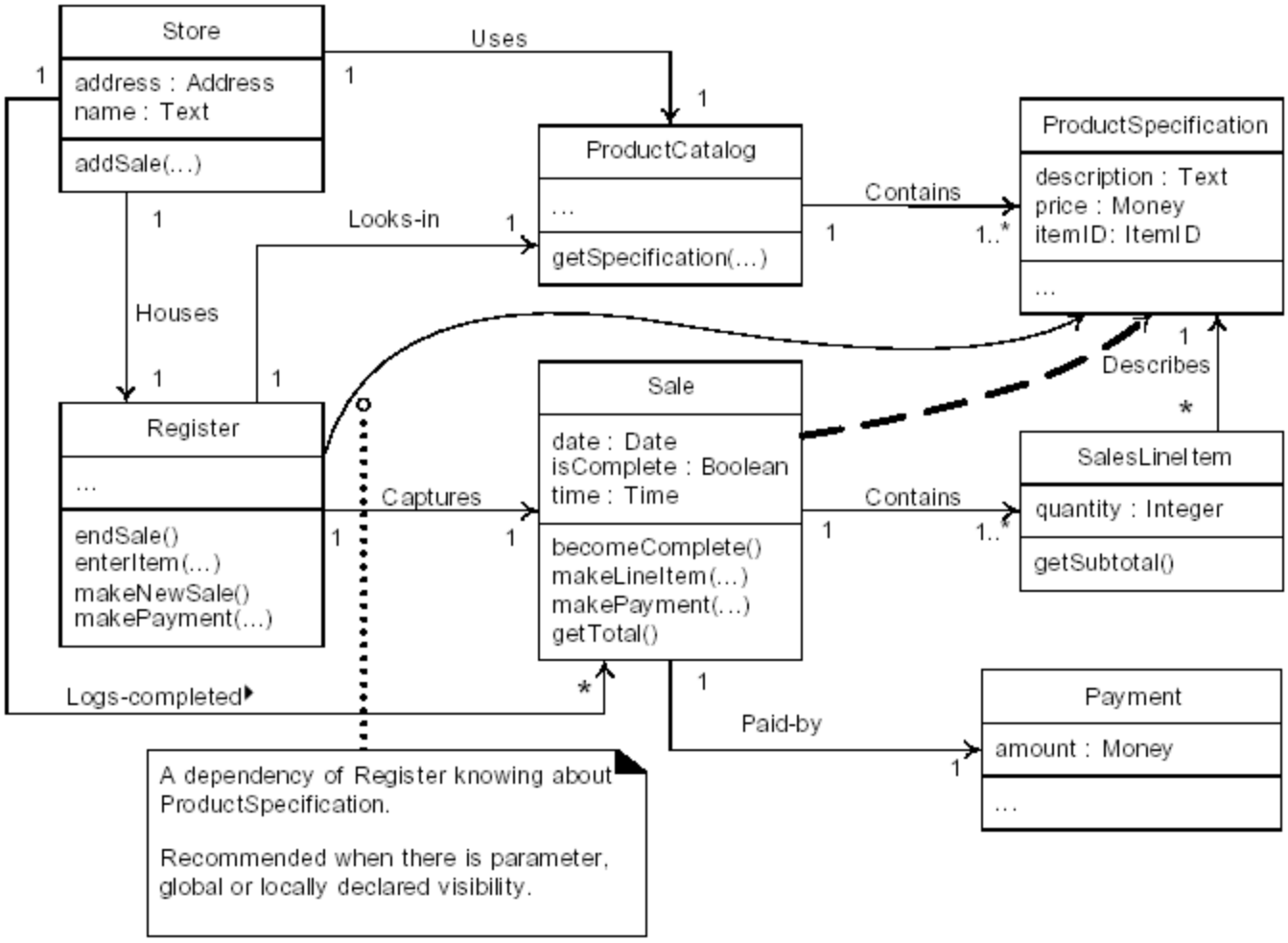


Associations with Navigability adornments

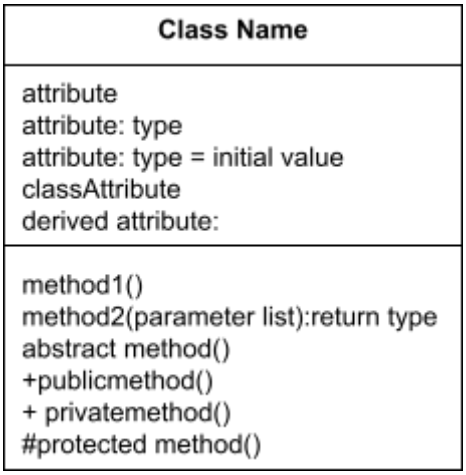
Adding Dependency Relationship

The UML includes general dependency relationship which indicates that **one element has knowledge of another element**. It is illustrated with **dashed arrow** line.

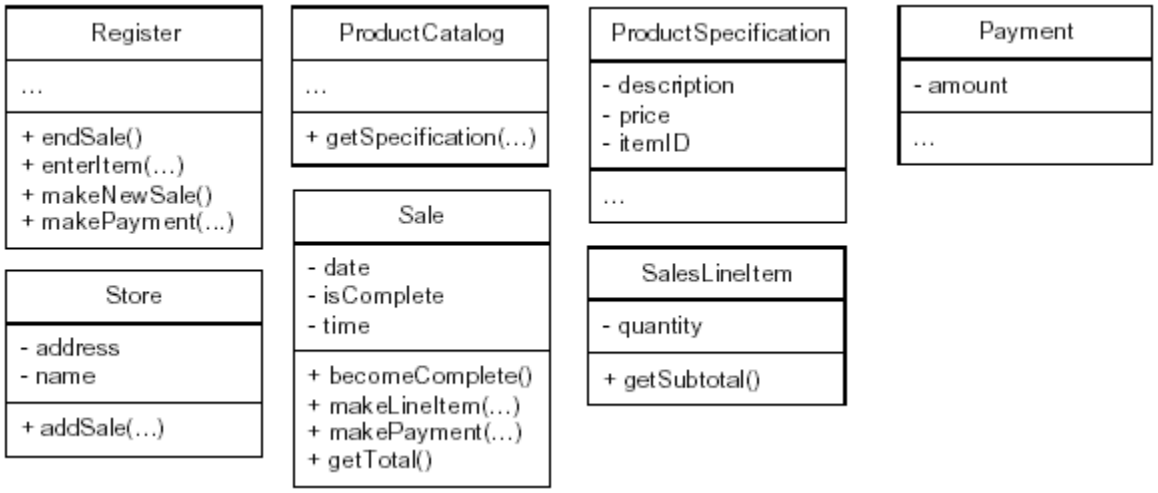
In Class Diagram its significance is to depict non-attribute visibility between classes.



Notations for members



Class Member details notation



Member details in POST class diagram