Chapter 3: Object Oriented Design

Object Oriented Design

The boundaries between analysis and design are fuzzy, although the focus of each is quite distinct. In analysis, the focus is to fully analyze the problem at hand and to model the world by discovering the classes and objects that form the vocabulary of the problem domain. In design, we invent the abstractions and mechanisms in our models that provide the design of the solution to be built.
An object contains encapsulated data and procedures grouped together to represent an entity. The 'object interface', how the object can be interacted with, is also defined. An object-oriented program is described by the interaction of these objects.

Object-oriented design is the discipline of defining the objects and their interactions to solve a problem that was identified and documented during object-oriented analysis.
Object-oriented design is the process of planning a system of interacting objects for the purpose of solving a software problem. It is one approach to software design.

From Requirements/Analysis to Design

A set of requirements-oriented artifacts (and thought) inspire design-oriented artifacts.

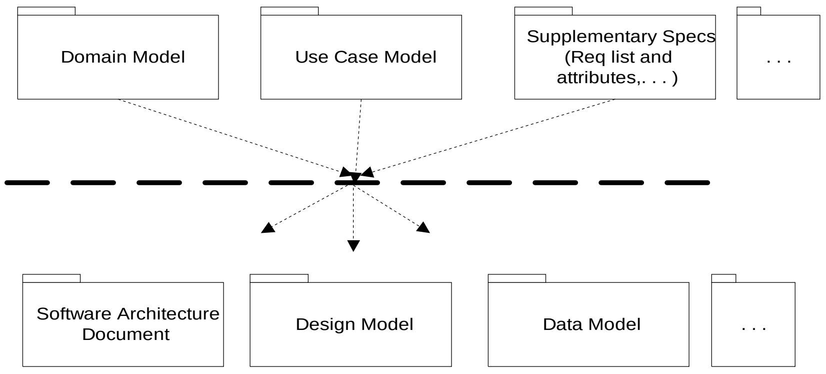

Domain Model
Use Case Model
Supplementary Specs (Req list and attributes,. . .)
. . .
Software Architecture Document
Design Model
Data Model
. . .

Fig: Showing transition from Analysis models to Design Models

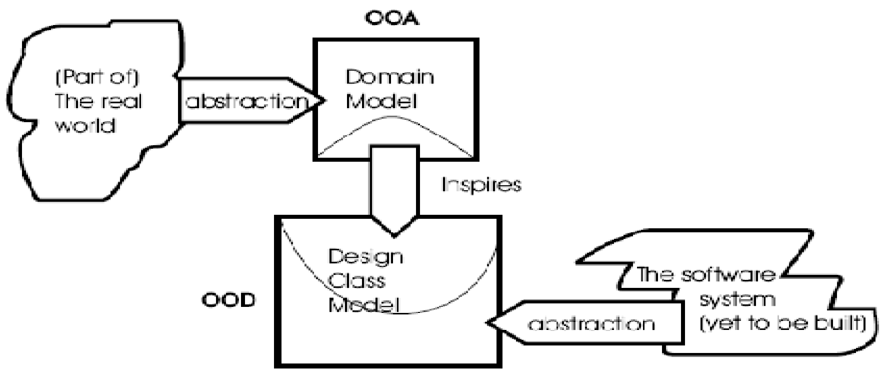


Fig: The phases of OOAD and how the transition from OOA to OOD works

Input (sources) for object-oriented design

The input for object-oriented design is provided by the output of object-oriented analysis. Realize that an output artifact does not need to be completely developed to serve as input of object-oriented design; analysis and design may occur in parallel, and in practice the results of one activity can feed the other in a short feedback cycle through an iterative process. Both analysis and design can be performed incrementally, and the artifacts can be continuously grown instead of completely developed in one shot.
Some typical input artifacts for object-oriented design are:

- **Conceptual model:** Conceptual model is the result of object-oriented analysis, it captures concepts in the problem domain. The conceptual model is explicitly chosen to be independent of implementation details, such as concurrency or data storage.
- **Use case:** Use case is a description of sequences of events that, taken together, lead to a system doing something useful. Each use case provides one or more scenarios that convey how the system should interact with the users called actors to achieve a specific business goal or function. Use case actors may be end users or other systems. In many circumstances use cases are further elaborated into use case diagrams. Use case diagrams are used to identify the actor (users or other systems) and the processes they perform.
- **System Sequence Diagram:** System Sequence diagram (SSD) is a picture that shows, for a particular scenario of a use case, the events that external actors generate, their order, and possible inter-system events.
- **User interface documentations** (if applicable): Document that shows and describes the look and feel of the end product's user interface. It is not mandatory to have this, but it helps to visualize the end-product and therefore helps the designer.
- **Relational data model** (if applicable): A data model is an abstract model that describes how data is represented and used. If an object database is not used, the relational data model should usually be created before the design, since the strategy chosen for object-relational mapping is an output of the OO design process. However, it is possible to develop the relational data model and the object-oriented design artifacts in parallel, and the growth of an artifact can stimulate the refinement of other artifacts.

Designing concepts

- ***Defining objects, creating class diagram from conceptual diagram:*** *Usually map entity to class.*
- ***Identifying attributes****.*
- ***Use design patterns*** *(if applicable): A design pattern is not a finished design, it is a description of a solution to a common problem, in a context[1]. The main advantage of using a design pattern is that it can be reused in multiple applications. It can also be thought of as a template for how to solve a problem that can be used in many different situations and/or applications. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved.*
- ***Define application framework*** *(if applicable): Application framework is a term usually used to refer to a set of libraries or classes that are used to implement the standard structure of an application for a specific operating system. By bundling a large amount of reusable code into a framework, much time is*

saved for the developer, since he/she is saved the task of rewriting large amounts of standard code for each new application that is developed.

- ***Identify persistent objects/data*** *(if applicable): Identify objects that have to last longer than a single runtime of the application. If a relational database is used, design the object relation mapping.*
- ***Identify and define remote objects*** *(if applicable).*

Output (deliverables) of object-oriented design

- **Sequence Diagrams/Collaboration:** Extend the System Sequence Diagram to add specific objects that handle the system events. A sequence diagram shows, as parallel vertical lines, different processes or objects that live simultaneously, and, as horizontal arrows, the messages exchanged between them, in the order in which they occur.
- **Design Class diagram:** A class diagram is a type of static structure UML diagram that describes the structure of a system by showing the system's classes, their attributes, and the relationships between the classes. The messages and classes identified through the development of the sequence diagrams can serve as input to the automatic generation of the global class diagram of the system.

How do developers design objects?
Here are three ways:

1. ***Code.*** *Design-while-coding (Java, C#, …), ideally with power tools such as* **refactorings**. *From mental model to code.*
2. ***Draw, then code.*** *Drawing some UML on a whiteboard or UML CASE tool, then switching to #1 with a text-strong IDE (e.g., Eclipse or Visual Studio).*
3. ***Only draw.*** *Somehow, the tool generates everything from diagrams. Many a dead tool vendor has washed onto the shores of this steep island. "Only draw" is a misnomer, as this still involves a text programming language attached to UML graphic elements.*

Designing Objects:
What are Static and Dynamic Modeling?

There are two kinds of object models: dynamic and static. Dynamic models, such as UML interaction diagrams sequence diagrams or communication diagrams), help design the logic, the behavior of the code or the method bodies. They tend to be the more interesting, difficult, important diagrams to create. Static models, such as UML class diagrams, help design the definition of packages, class names, attributes, and method signatures (but not method bodies).

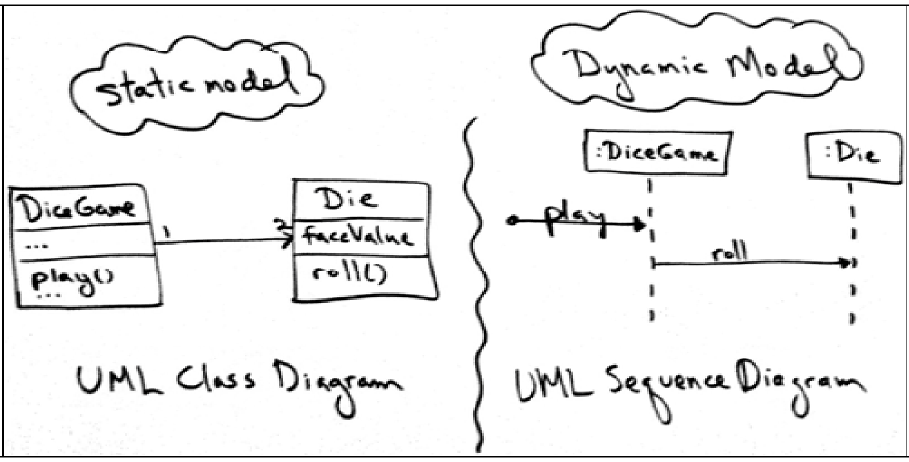


Fig: Static and dynamic UML diagrams for object modeling.

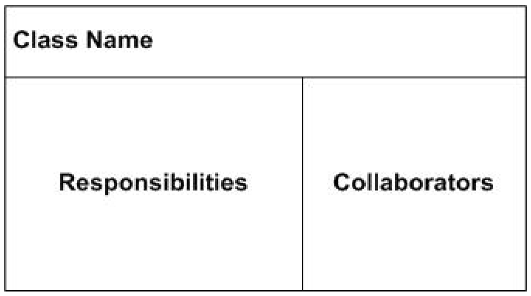
There's a relationship between static and dynamic modeling and the agile modeling practice of create models in parallel: Spend a short period of time on interaction diagrams (dynamics), then switch to a wall of related class diagrams (statics).

Object Design Techniques:
CRC Cards

A popular text-oriented modeling technique is Class Responsibility Collaboration (**CRC**) cards, created by the agile, influential minds of Kent Beck and Ward Cunningham (also founders of the ideas of XP and design patterns). **Class Responsibility Collaboration** (**CRC**) **cards** are a brainstorming tool used in the design of object-oriented software.
CRC cards are paper index cards on which one writes the responsibilities and collaborators of classes. Each card represents one class. A CRC modeling session involves a group sitting around a table, discussing and writing on the cards as they play "what if" scenarios with the objects, considering what they must do and what other objects they must collaborate with.

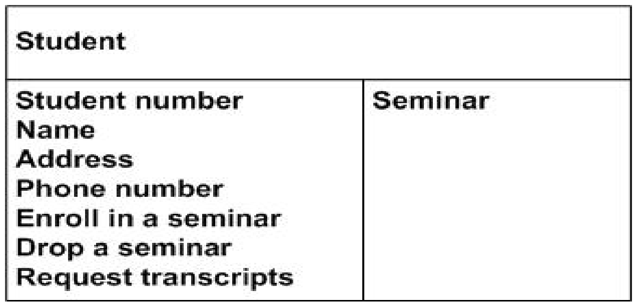
CRC cards are usually created from index cards on which are written:

1. The class name
2. Its Super and Sub classes (if applicable)
3. The responsibilities of the class.
4. The names of other classes with which the class will collaborate to fulfill its responsibilities.
5. Author



Class Name	
Responsibilities	Collaborators

Fig: CRC card Format



Student	
Student number Name Address Phone number Enroll in a seminar Drop a seminar Request transcripts	Seminar

Fig: CRC card Example

Class

A Class represents a collection of similar objects. Objects are things of interest in the system being modeled. They can be a person, place, thing, or any other concept important to the system at hand. The Class name appears across the top of the CRC card.

Responsibility

A Responsibility is anything that the class knows or does. These responsibilities are things that the class has knowledge about itself, or things the class can do with the knowledge it has.

For example, a person class might have knowledge (and responsibility) for its name, address, and phone number. In another example an automobile class might have knowledge of its size, its number of doors, or it might be able to do things like stop and go. The Responsibilities of a class appear along the left side of the CRC card.

Collaborator

A Collaborator is another class that is used to get information for, or perform actions for the class at hand. It often works with a particular class to complete a step (or steps) in a scenario. The Collaborators of a class appear along the right side of the CRC card.

How do you create CRC models

- ***Find classes***. *Finding classes is fundamentally an analysis task because it deals with identifying the building blocks for your application. A good rule of thumb is that you should look for the three-to-five main classes right away, such as Student, Seminar, and Professor in Figure .*
- ***Find responsibilities***. *You should ask yourself what a class does as well as what information you wish to maintain about it. You will often identify a responsibility for a class to fulfill a collaboration with another class.*
- ***Define collaborators***. *A class often does not have sufficient information to fulfill its responsibilities. Therefore, it must collaborate (work) with other classes to get the job done. Collaboration will be in one of two forms: a request for information or a request to perform a task.*
- ***Move the cards around***. *To improve everyone's understanding of the system, the cards should be placed on the table in an intelligent manner. Two cards that collaborate with one another should be placed close together on the table, whereas two cards that don't collaborate should be placed far apart*

Realization of Use case

A use-case realization represents how a use case will be implemented in terms of collaborating objects. This artifact can take various forms. It can include, for example, a textual description (a document), class diagrams of participating classes and subsystems, and interaction diagrams (communication and sequence diagrams) that illustrate the flow of interactions between class and subsystem instances.

The reason for separating the use-case realization from its use case is that doing so allows the use cases to be managed separately from their realizations. This is particularly important for larger projects or families of systems where the same use cases can be designed differently in different products within the product family. Consider the case of a family of telephone switches which have many use cases in common, but which design and implement them differently according to product positioning, performance and price.

For larger projects, separating the use case and its realization allows changes to the design of the use case without affecting the baseline use case itself.

In a model, a use-case realization is represented as a UML collaboration that groups the diagrams and other information (such as textual descriptions) that form part of the use-case realization.

UML diagrams that support use-case realizations can be produced in an analysis context, a design context, or both, depending on the needs of the project. For each use case in the use-case model, there can be a use-case realization in the analysis/design model with a realization relationship to the use case. In UML this is shown as a dashed arrow, with an arrowhead like a generalization relationship, indicating that a realization is a kind of inheritance, as well as a dependency.

Use case realization : *Make sure that the sequence diagrams realize (show) the behavior outlined in the use cases and assign behavior to classes in the class diagram. Together, these three continue to evolve and sharpen each other. This can be shown as following fig:*

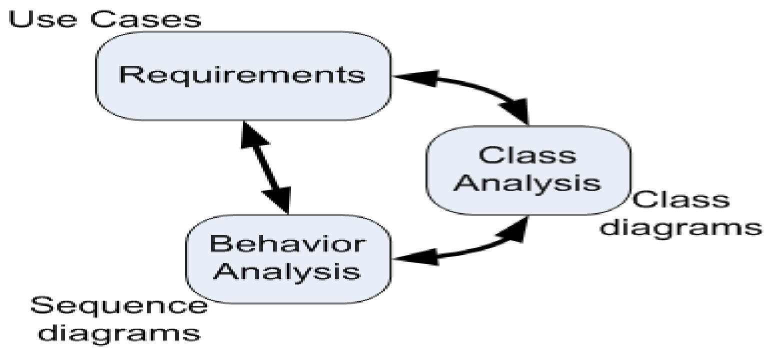
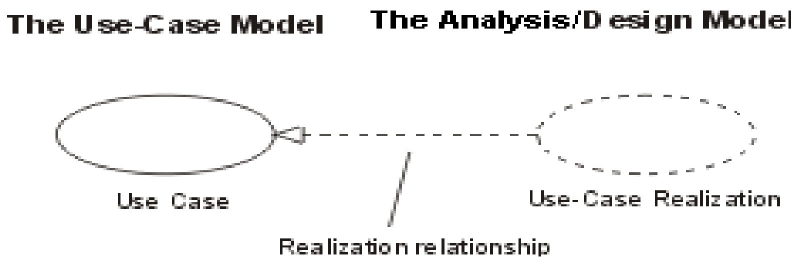


Fig: Use case realization process



A use-case realization in the design can be traced to a use case in the use-case model.

Class Diagrams Owned by a Use-Case Realization

For each use-case realization there can be one or more class diagrams depicting its participating classes. A class and its objects often participate in several use-case realizations. It is important while designing to coordinate all the requirements on a class and its objects that different use-case realizations can have. The figure below shows an analysis class diagram for the realization of the Withdraw Cash Item use case.

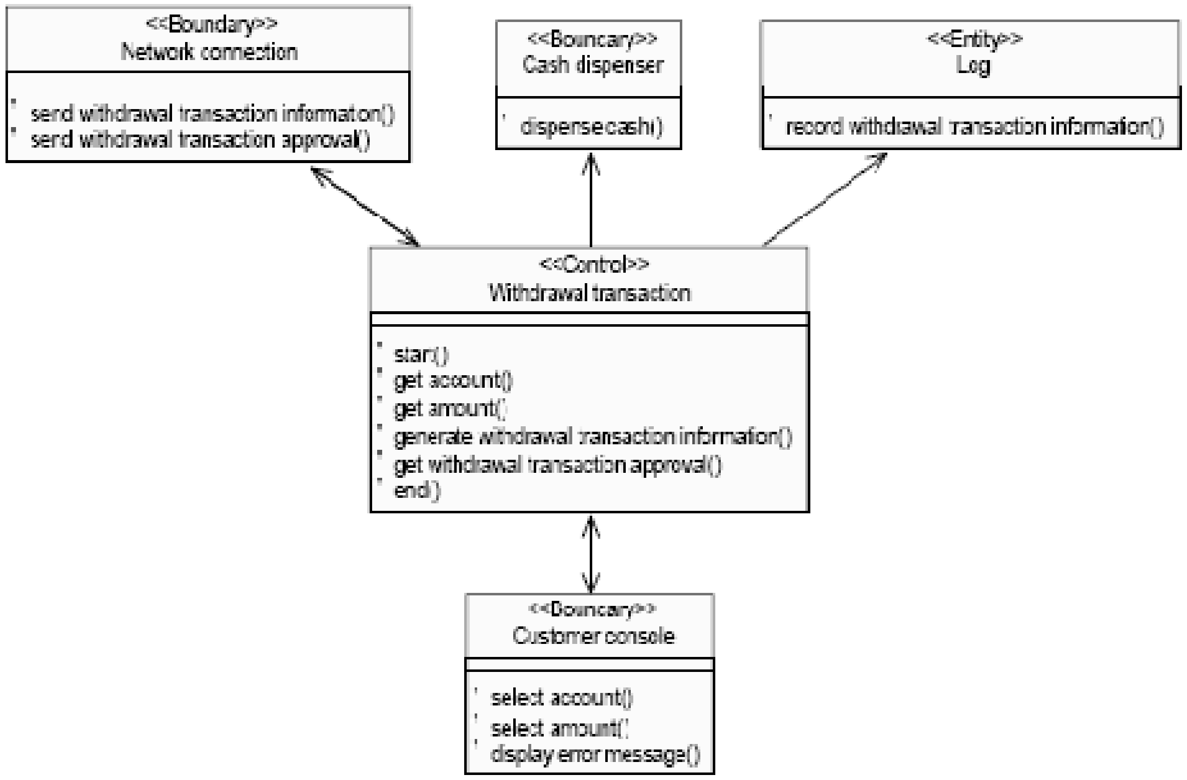


Fig: Use case realization of Cash Withdraw Use case

The use case Receive Deposit Item and its analysis-level class diagram.

Communication and Sequence Diagrams Owned by a Use-Case Realization

For each use-case realization there can be one or more interaction diagrams depicting its participating objects and their interactions. There are two types of interaction diagrams: sequence diagrams and communication diagrams. They express similar information, but show it in different ways. Sequence diagrams show the explicit sequence of messages and are better when it is important to visualize the time ordering of messages, whereas communication diagrams show the communication links between objects and are better for understanding all of the effects on a given object and for algorithm design.

Realizing use cases through interaction diagrams helps to keep the design simple and cohesive. Assigning responsibilities to classes on the basis of what the use-case scenario explicitly requires encourages the design to contain the following:

- Only the functionality actually used in support of a use case scenario,
- Functionality that can be tested through an associated test case,
- Functionality that is more easily traceable to requirements and changes,
- Explicitly declared class dependencies that are easier to manage.

These factors help improve the overall quality of the system.

Example, the simple use case for a automobile navigation system below.

GPS Navigate to Address Use Case

1. *Driver starts navigational system*
   System prompts for whether the driver needs help finding an address, intersection, or point of interest
2. *Driver selects address*
   System prompts for address #, street, city
3. *Driver provides address info*
   System computes location of address
   System computes car's current location
   System computes route from current location to address location
   System locates digital map based on address location
   System displays appropriate digital map with route information

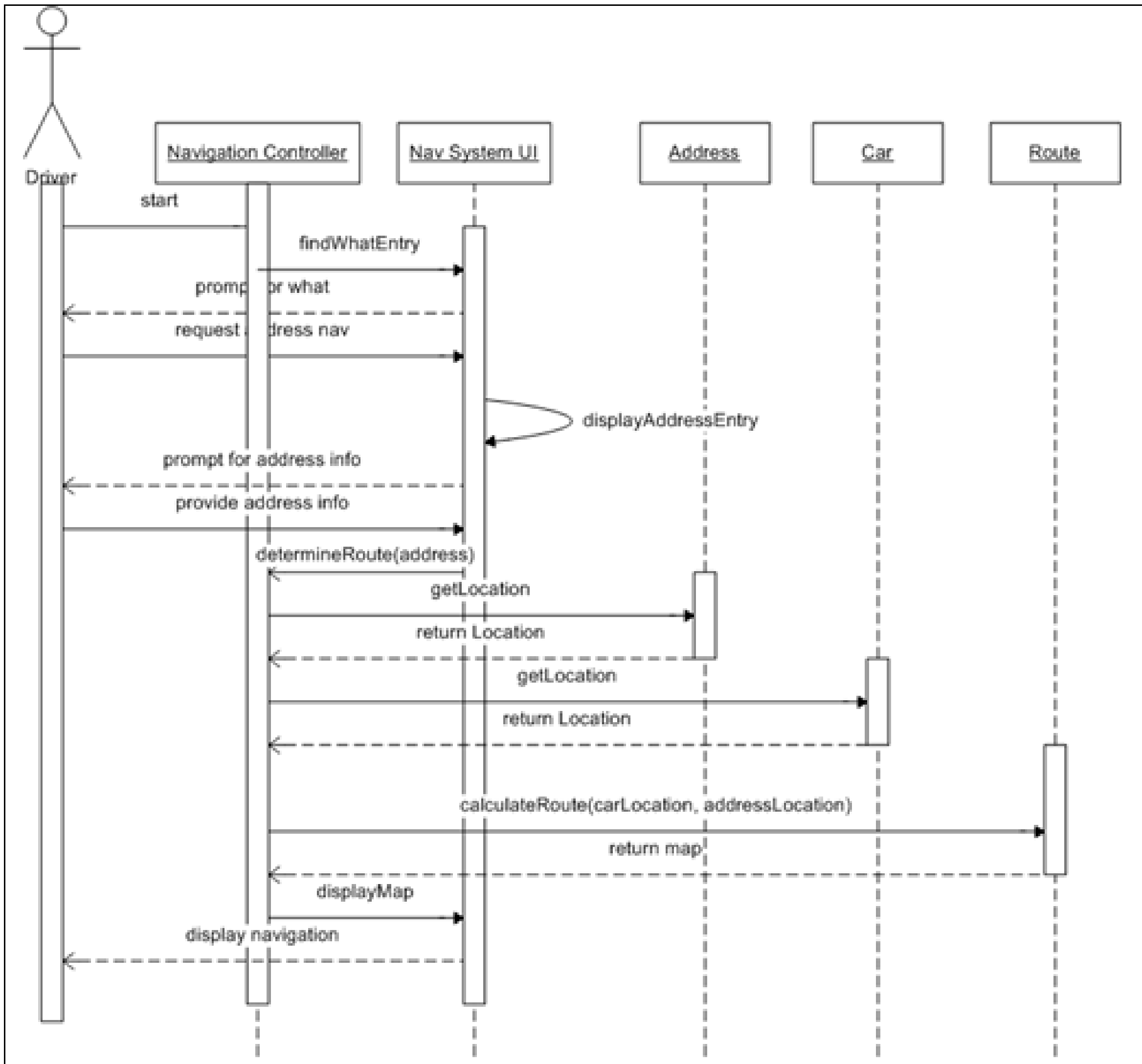


Fig: Realizing Use Cases Automobile navigation System through sequence diagram

UML Interaction Diagrams

The UML includes **interaction diagrams** to illustrate how objects interact via messages. They are used for **dynamic object modeling**. There are two common types: sequence and communication interaction diagrams.

Sequence and Communication Diagrams

The term **interaction diagram** is a generalization of two more specialized UML diagram types:

- Sequence diagrams
- communication diagrams

Both can express similar interactions.

A related diagram is the **interaction overview diagram**; it provides a big-picture overview of how a set of interaction diagrams are related in terms of logic and process-flow. However, it's new to UML 2, and so it's too early to tell if it will be practically useful.
Sequence diagrams are the more notationally rich of the two types, but communication diagrams have their use as well, especially for wall sketching.

Sequence diagrams

A sequence diagram shows interaction among objects as a two dimensional chart. The chart is read from top to bottom. The objects participating in the interaction are shown at the top of the chart as boxes attached to a vertical dashed line. Inside the box the name of the object is written with a colon separating it from the name of the class and both the name of the object and the class is underlined.
The objects appearing at the top signify that the object already existed when the use case execution was initiated. However, if some object is created during the execution of the use case and participates in the interaction (e.g. a method call), then the object should be shown at the appropriate place on the diagram where it is created. The vertical dashed line is called the object's lifeline. The lifeline indicates the existence of the object at any particular point of time. The rectangle drawn on the lifetime is called the activation symbol and indicates that the object is active as long as the rectangle exists. Each message is indicated as an arrow between the lifeline of two objects. The messages are shown in chronological order from the top to the bottom. That is, reading the diagram from the top to the bottom would show the sequence in which the messages occur. Each message is labeled with the message name.

Some control information can also be included.
Two types of control information are particularly valuable:

a. ***A condition*** *(e.g. [invalid]) indicates that a message is sent, only if the condition is true.*
b. ***An iteration*** *marker shows the message is sent many times to multiple receiver objects as would happen when a collection or the elements of an array are being iterated. The basis of the iteration can also be indicated e.g. [for every book object].*

Illustrate interactions in a kind of fence format, in which each new object is added to the right,

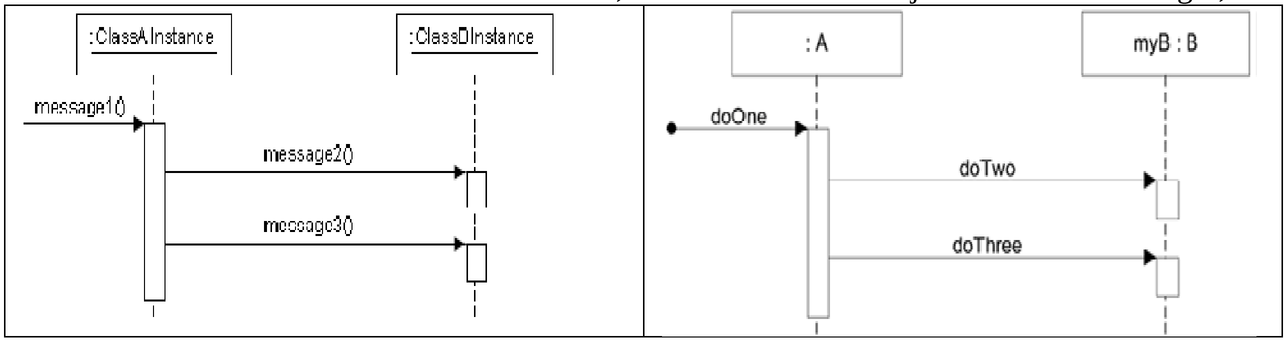


Fig: Example of Sequence Diagram
What might this represent in code? Probably, that class *A* has a method named *doOne* and an attribute of type *B*. Also, that class *B* has methods named *doTwo* and *doThree*. Perhaps the partial definition of class *A* is:

```
public class A
{
private B myB = new B();
public void doOne()
{
myB.doTwo();
myB.doThree();
}
// …
}
```

Basic Sequence diagram Notation:

Lifeline box

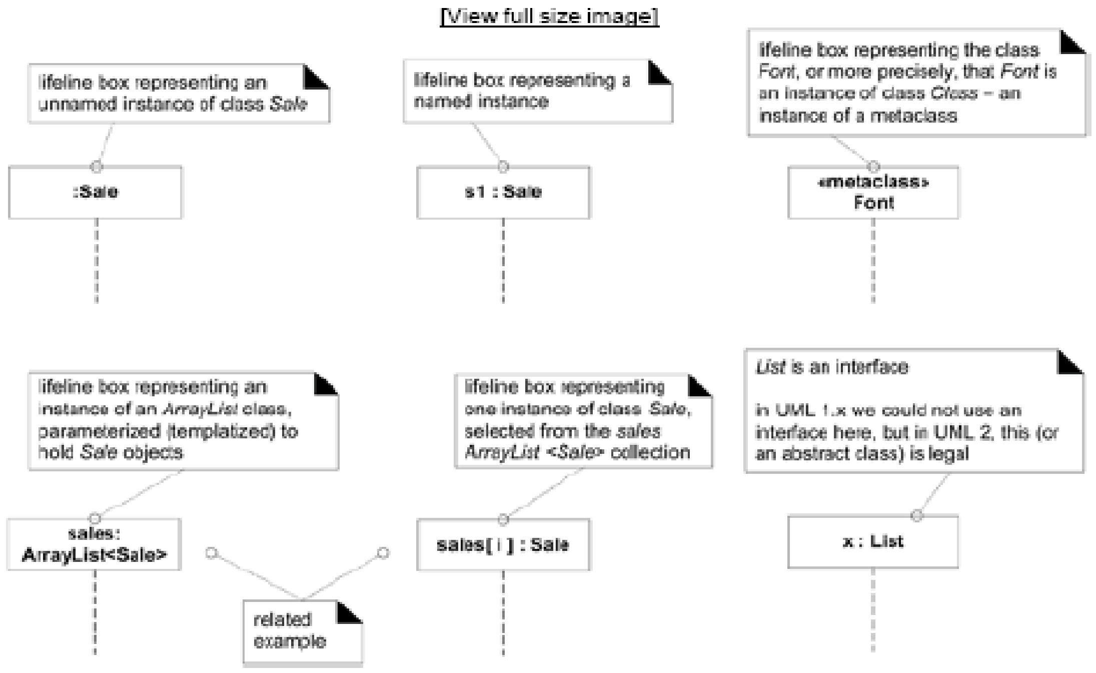

[View full size image]
lifeline box representing an unnamed instance of class *Sale*
:Sale
lifeline box representing a named instance
s1 : Sale
lifeline box representing the class *Font*, or more precisely, that *Font* is an instance of class *Class* – an instance of a metaclass
«metaclass» Font
lifeline box representing an instance of an *ArrayList* class, parameterized (templatized) to hold *Sale* objects
sales: ArrayList<Sale>
related example
lifeline box representing one instance of class *Sale*, selected from the *sales* *ArrayList <Sale>* collection
sales[i] : Sale
List is an interface
in UML 1.x we could not use an interface here, but in UML 2, this (or an abstract class) is legal
x : List

Fig: Notation for lifeline box

Messages

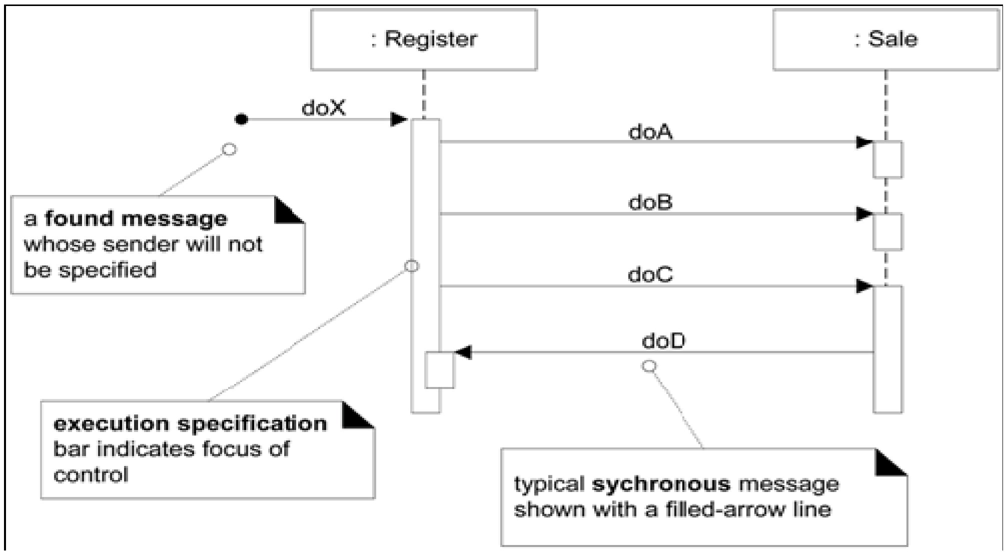

: Register
: Sale
doX
doA
doB
doC
doD
a **found message** whose sender will not be specified
execution specification bar indicates focus of control
typical **sychronous** message shown with a filled-arrow line

Fig: Message and focus of control

Reply or Returns

- Using the message syntax returnVar = message(parameter).
- Using a reply (or return) message line at the end of an activation bar.

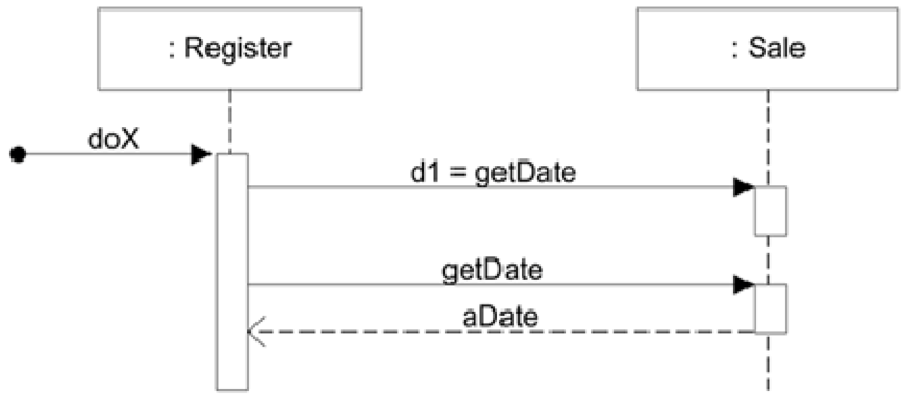
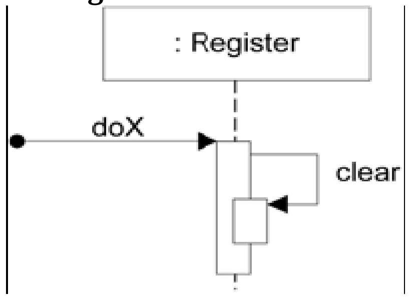
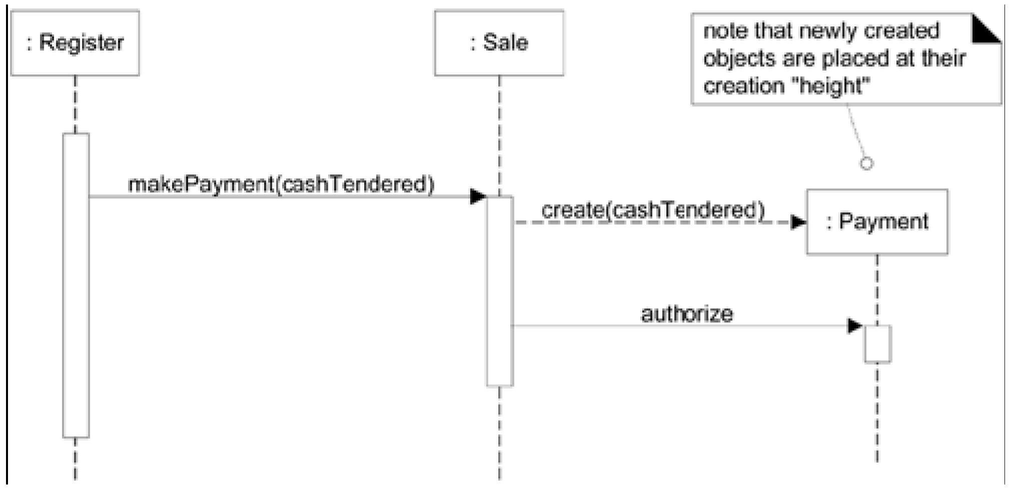


Fig: Two ways to show a return result from a message.

Messages to "self" or "this"



Creation of Instances



: Register
: Sale
note that newly created objects are placed at their creation "height"
makePayment(cashTendered)
create(cashTendered)
: Payment
authorize

Object Lifelines and Object Destruction

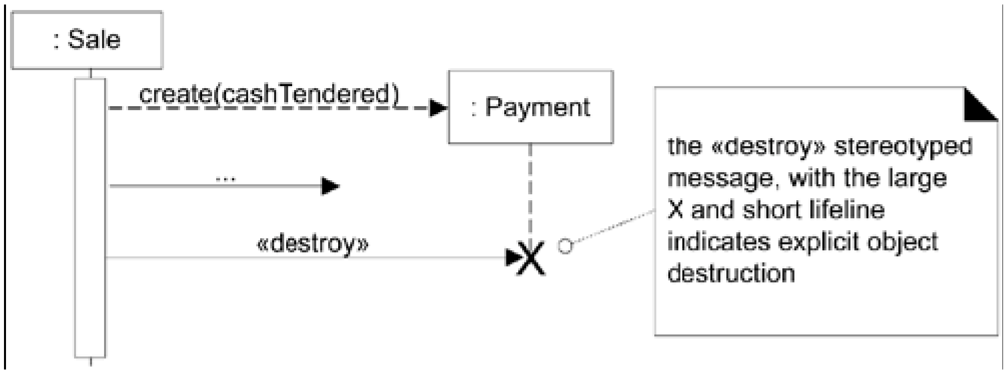
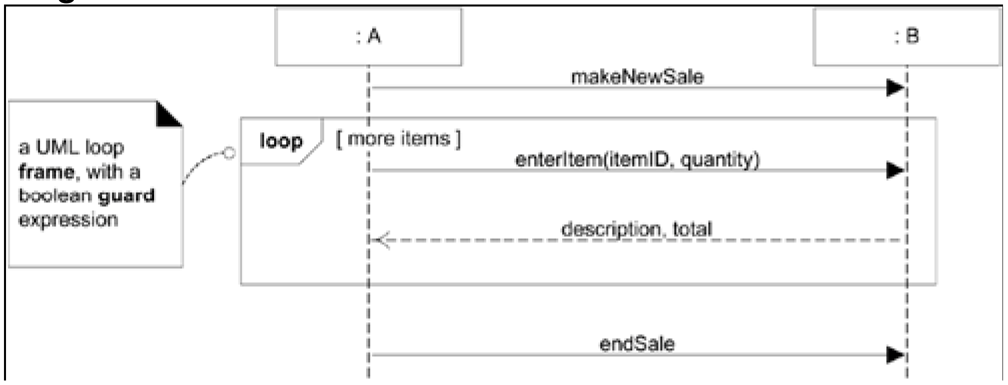

: Sale
create(cashTendered)
: Payment
...
«destroy»
the «destroy» stereotyped message, with the large X and short lifeline indicates explicit object destruction

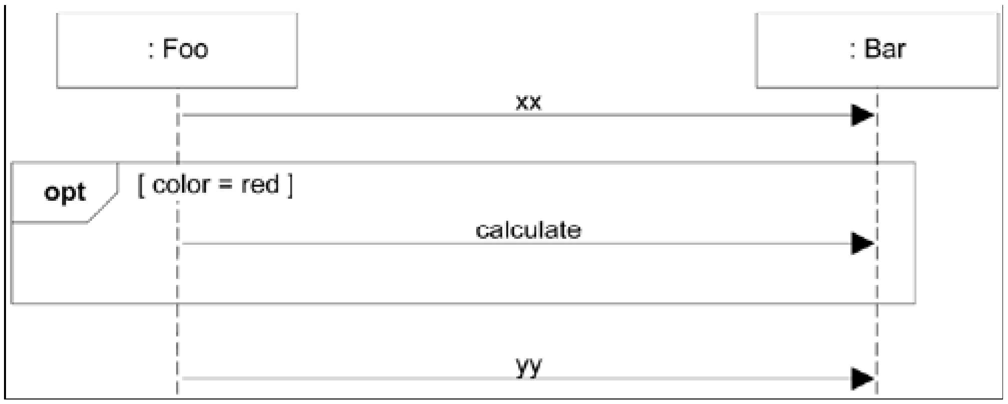
Diagram Frames


: A
: B
makeNewSale
a UML loop **frame**, with a boolean **guard** expression
loop
[more items]
enterItem(itemID, quantity)
description, total
endSale

The following table summarizes some common frame operators:

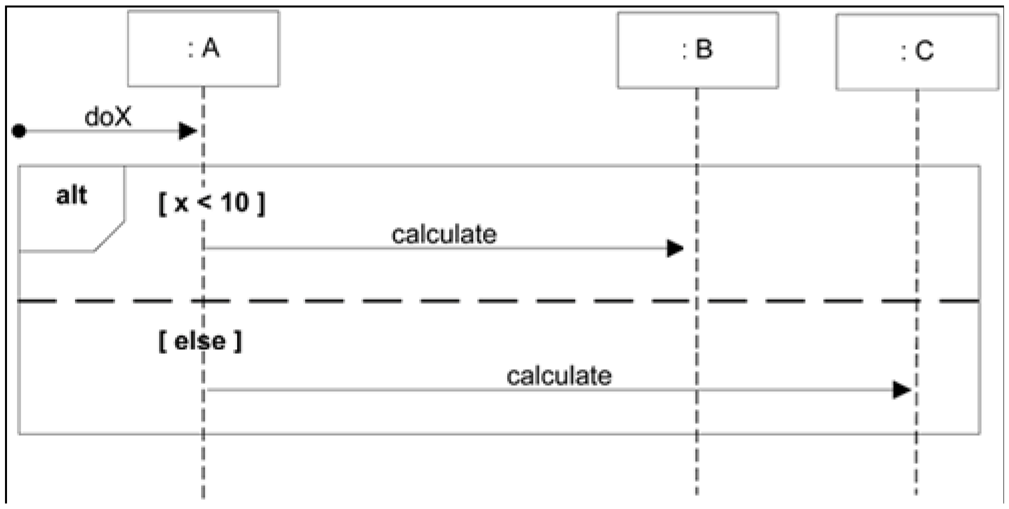
Frame Operator	Meaning
alt	Alternative fragment for mutual exclusion conditional logic expressed in the guards.
loop	Loop fragment while guard is true. Can also write *loop(n)* to indicate looping n times. There is discussion that the specification will be enhanced to define a *FOR* loop, such as *loop(i,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

Conditional Messages

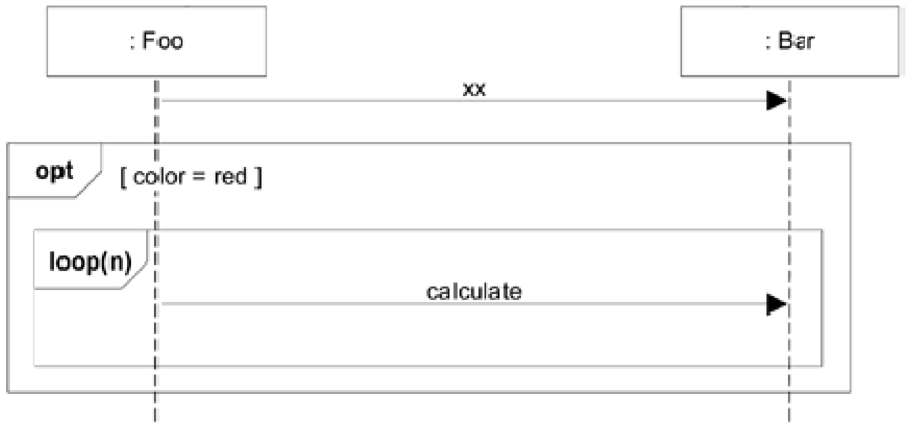


Mutually Exclusive Conditional Messages

An ALT frame is placed around the mutually exclusive alternatives.



Nesting of frames.



Example

The sequence diagram for the book renewal use case for the Library Automation Software is shown in fig. . The development of the sequence diagram in the development methodology would help us in determining the responsibilities of the different classes; i.e. what methods should be supported by each class.

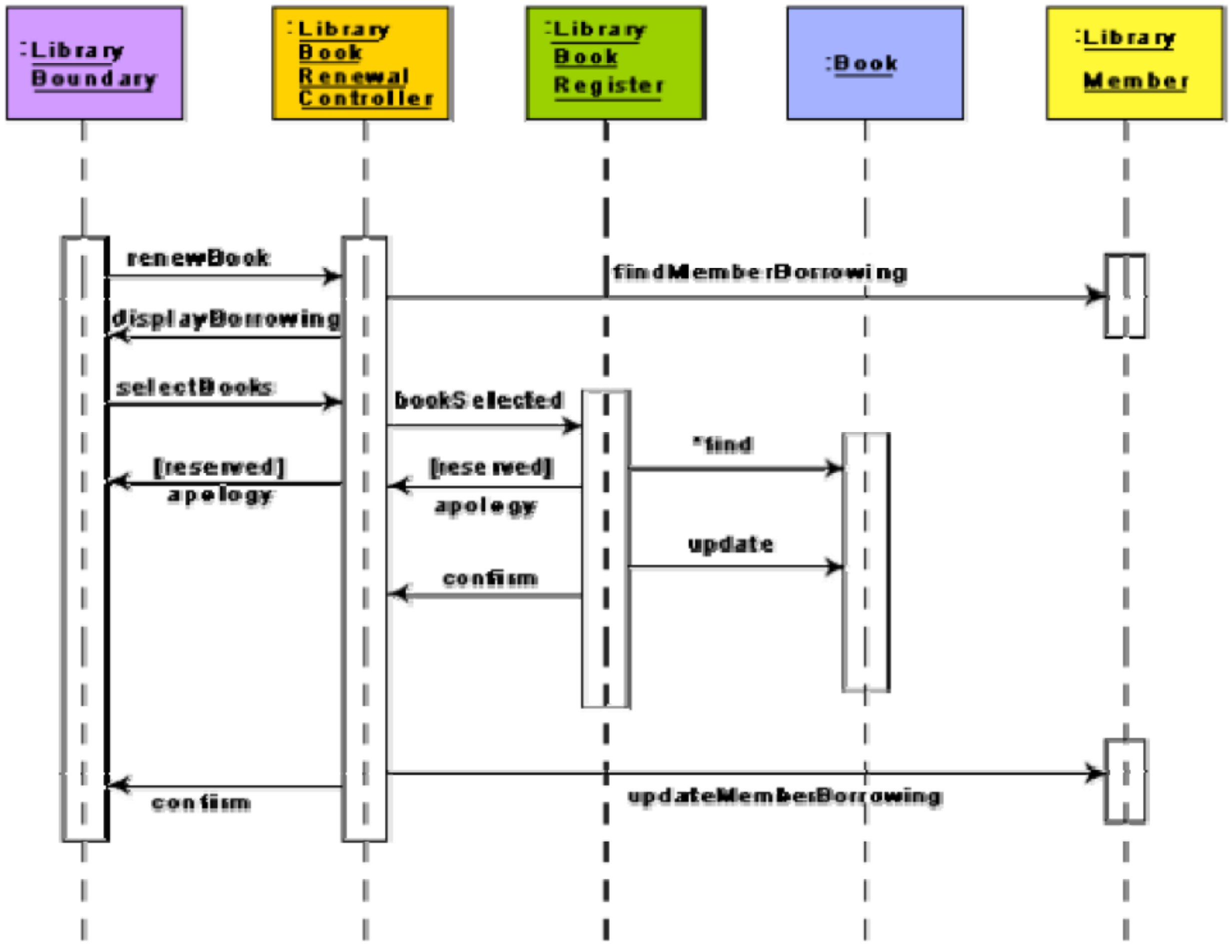


Fig: Sequence diagram for book renew use case.

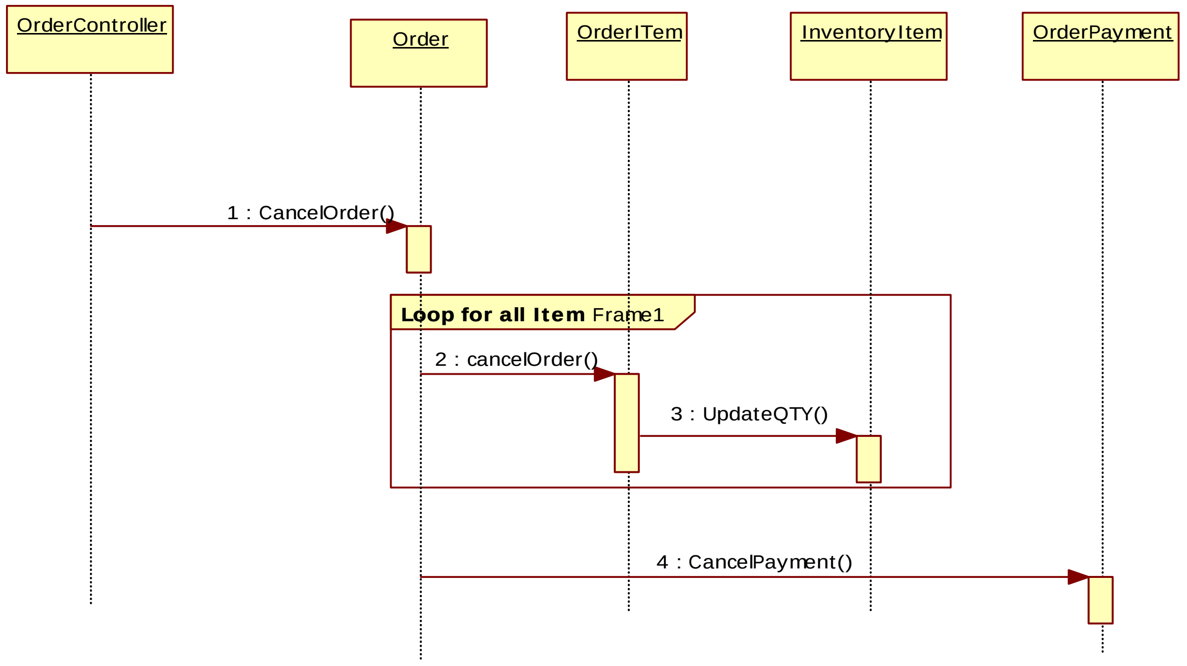


Fig: Sequence Diagram for Cancel Order

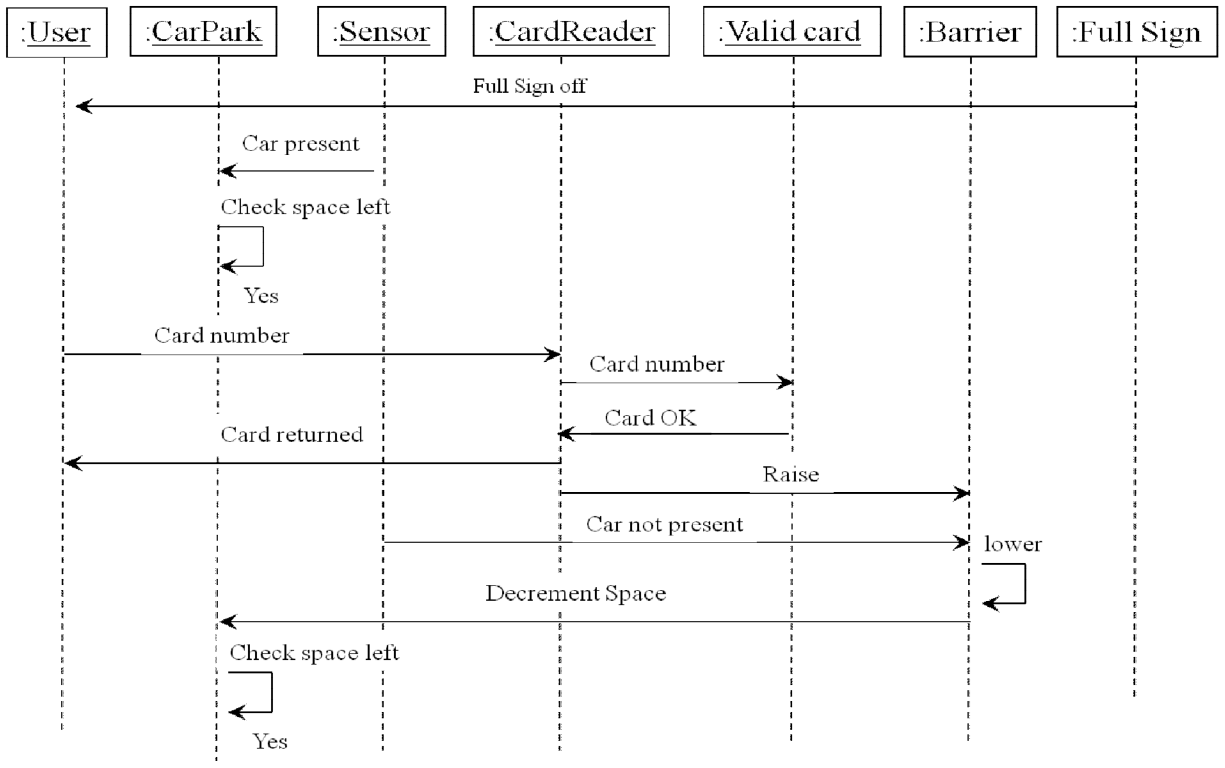


Fig: Sequence diagram for car parking

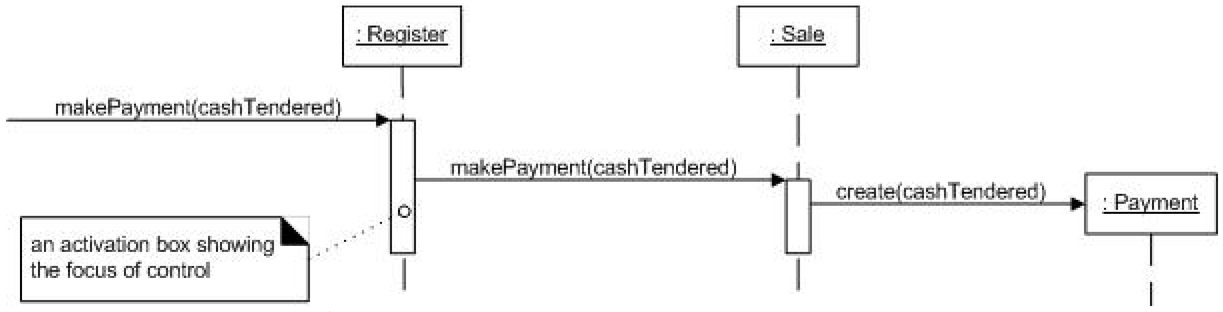

: Register
: Sale
: Payment
makePayment(cashTendered)
makePayment(cashTendered)
create(cashTendered)
an activation box showing the focus of control

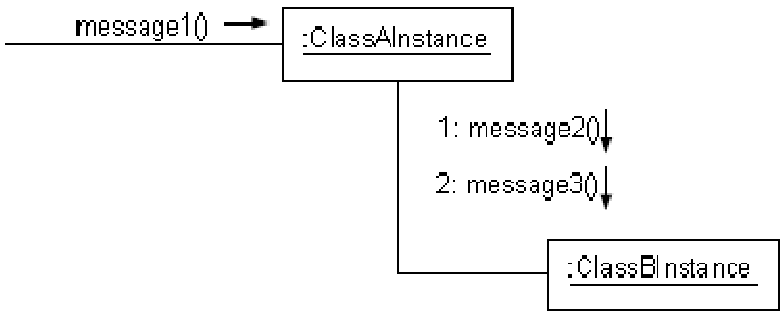
Fig: Sequence diagram of makePayment use case

The sequence diagram shown in Figure *makePayment* is read as follows:
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*.

Communication/Collaboration diagrams

A collaboration diagram shows both structural and behavioral aspects explicitly. This is unlike a sequence diagram which shows only the behavioral aspects. The structural aspect of a collaboration diagram consists of objects and the links existing between them. In this diagram, an object is also called a collaborator. The behavioral aspect is described by the set of messages exchanged among the different collaborators. The link between objects is shown as a solid line and can be used to send messages between two objects. The message is shown as a labeled arrow placed near the link. Messages are prefixed with sequence numbers because they are only way to describe the relative sequencing of the messages in this diagram. The use of the collaboration diagrams in our development process would be to help us to determine which classes are associated with which other classes.

Illustrate object interactions in a graph or network format, in which objects can be placed anywhere on the diagram (the essence of their wall sketching advantage), as shown in fig



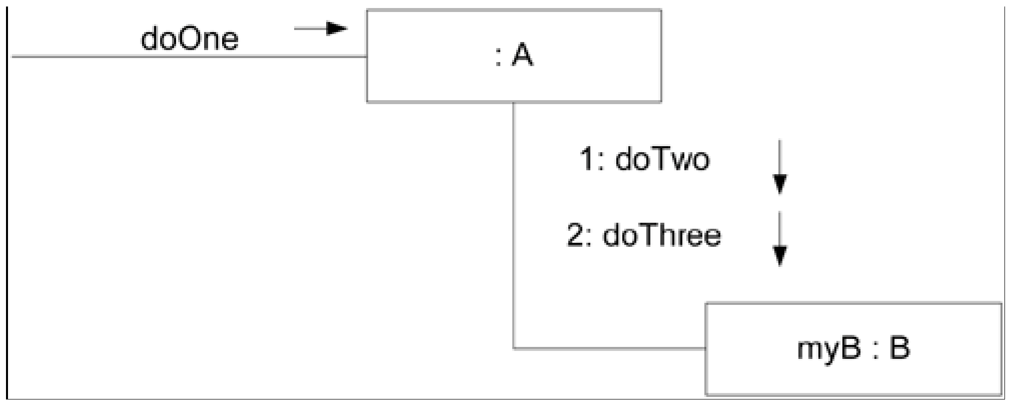


Fig: Communication Diagram (Collaboration)

The collaboration diagram for the example of fig in sequence diagram is shown in fig..

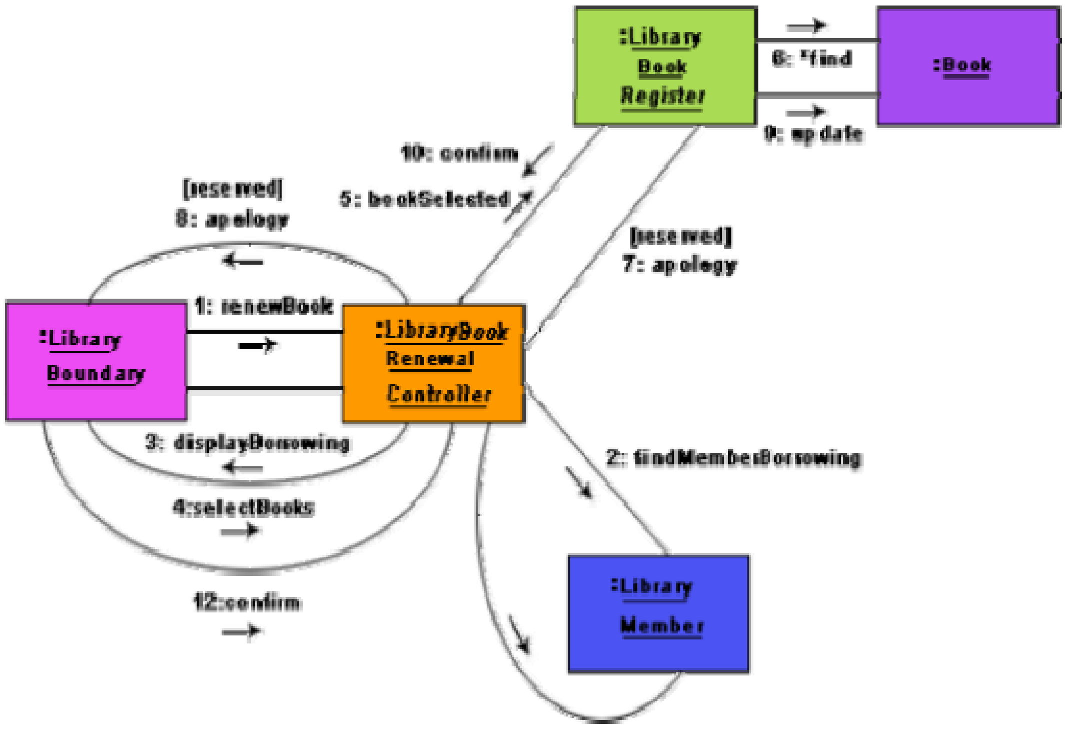


Fig: Collaboration Diagram for book renew use case

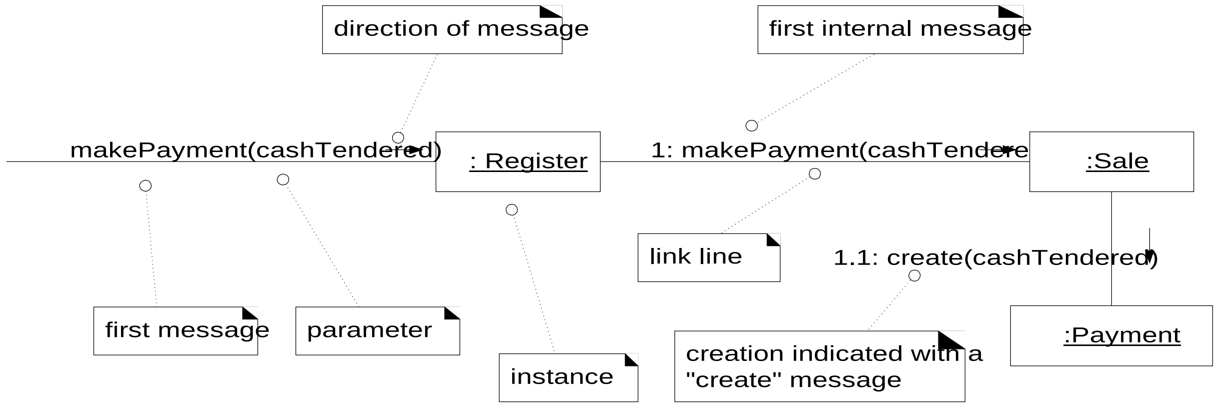

direction of message
first internal message
makePayment(cashTendered)
: Register
1: makePayment(cashTendered)
:Sale
link line
1.1: create(cashTendered)
first message
parameter
instance
creation indicated with a "create" message
:Payment

Fig: Example of collaboration diagram for makePayment

The collaboration diagram shown in Figure 15.3 is read as follows:

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.*

Comparison of Sequence and Collaboration Diagrams

Type	Strengths	Weaknesses
sequence	clearly shows sequence or time ordering of messages large set of detailed notation options	forced to extend to the right when adding new objects; consumes horizontal space
communication	space economicalflexibility to add new objects in two dimensions	more difficult to see sequence of messages fewer notation options

Common Interaction Diagram Notation

Classes and Instances

The UML has adopted a simple and consistent approach to illustrate instances vs. classifiers): *For* any kind of UML element (class, actor, ...), an instance uses the same graphic symbol as the type, but the designator string is underlined.

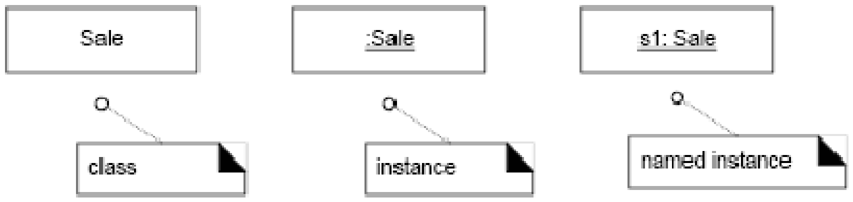


Fig: Class and corresponding object(instance) notation

Therefore, to show an instance of a class in an interaction diagram, the regular class box graphic symbol is used, but the name is underlined. A name can be used to uniquely identify the instance. If none is used, note that a ":" precedes the class name.

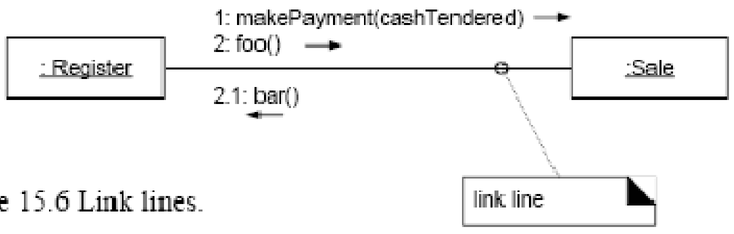
Basic Message Expression Syntax

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:
spec := getProductSpect(id)
spec := getProductSpect(id:ItemID)
spec := getProductSpect(id:ItemID) ProductSpecification

Basic Collaboration Diagram Notation

Links

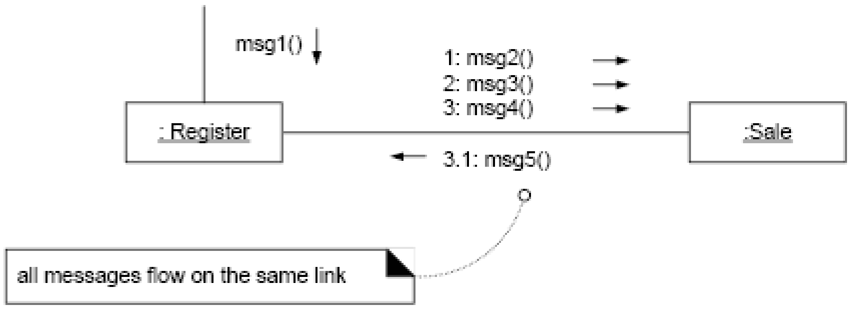
A link is a connection path between two objects; it indicates some form of navigation and visibility between the objects is possible . More formally, a link is an instance of an association. For example, there is a link.or path of navigation.from a *Register* to a *Sale,* along which messages may flow, such as the *makePayment* message.



gure 15.6 Link lines.

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.



Messages to "self" or "this"

A message can be sent from an object to itself .This is illustrated by a link to itself, with messages flowing along the link.

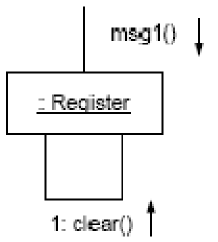
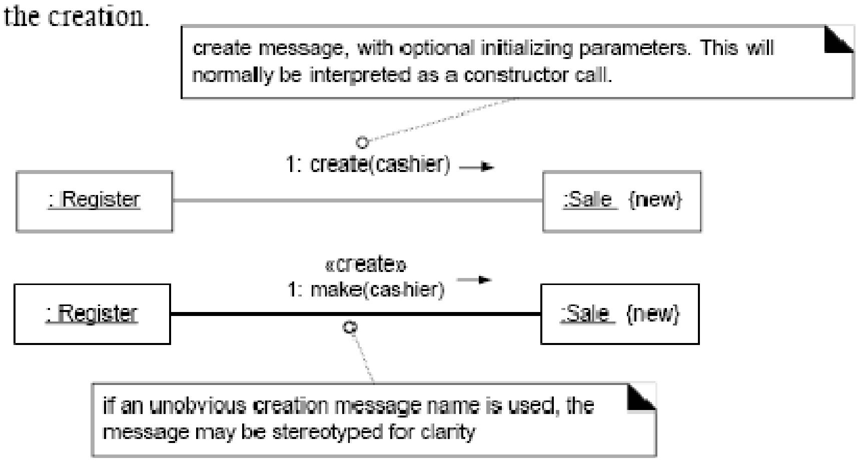


Fig: Notation for self message

Creation of Instances

Any message can be used to create an instance, but there is 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. Furthermore, the UML property *{new}* may optionally be added to the instance box to highlight the creation.

highlight the creation.

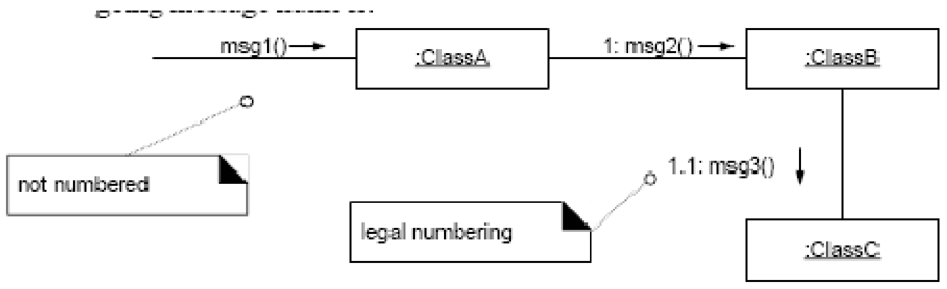


create message, with optional initializing parameters. This will normally be interpreted as a constructor call.
1: create(cashier)
: Register
:Sale {new}
«create»
1: make(cashier)
: Register
:Sale {new}
if an unobvious creation message name is used, the message may be stereotyped for clarity

Message Number Sequencing

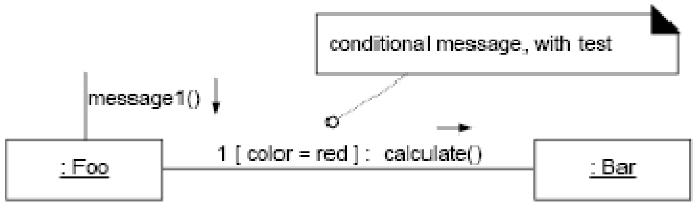
The order of messages is illustrated with **sequence numbers**
1. The first message is not numbered. Thus,*msg1()* is unnumbered.
2. The order and nesting of subsequent messages is shown with a legal num bering scheme in which nested messages have a number appended to them.

Nesting is denoted by prepending the incoming message number to the out going message number.



Conditional Messages:

A conditional message is shown by following 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.*


conditional message, with test
message1()
1 [color = red] : calculate()
: Foo
: Bar

Mutually Exclusive Conditional Paths

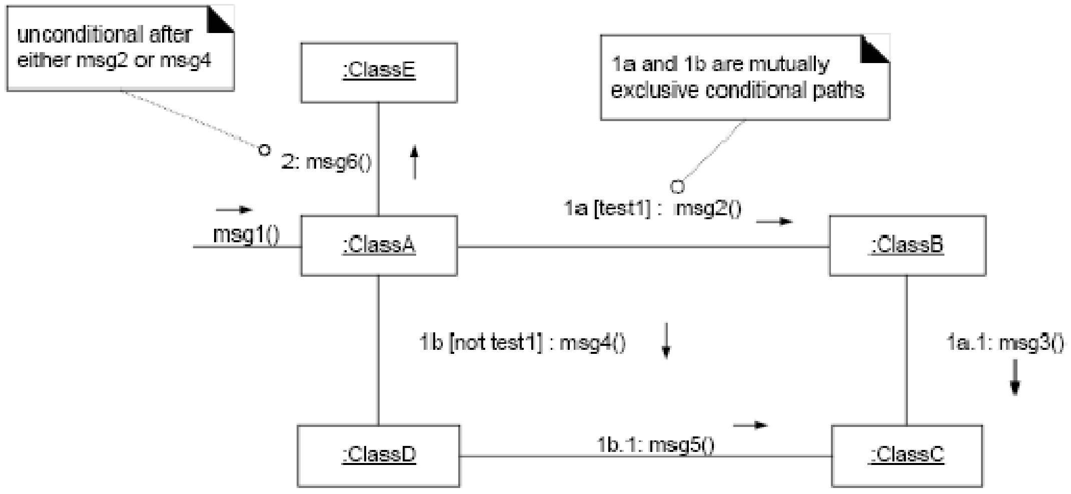


Figure 15.13 Mutually exclusive messages.

In this case it is necessary to modify the sequence expressions with a conditional path letter. The first letter used is a by convention. Figure states that either *1a* or *1b* could execute after *msg1*. Both are sequence number 1 since either could be the first internal message.
Note that subsequent nested messages are still consistently prepended with their outer message sequence. Thus *Ib. 1* is nested message within *Ib.*

Iteration or Looping

Iteration notation is shown in Figure below. If the details of the iteration clause are not important to the modeler, a simple '*' can be used.

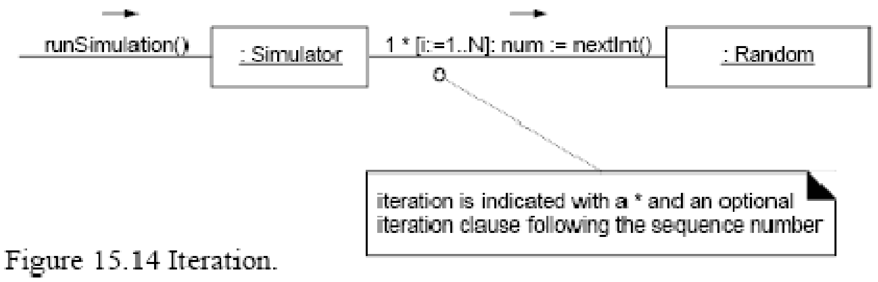
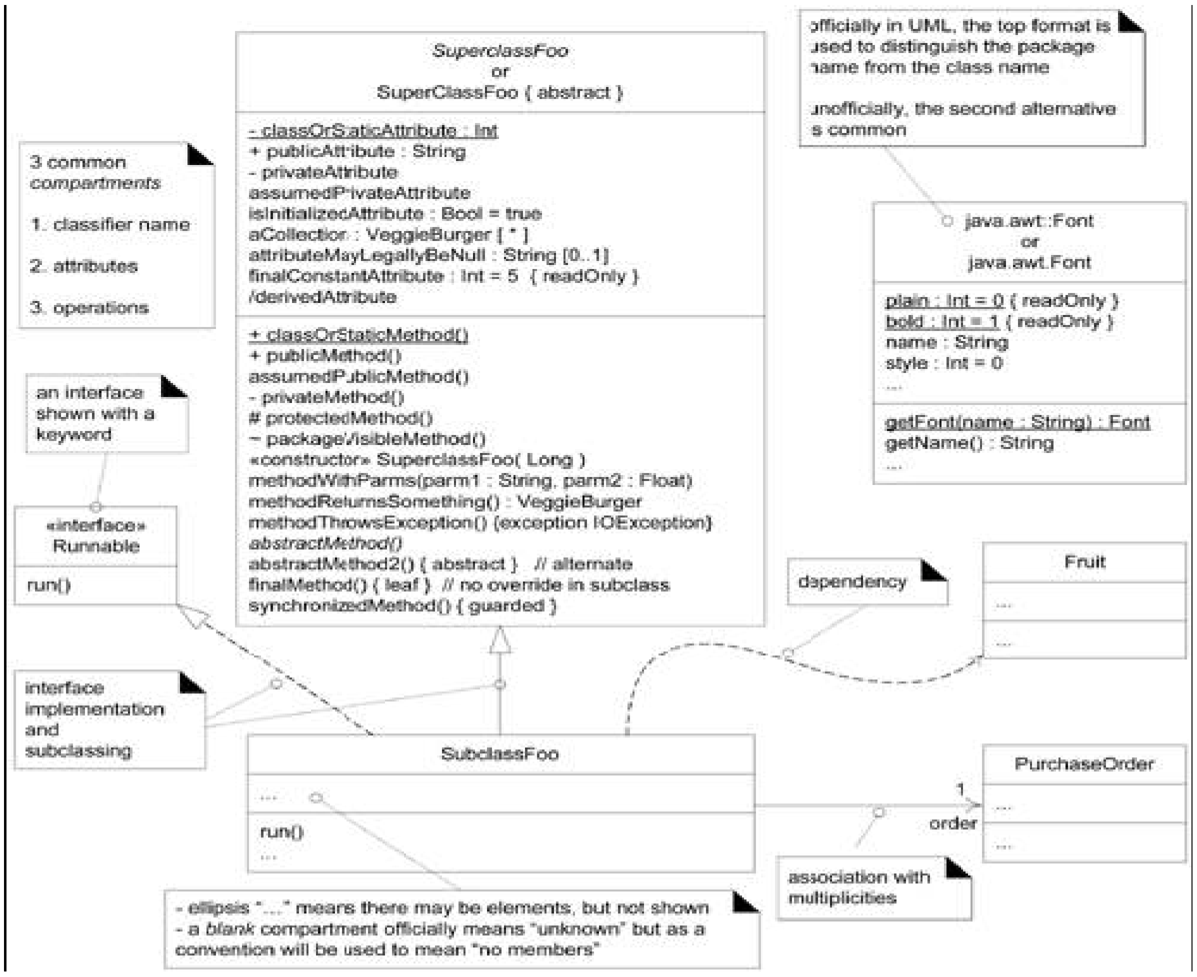

runSimulation()
: Simulator
1 * [i:=1..N]: num := nextInt()
: Random
iteration is indicated with a * and an optional iteration clause following the sequence number

Figure 15.14 Iteration.

Class Diagrams

The UML includes **class diagrams** to illustrate classes, interfaces, and their associations. They are used for **static object modeling**. A class diagram describes the static structure of a system. It shows how a system is structured rather than how it behaves. The static structure of a system comprises of a number of class diagrams and their dependencies. The main constituents of a class diagram are classes and their relationships: generalization, aggregation, association, and various kinds of dependencies.

Common Class Diagram Notation



SuperclassFoo
or
SuperClassFoo { abstract }
- classOrStaticAttribute : Int
+ publicAttribute : String
- privateAttribute
assumedPrivateAttribute
isInitializedAttribute : Bool = true
aCollection : VeggieBurger [*]
attributeMayLegallyBeNull : String [0..1]
finalConstantAttribute : Int = 5 { readOnly }
/derivedAttribute
+ classOrStaticMethod()
+ publicMethod()
assumedPublicMethod()
- privateMethod()
# protectedMethod()
~ packageVisibleMethod()
«constructor» SuperclassFoo(Long)
methodWithParms(parm1 : String, parm2 : Float)
methodReturnsSomething() : VeggieBurger
methodThrowsException() {exception IOException}
abstractMethod()
abstractMethod2() { abstract } // alternate
finalMethod() { leaf } // no override in subclass
synchronizedMethod() { guarded }
3 common compartments
1. classifier name
2. attributes
3. operations
officially in UML, the top format is used to distinguish the package name from the class name
unofficially, the second alternative is common
java.awt::Font
or
java.awt.Font
plain : Int = 0 { readOnly }
bold : Int = 1 { readOnly }
name : String
style : Int = 0
...
getFont(name : String) : Font
getName() : String
...
an interface shown with a keyword
«interface»
Runnable
run()
interface implementation and subclassing
dependency
Fruit
...
...
SubclassFoo
...
run()
...
PurchaseOrder
...
...
1
order
association with multiplicities
- ellipsis "..." means there may be elements, but not shown
- a blank compartment officially means "unknown" but as a convention will be used to mean "no members"

Design Class Diagram(DCD)

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

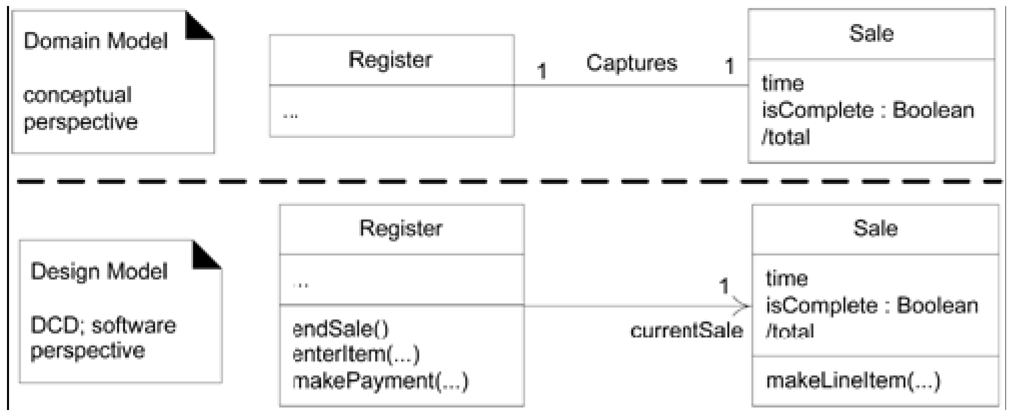


Fig: Different Perspectives in Class Diagram

Classifier

A UML **classifier** is "a model element that describes behavioral and structure features" . Classifiers can also be specialized. They are a generalization of many of the elements of the UML, including classes, interfaces, use cases, and actors. In class diagrams, the two most common classifiers are regular classes and interfaces.

Types of Relationship In Class Diagrams:

1. Associatation
2. Aggregation
3. Composition
4. Inheritance(Generalization/Specialization)
5. Dependency

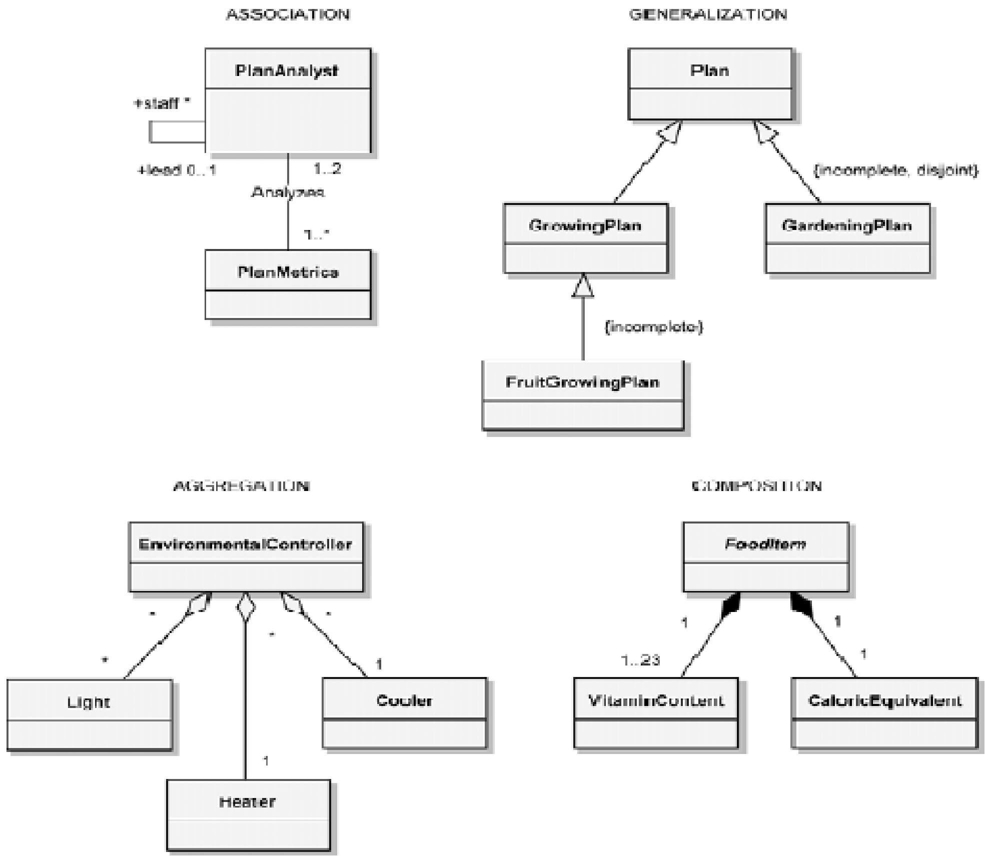
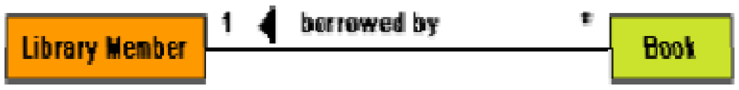


Figure 5–35 Class Relationship Icons

Associations

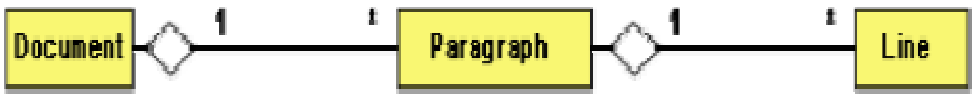
Associations are needed to enable objects to communicate with each other. An association describes a connection between classes. The association relation between two objects is called object connection or link. Links are instances of associations. A link is a physical or conceptual connection between object instances. For example, suppose Amit has borrowed the book Graph Theory.
Here, borrowed is the connection between the objects Amit and Graph Theory book. Mathematically, a link can be considered to be a tuple, i.e. an ordered list of object instances. An association describes a group of links with a common structure and common semantics. For example, consider the statement that
Library Member borrows Books. Here, borrows is the association between the class LibraryMember and the class Book. Usually, an association is a binary relation (between two classes). However, three or more different classes can be involved in an association. A class can have an association relationship with itself (called recursive association). In this case, it is usually assumed that two different objects of the class are linked by the association relationship.
Association between two classes is represented by drawing a straight line between the concerned classes. Fig. 7.9 illustrates the graphical representation of the association relation. The name of the association is written along side the association line. An arrowhead may be placed on the association line to indicate the reading direction of the association. The arrowhead should not be misunderstood to

be indicating the direction of a pointer implementing an association. On each side of the association relation, the multiplicity is noted as an individual number or as a value range. The multiplicity indicates how many instances of one class are associated with each other. Value ranges of multiplicity are noted by specifying the minimum and maximum value, separated by two dots, e.g. 1.5. An asterisk is a wild card and means many (zero or more).

	The association of fig should be read as "Many books may be borrowed by a Library Member". Observe that associations (and links) appear as verbs in the problem statement.

Aggregation

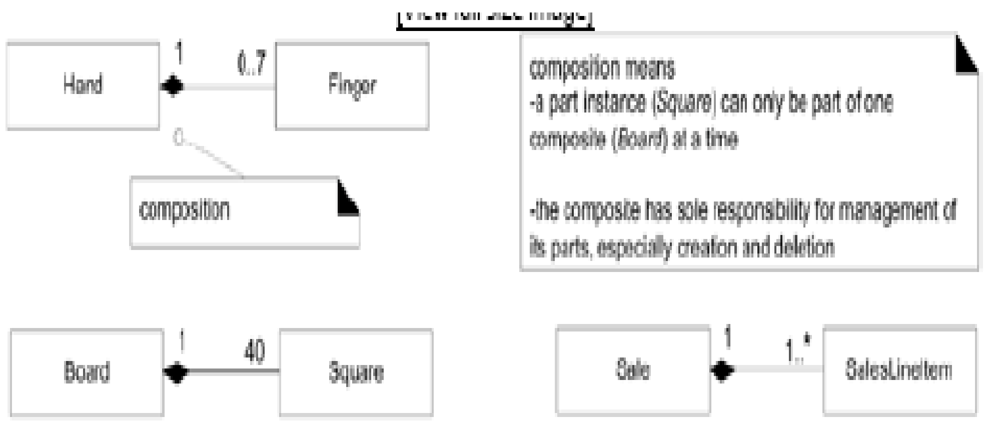
Aggregation is a special type of association where the involved classes represent a whole-part relationship. The aggregate takes the responsibility of forwarding messages to the appropriate parts. Thus, the aggregate takes the responsibility of delegation and leadership. When an instance of one object contains instances of some other objects, then aggregation (or composition) relationship exists between the composite object and the component object. Aggregation is represented by the diamond symbol at the composite end of a relationship. The number of instances of the component class aggregated can also be shown as in fig.



Aggregation relationship cannot be reflexive (i.e. recursive). That is, an object cannot contain objects of the same class as itself. Also, the aggregation relation is not symmetric. That is, two classes A and B cannot contain instances of each other. However, the aggregation relationship can be transitive. In this case, aggregation may consist of an arbitrary number of levels.

Composition

Composition is a stricter form of aggregation, in which the parts are existence-dependent on the whole. This means that the life of the parts closely ties to the life of the whole. When the whole is created, the parts are created and when the whole is destroyed, the parts are destroyed. A typical example of composition is an invoice object with invoice items. As soon as the invoice object is created, all the invoice items in it are created and as soon as the invoice object is destroyed, all invoice items in it are also destroyed. The composition relationship is represented as a filled diamond drawn at the composite-end. An example of the composition relationship is shown in fig..


Hand
1
0..7
Finger
composition
composition means
-a part instance (Square) can only be part of one composite (Board) at a time
-the composite has sole responsibility for management of its parts, especially creation and deletion
Board
1
40
Square
Sale
1
1..*
SalesLineItem

Association vs. Aggregation vs. Composition

- Association is the most general (m:n) relationship. Aggregation is a stronger relationship where one is a part of the other. Composition is even stronger than aggregation, ties the lifecycle of the part and the whole together.
- Association relationship can be reflexive (objects can have relation to itself), but aggregation cannot be reflexive. Moreover, aggregation is anti-symmetric (If B is a part of A, A can't be a part of B).
- Composition has the property of exclusive aggregation i.e. an object can be a part of only one composite at a time. For example, a **Frame** belongs to exactly one **Window** whereas in simple aggregation, a part may be shared by several objects. For example, a **Wall** may be a part of one or more **Room** objects.
- In addition, in composition, the whole has the responsibility for the disposition of all its parts, i.e. for their creation and destruction.
    - in general, the lifetime of parts and composite coincides o parts with non-fixed multiplicity may be created after composite itself
    - parts might be explicitly removed before the death of the composite

- For example, when a **Frame** is created, it has to be attached to an enclosing **Window**. Similarly, when the **Window** is destroyed, it must in turn destroy its **Frame** parts.

Inheritance vs. Aggregation/Composition

- Inheritance describes *'is a'* / *'is a kind of'* relationship between classes (base class - derived class) whereas aggregation describes *'has a'* relationship between classes. Inheritance means that the object of the derived class inherits the properties of the base class; aggregation means that the object of the whole has objects of the part. For example, the relation "cash payment *is a kind of* payment" is modeled using inheritance; "purchase order has a few items" is modeled using aggregation.
- Inheritance is used to model a "generic-specific" relationship between classes whereas aggregation/composition is used to model a "whole-part" relationship between classes.
- Inheritance means that the objects of the subclass can be used anywhere the super class may appear, but not the reverse; i.e. wherever we could use instances of 'payment' in the system, we could substitute it with instances of 'cash payment', but the reverse can not be done.

- Inheritance is defined statically. It can not be changed at run-time. Aggregation is defined dynamically and can be changed at run-time. Aggregation is used when the type of the object can change over time.

For example, consider this situation in a business system. A **BusinessPartner** might be a **Customer** or a **Supplier** or both. Initially we might be tempted to model it as in Fig 7.12(a). But in fact, during its lifetime, a business partner might become a customer as well as a supplier, or it might change from one to the other. In such cases, we prefer aggregation instead (see Fig 7.12(b). Here, a business partner is a **Customer** if it has an aggregated **Customer** object, a **Supplier** if it has an aggregated **Supplier** object and a "C**ustomer_Supplier**" if it has both.
Here, we have only two types. Hence, we are able to model it as inheritance. But what if there were several different types and combinations there of? The inheritance tree would be absolutely incomprehensible.
Also, the aggregation model allows the possibility for a business partner to be neither - i.e. has neither a customer nor a supplier object aggregated with it.
• The advantage of aggregation is the integrity of encapsulation. The operations of an object are the interfaces of other objects which imply low implementation dependencies. The significant disadvantage of aggregation is the increase in the number of objects and their relationships. On the other hand, inheritance allows for an easy way to modify implementation for reusability. But the significant disadvantage is that it breaks encapsulation, which implies implementation dependence.

Dependency/Using relationships

Dependency relationship indicates that one element (of any kind, including classes, use cases, and so on) has knowledge of another element. A dependency is a using relationship that states a change in specification of one thing may affect another thing that uses it, but not necessarily the reverse. The dependency relationship is useful to depict non-attribute visibility between classes, For Parameters and Global or local visibility

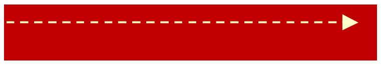


Fig: Notation of Dependency

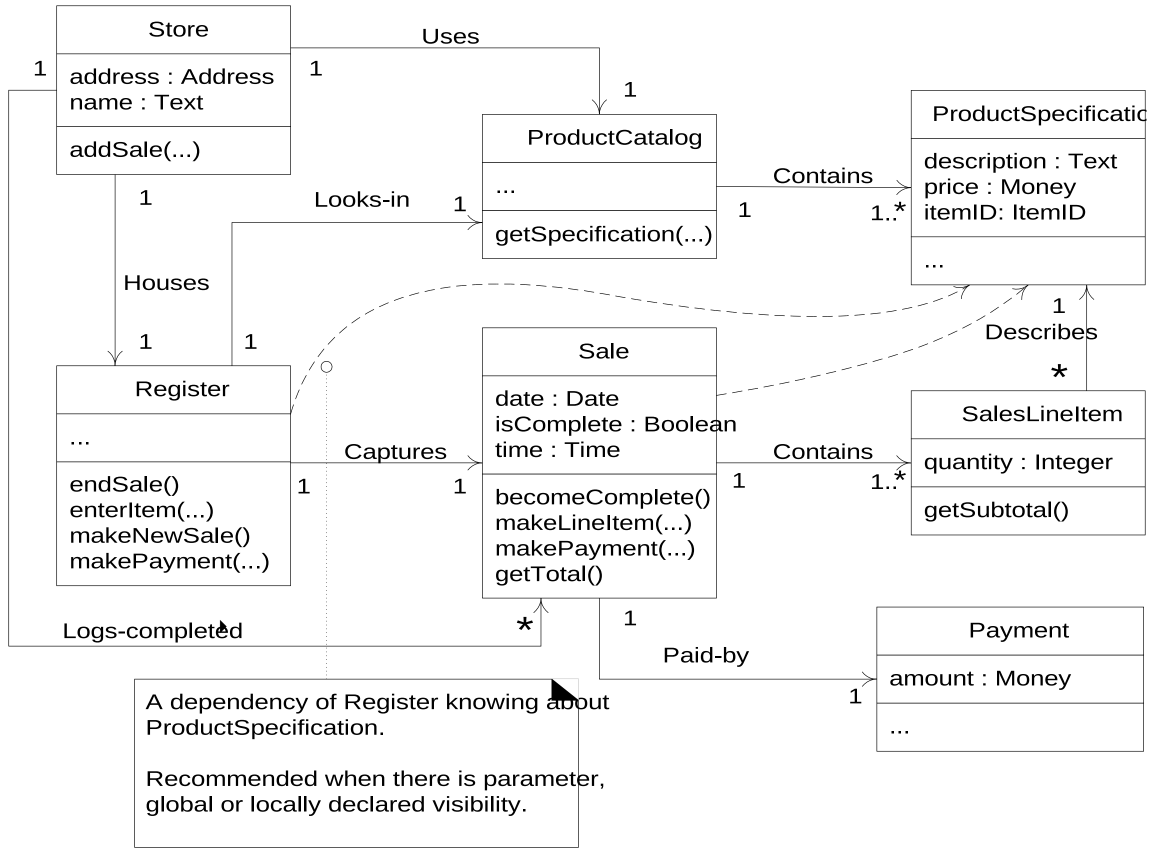


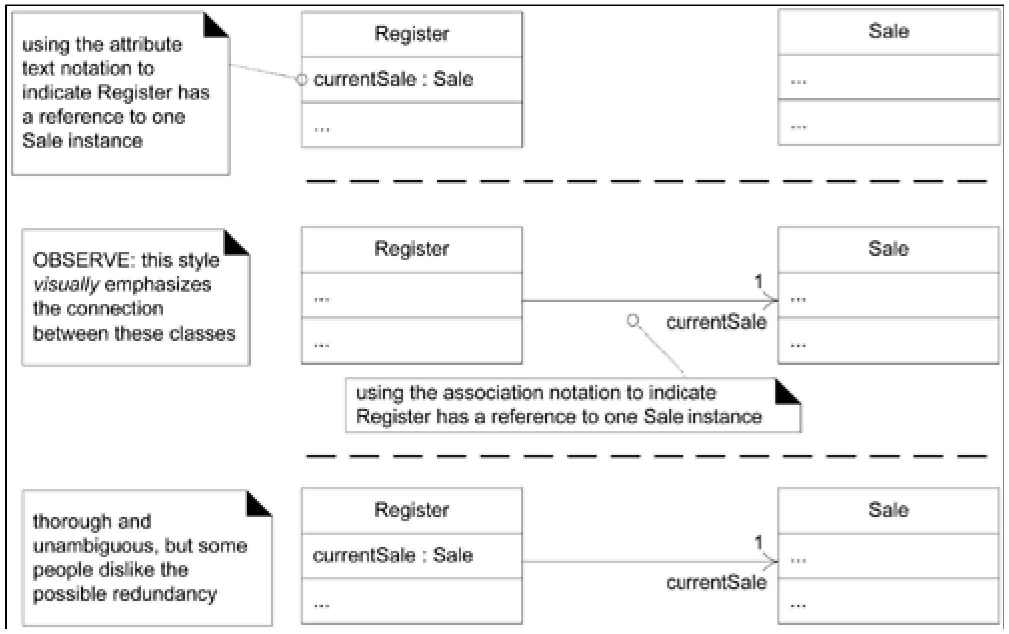
Fig: Dependency relationships non-attribute visibility

Attribute Text and Association Lines

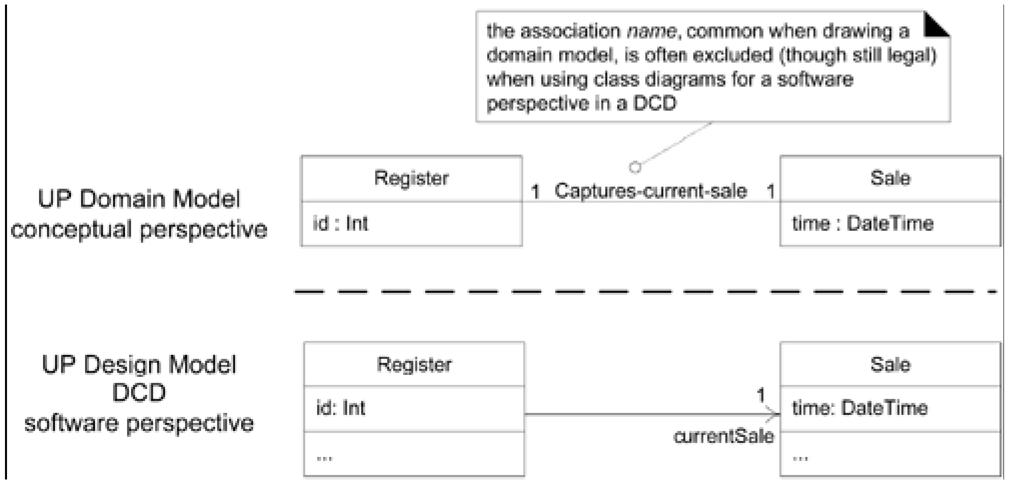
Attributes of a classifier (also called structural properties in the UML) are shown several ways:

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

Figure shows these notations being used to indicate that a *Register* object has an attribute (a reference to) one *Sale* object.


using the attribute text notation to indicate Register has a reference to one Sale instance
Register
currentSale : Sale
...
Sale
...
...
OBSERVE: this style *visually* emphasizes the connection between these classes
Register
...
...
1
currentSale
Sale
...
...
using the association notation to indicate Register has a reference to one Sale instance
thorough and unambiguous, but some people dislike the possible redundancy
Register
currentSale : Sale
...
1
currentSale
Sale
...
...
Fig: Attribute text versus association line notation for a UML attribute.

The full format of the attribute text notation is:
visibility name : type multiplicity = default {property-string}
Also, the UML allows any other programming language syntax to be used for the attribute declaration, as long as the reader or tool are notified.
visibility marks include + (public), - (private), and so forth.
Guideline: Attributes are usually assumed private if no visibility is given.


the association *name*, common when drawing a domain model, is often excluded (though still legal) when using class diagrams for a software perspective in a DCD
UP Domain Model
conceptual perspective
Register
id : Int
1 Captures-current-sale 1
Sale
time : DateTime
UP Design Model
DCD
software perspective
Register
id: Int
...
1
currentSale
Sale
time: DateTime
...
Fig: association notation usage in different perspectives.

attribute-as-association line has the following style:

- a **navigability arrow** pointing from the source (*Register*) to target (*Sale*) object, indicating a *Register* object has an attribute of one *Sale*
- a multiplicity at the target end, but not the source end use the multiplicity notation

- a **rolename** (*currentSale*) only at the target end to show the attribute name
- no association name

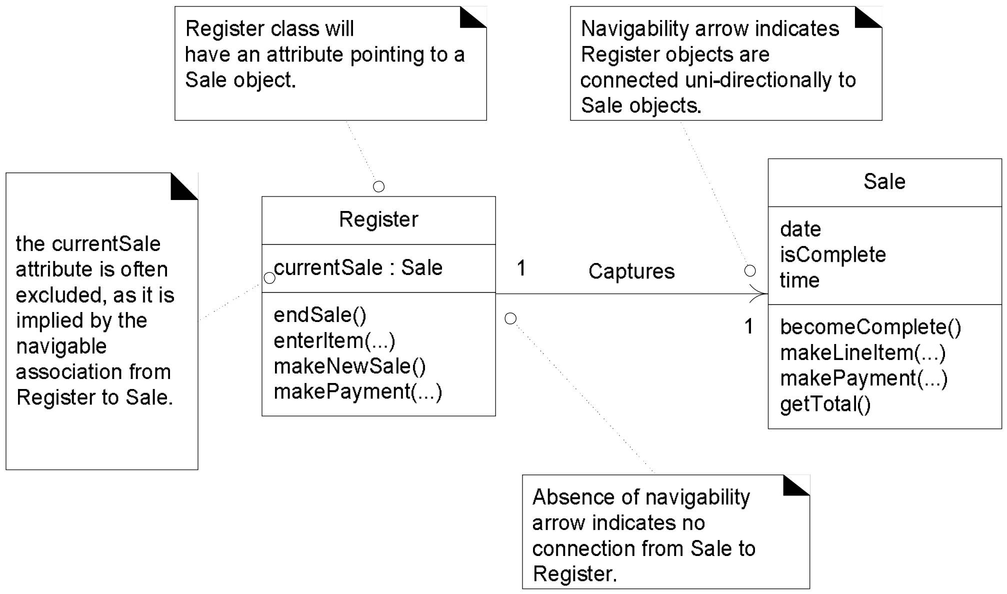

Register class will have an attribute pointing to a Sale object.
Navigability arrow indicates Register objects are connected uni-directionally to Sale objects.
the currentSale attribute is often excluded, as it is implied by the navigable association from Register to Sale.
Register
currentSale : Sale
endSale()
enterItem(...)
makeNewSale()
makePayment(...)
1
Captures
1
Sale
date
isComplete
time
becomeComplete()
makeLineItem(...)
makePayment(...)
getTotal()
Absence of navigability arrow indicates no connection from Sale to Register.

Fig: Associations with navigability adornments

Notes, Comments, Constraints, and Method Bodies

Note symbols can be used on any UML diagram, but are especially common on class diagrams. A UML **note symbol** is displayed as a dog-eared rectangle with a dashed line to the annotated element; A note symbol may represent several things, such as:

- a UML **note** or **comment**, which by definition have no semantic impact
- a UML **constraint**, in which case it must be encased in braces '{…}'
- a **method** body the implementation of a UML operation .

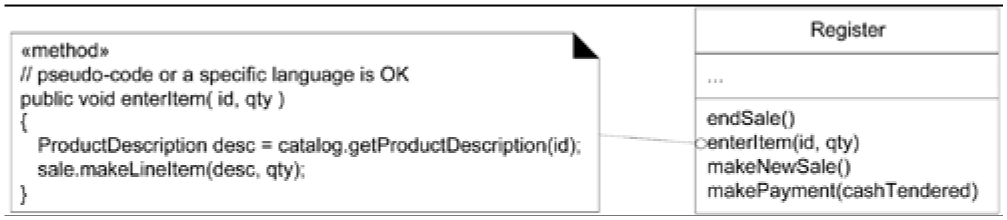

```
«method»
// pseudo-code or a specific language is OK
public void enterItem( id, qty )
{
  ProductDescription desc = catalog.getProductDescription(id);
  sale.makeLineItem(desc, qty);
}
```

Fig: Example of Method body notation in Note Symbol

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*. In the UML, it is illustrated with a dashed line from the association to the association class.

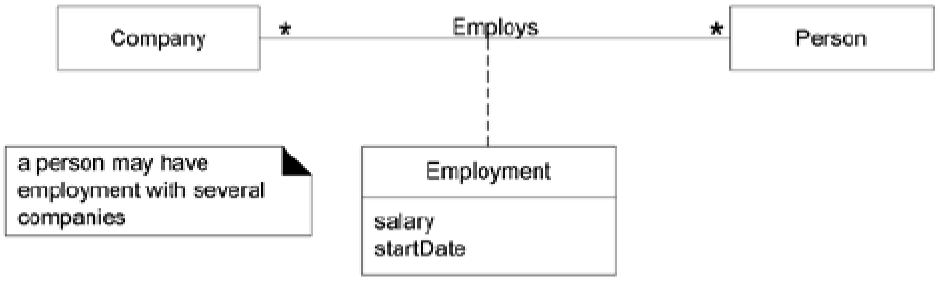
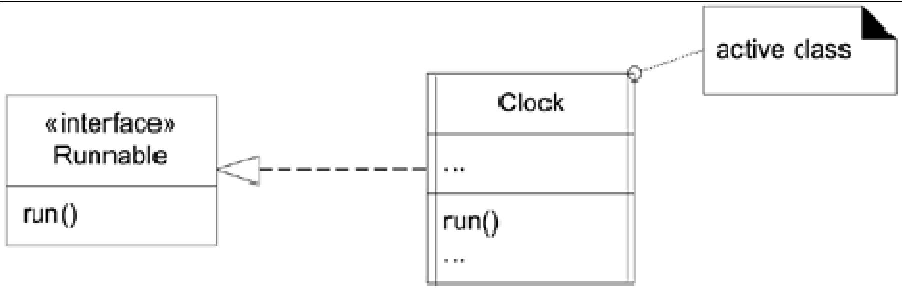


Fig: Association Class

Singleton Classes

Active Class

 active class «interface» Runnable run() Clock ... run() ... Fig: 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.

Determining Visibility:

Visibility is the ability of an object to "see" or have a reference to another object. More generally, it is related to the issue of scope: Is one resource (such as an instance) within the scope of another? 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.

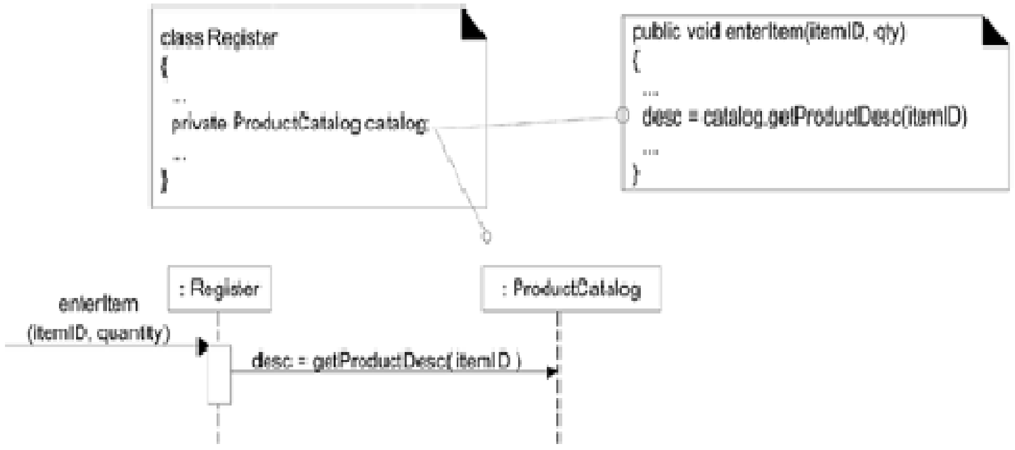
For example, to create an interaction diagram in which a message is sent from a *Register* instance to a *ProductCatalog* instance, the *Register* must have visibility to the *ProductCatalog*. A typical visibility solution is that a reference to the *ProductCatalog* instance is maintained as an attribute of the *Register*.

Attribute Visibility

Attribute visibility from A to B exists when B is an attribute of A. It is a relatively permanent visibility because it persists as long as A and B exist. This is a very common form of visibility in object-oriented systems.

To illustrate, in a Java class definition for *Register*, a *Register* instance may have attribute visibility to a *ProductCatalog*, since it is an attribute (Java instance variable) of the *Register*.

```
public class Register
{
...
private ProductCatalog catalog;
...
}
```



class Register
{
...
private ProductCatalog catalog;
...
}
public void enterItem(itemID, qty)
{
...
desc = catalog.getProductDesc(itemID)
...
}
: Register
: ProductCatalog
enterItem
(itemID, quantity)
desc = getProductDesc(itemID)

Fig: Attribute Visibility

Parameter Visibility

Parameter visibility from A to B exists when B is passed as a parameter to a method of A. It is a relatively temporary visibility because it persists only within the scope of the method. After attribute visibility, it is the second most common form of visibility in object-oriented systems.

To illustrate, when the *makeLineItem* message is sent to a *Sale* instance, a *ProductDescription* instance is passed as a parameter. Within the scope of the *makeLineItem* method, the *Sale* has parameter visibility to a *ProductDescription*

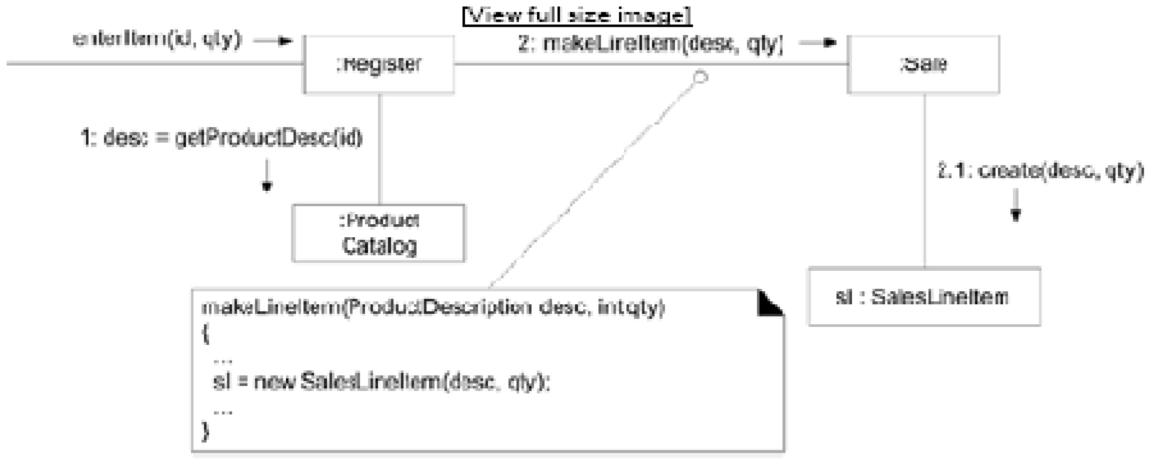

[View full size image]
enterItem(id, qty)
:Register
2: makeLineItem(desc, qty)
:Sale
1: desc = getProductDesc(id)
2.1: create(desc, qty)
:Product Catalog
sl : SalesLineItem
makeLineItem(ProductDescription desc, int qty)
{
...
sl = new SalesLineItem(desc, qty);
...
}

Fig: Parameter Visibility

It is common to transform parameter visibility into attribute visibility. When the *Sale* creates a new *SalesLineItem*, it passes the *ProductDescription* in to its initializing method (in C++ or Java, this would be its **constructor**). Within the initializing method, the parameter is assigned to an attribute, thus establishing attribute visibility

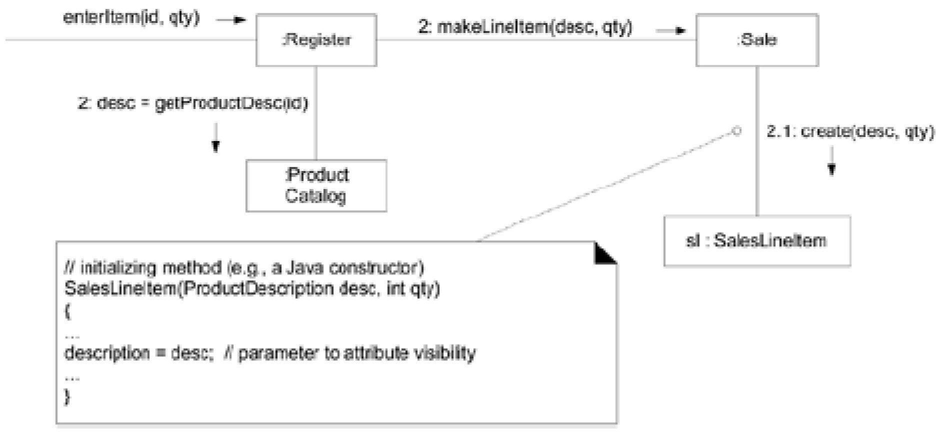

enterItem(id, qty)
:Register
2: makeLineItem(desc, qty)
:Sale
2: desc = getProductDesc(id)
2.1: create(desc, qty)
:Product Catalog
sl : SalesLineItem
// initializing method (e.g., a Java constructor)
SalesLineItem(ProductDescription desc, int qty)
{
...
description = desc; // parameter to attribute visibility
...
}

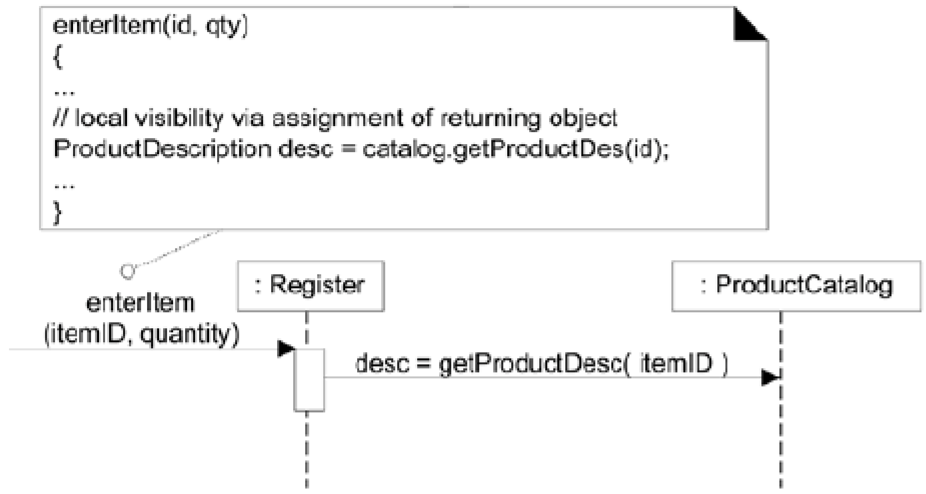
Fig: Parameter to attribute visibility

Local 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. After parameter visibility, it is the third most common form of visibility in object-oriented systems.
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.

As with parameter visibility, it is common to transform locally declared visibility into attribute visibility. An example of the second variation (assigning the returning object to a local variable) can be found in the *enterItem* method of class *Register*.



```
enterItem(id, qty)
{
...
// local visibility via assignment of returning object
ProductDescription desc = catalog.getProductDes(id);
...
}
```

Fig: Local visibility

Global Visibility
Global visibility from A to B exists when B is global to A. It is a relatively permanent visibility because it persists as long as A and B exist. It is the least common form of visibility in object-oriented 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.

Constraints
Constraints may be used on most UML diagrams, but are especially common on class diagrams. 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.

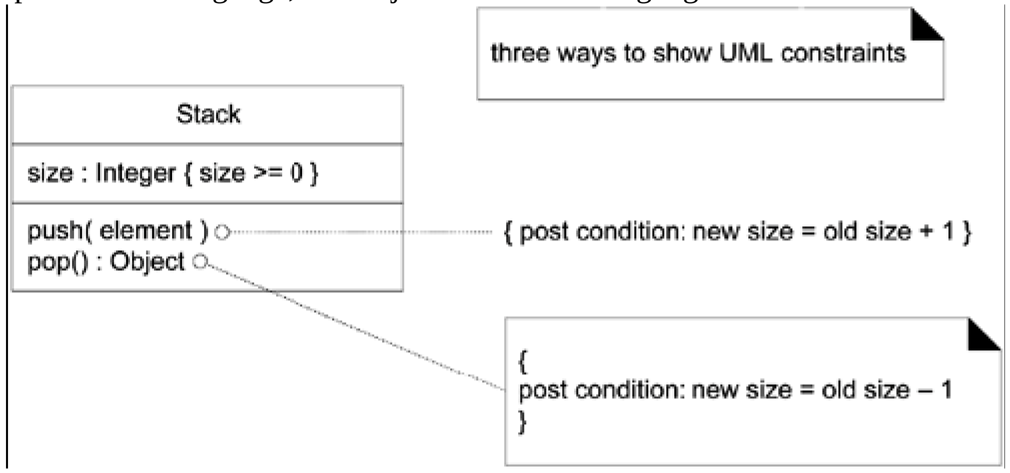

three ways to show UML constraints
Stack
size : Integer { size >= 0 }
push(element)
pop() : Object
{ post condition: new size = old size + 1 }
{
post condition: new size = old size – 1
}

Fig: Constraint Example

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. For example, if a *ProductCatalog* contains many *ProductDescriptions*, and each one can be selected by an *itemID*,

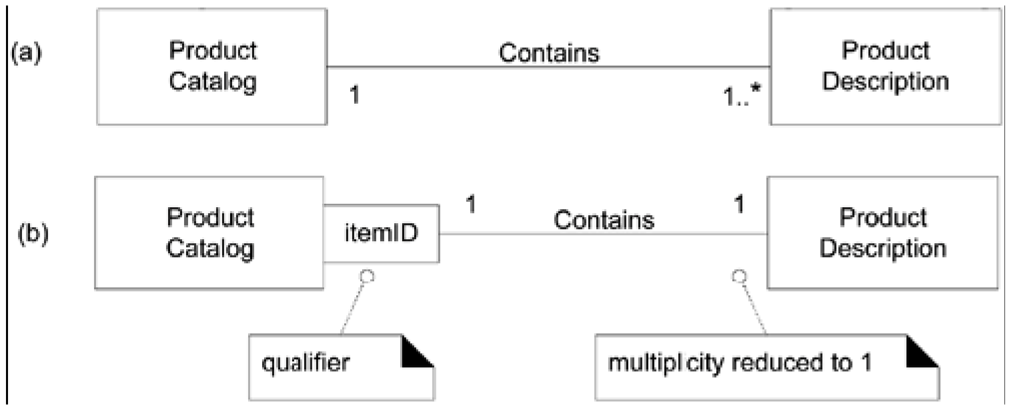
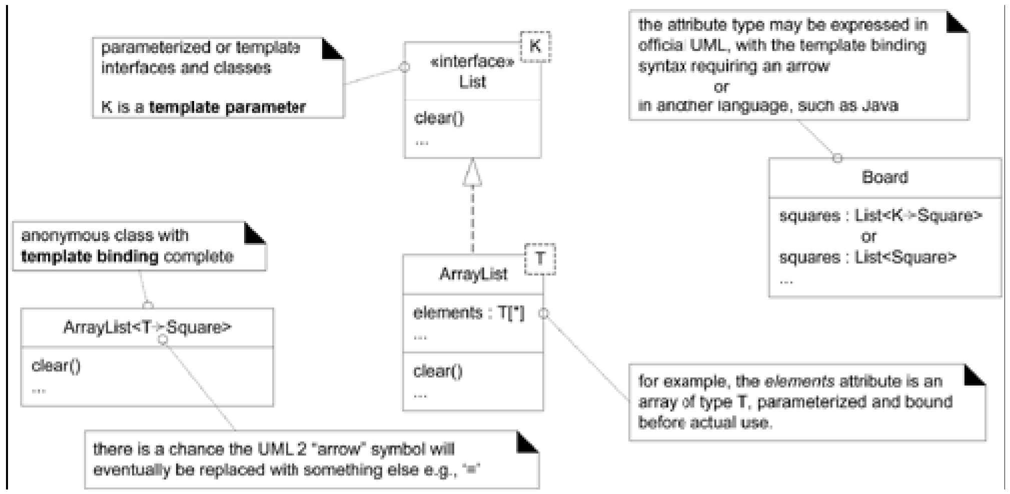


Fig: Qualifier

Template Classes and Interfaces

Many languages (Java, C++, …) support templatized types, also known (with shades of variant meanings) as templates, parameterized types, and generics.[7] 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>();
// …
}
```



parameterized or template interfaces and classes
K is a template parameter
«interface» List
K
clear()
...
the attribute type may be expressed in official UML, with the template binding syntax requiring an arrow
or
in another language, such as Java
Board
squares : List<K->Square>
or
squares : List<Square>
...
anonymous class with template binding complete
ArrayList
T
elements : T[*]
...
clear()
...
ArrayList<T->Square>
clear()
...
for example, the elements attribute is an array of type T, parameterized and bound before actual use.
there is a chance the UML 2 "arrow" symbol will eventually be replaced with something else e.g., '='

Fig: Interface and Template class

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.

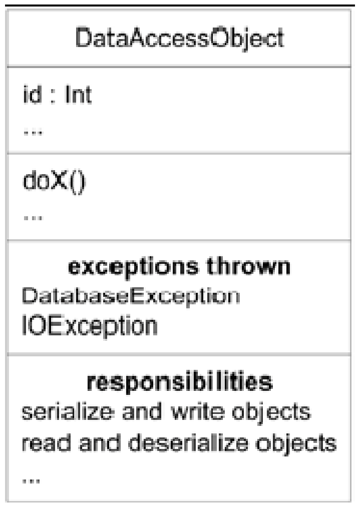
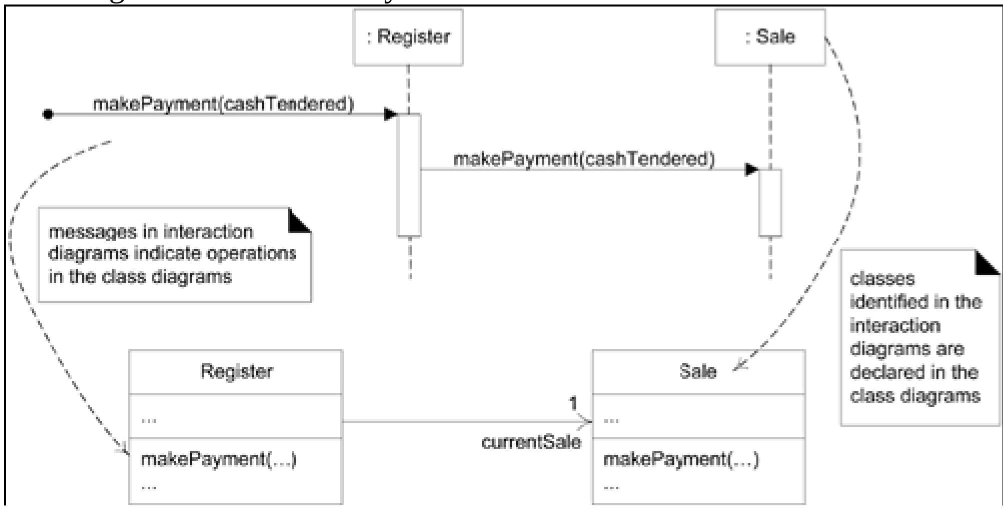


Fig: User defied compartments

Relationship Between Class diagram and interaction diagram

When we draw interaction diagrams, a set of classes and their methods emerge from the creative design process of dynamic object modeling. For example, if we started with the (trivial for explanation) makePayment sequence diagram in Figure 16.21, we see that a Register and Sale class definition in a class diagram can be obviously derived.


: Register
: Sale
makePayment(cashTendered)
makePayment(cashTendered)
messages in interaction diagrams indicate operations in the class diagrams
classes identified in the interaction diagrams are declared in the class diagrams
Register
...
makePayment(...)
...
1
currentSale
Sale
...
makePayment(...)
...

What are Patterns?

Experienced OO developers (and other software 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 named, may be called patterns.
Design patterns are reusable solutions to problems that recur in many applications. A pattern serves as a guide for creating a “good” design. Patterns are based on sound common sense and the application of

fundamental design principles. These are created by people who spot repeating themes across designs. The pattern solutions are typically described in terms of class and interaction diagrams. Examples of design patterns are expert pattern, creator pattern, controller pattern etc.

The pattern facilitates reuse of knowledge. Additionally, the pattern helps in the communication among software developers – using the name of the pattern conveys a large amount of knowledge in a very dense way.
A design pattern is a general repeatable solution to a commonly occurring problem in software design. A ***design pattern isn't a finished design that can be transformed directly into code***. It is a description or template for how to solve a problem that can be used in many different situations.
A pattern is usually described in ***four components***; these components explain what it is about and how it is to be used.

Components of Pattern
Pattern name: The pattern name is used to identify the pattern once it has been introduced. It is a way to communicate the pattern to other people and is therefore vital in spreading its reach, this is a fact mentioned in.
Problem description: In this section, the problem is described that is the reason for applying the pattern. It may be accompanied by a list of preconditions that must be fulfilled – only when these conditions are met, the pattern is applied.
Solution: Here the workings of the pattern are explained: which classes interact when and how these interactions are achieved. The description is on an abstract level to make sure that it can be applied in many situations.
Consequences: Here the effects of the pattern are explained. This might be both advantages and disadvantages of applying the pattern. The consequences are often related to the impact on flexibility, extensibility and portability the application of the pattern has

For example, here is a sample pattern:

Pattern Name: **Information Expert**

Problem: What is a basic principle by which to assign responsibilities to objects?

Solution: Assign a responsibility to the class that has the information needed to fulfill it.

In OO design, **a pattern** is a named description of a problem and solution that can be applied to new contexts; ideally, a pattern advises us on how to apply its solution in varying circumstances and considers the forces and trade-offs. Many patterns, given a specific category of problem, guide the assignment of responsibilities to objects.

Most simply, a good pattern is a named and well-known problem/solution pair that can be applied in new contexts, with advice on how to apply it in novel situations and discussion of its trade-offs, implementations, variations, and so forth.

Advantages

Naming a pattern, design idea, or principle has the following advantages:

- It supports chunking and incorporating that concept into our understanding and memory.
- It facilitates communication.

Design patterns are very useful in creating good software design solutions. In addition to providing the model of a good solution, design patterns include a clear specification of the problem, and also explain the circumstances in which the solution would and would not work.

Thus, a design pattern has four important parts:

- The problem.
- The context in which the problem occurs.
- The solution.
- The context within which the solution works.

Uses of Design Patterns

- Design patterns can speed up the development process by providing tested, proven development paradigms
- Design patterns provide general solutions, documented in a format that doesn't require specifics tied to a particular problem.
- Patterns allow developers to communicate using well-known, well understood names for software interactions. Common design patterns can be improved over time, making them more robust than ad-hoc designs.

The design pattern solutions are typically described in terms of class and interaction diagrams. These are often given in a parameterized form. We now describe a few important patterns:

1. Information Expert
2. Creator
3. Controller
4. Facade
5. Model view Separation pattern
6. Intermediary pattern or proxy

Applying GRASP to Object Design

GRASP stands for General Responsibility Assignment Software Patterns. The name was chosen to suggest the importance of grasping these principles to successfully design object-oriented software.
There are nine GRASP patterns:

- *Creator*
- *Information Expert*
- *Controller*
- *Low Coupling*
- *High Cohesion .*
- *Pure Fabrication*
- *Indirection*
- *Polymorphism*
- *Protected Variations*

1. **Creator**

Problem : Who should be responsible for creating a new instance of some class?

The creation of objects is one of the most common activities in an object-oriented system. Consequently, it is useful to have a general principle for the assignment of creation responsibilities. Assigned well, the design can support low coupling, increased clarity, encapsulation, and reusability.

Solution
Assign class B the responsibility to create an instance of class A if one of these is true (the more the better):

- B "contains" or compositely aggregates A.
- B records A.
- B closely uses A.
- B has the initializing data for A 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, usually prefer a class B which aggregates or contains class A.

2. **Information Expert**

Problem: Which class should be responsible for doing certain things?
Solution: Assign responsibility to the information expert – the class that has the information necessary to fulfill the required responsibility. The expert pattern expresses the common intuition that objects do things related to the information they have.

3. **Controller**

Problem: Who should be responsible for handling the actor requests?
Solution: For every use case, there should be a separate controller object which would be responsible for handling requests from the actor. Also, the same controller should be used for all the actor requests pertaining to one use case so that it becomes possible to maintain the necessary information about the state of the use case. The state information maintained by a controller can be used to identify the out-of-sequence actor requests, e.g. whether voucher request is received before arrange payment request.

4. **Façade Pattern:**

Problem: How should the services be requested from a service package?
Context in which the problem occurs: A package as already discussed is a cohesive set of classes – the classes have strongly related responsibilities. For example, an RDBMS interface package may contain classes that allow one to perform various operations on the RDBMS.
Solution: A class (such as DBfacade) can be created which provides a common interface to the services of the package.

5. **Model view Separation Pattern**

Problem: How should the non-GUI classes communicate with the GUI classes?
Context in which the problem occurs: This is a very commonly occurring pattern which occurs in almost every problem. Here, model is a synonym for the domain layer objects, view is a synonym for the presentation layer objects such as the GUI objects.
Solution: The model view separation pattern states that model objects should not have direct knowledge (or be directly coupled) to the view objects. This means that there should not be any direct calls from other objects to the GUI objects. This results in a good solution, because the GUI classes are related to a particular application whereas the other classes may be reused.
There are actually two solutions to this problem which work in different circumstances as follows:

Solution 1: Polling or Pull from above
It is the responsibility of a GUI object to ask for the relevant information from the other objects, i.e. the GUI objects pull the necessary information from the other objects whenever required.

Solution 2: Publish- subscribe pattern

An event notification system is implemented through which the publisher can indirectly notify the subscribers as soon as the necessary information becomes available. An event manager class can be defined which keeps track of the subscribers and the types of events they are interested in. An event is published by the publisher by sending a message to the event manager object.

6. **Intermediary Pattern or Proxy**

Problem: How should the client and server objects interact with each other?

Context in the problem occurs: The client and server terms as used here refer to software components existing across a network. The clients are consumers of services provided by the servers.

Solution: A proxy object at the client side can be defined which is a local sit-in for the remote server object. The proxy hides the details of the network transmission. The proxy is responsible for determining the server address, communicating the client request to the server, obtaining the server response and seamlessly passing that to the client. The proxy can also augment (or filter) information that is exchanged between the client and the server. The proxy could have the same interface as the remote server object so that the client feels as if it is interacting directly with the remote server object and the complexities of network transmissions are abstracted out.

7. **Low Coupling :** "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 . These elements include classes, subsystems, systems, and so on." The design principle of this pattern is to assign the responsibilities so that coupling remains low. This principle can often be broken down to knowledge: when classes can exist without having knowledge of each other,

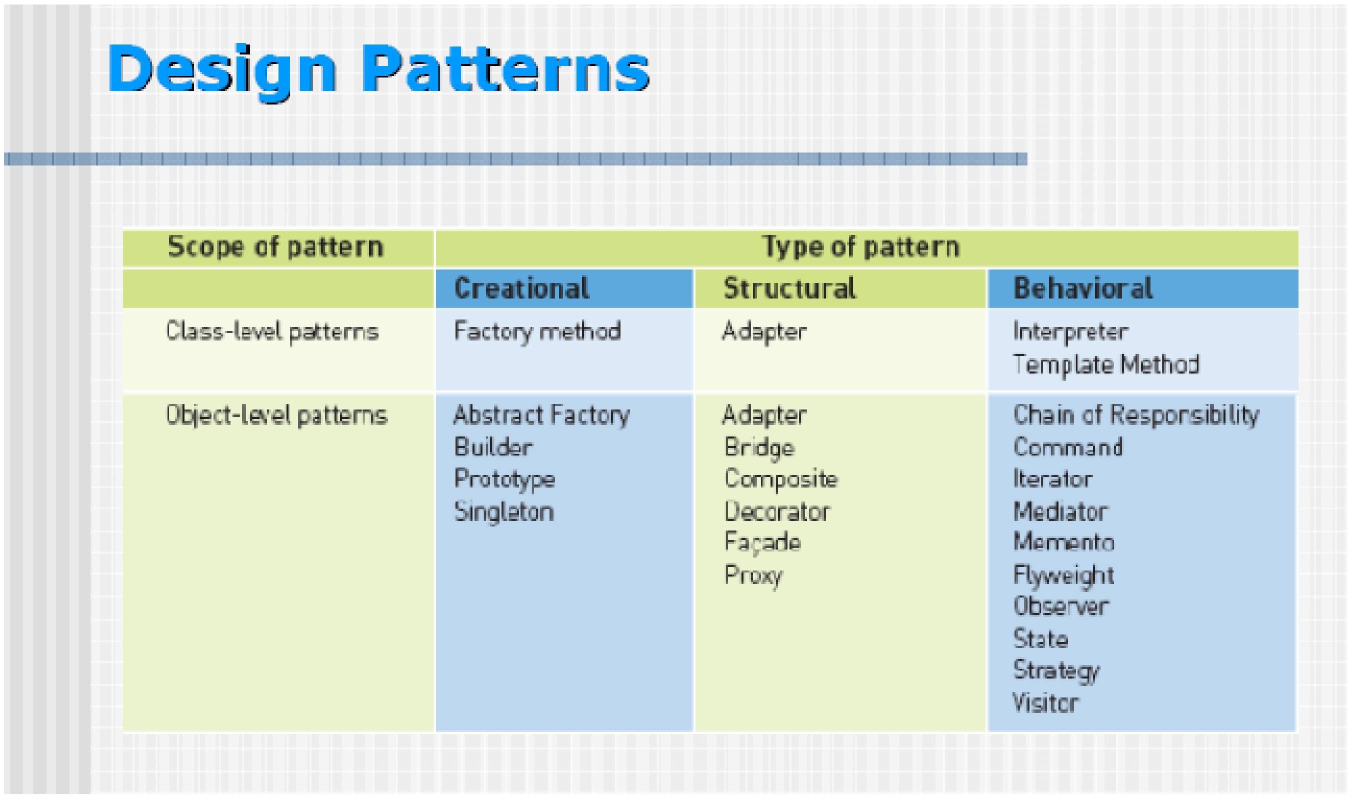
this knowledge should not be added. Low coupling is especially important with unstable objects. It should not be a problem to couple with highly stable business objects which are at the core of an application or with the Java API, but it can be a problem to couple with objects of the GUI which might change anytime

8. **Low Cohesion :** *"Cohesion is a measure of how strongly related and focused the responsibilities of an element are. An element with highly related responsibilities, an which does not do a tremendous amount of work, has high cohesion. These elements include classes, subsystems, and so on."*

It is undesirable in OOD and OOP to have classes which have hundreds of methods and attributes - for the sake of maintainability and usability it is necessary to partition the responsibilities of this class functionally and draw them out to different classes. This problem often arises with *factory classes*, being designed for the creation of objects. When the types of objects get numerous in a system, the factory class can consist of a collection of a hundred and more methods. The class becomes incomprehensible and unmaintainable. The solution is to partition of the factory class, in the same way as the objects being created are partitioned.

Gang of Four (GoF) Patterns in Object :

1. **Creational Pattern**
2. **Structural Pattern**
3. **Behavioral Pattern**



Design Patterns

Scope of pattern	Type of pattern		
	Creational	Structural	Behavioral
Class-level patterns	Factory method	Adapter	Interpreter Template Method
Object-level patterns	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Façade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

Creational design patterns

This design patterns is all about class instantiation. This pattern can be further divided into class-creation patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.

Structural design patterns

This design patterns is all about Class and Object composition. Structural class-creation patterns use inheritance to compose interfaces. Structural object-patterns define ways to compose objects to obtain new functionality.

Behavioral design patterns

This design patterns is all about Class's objects communication. Behavioral patterns are those patterns that are most specifically concerned with communication between objects.

For more Visit link :
http://www.oodesign.com/

Pattern Vs Design Pattern Vs Framework

The terms Pattern, Design Pattern and Framework are most often used interchangeably. However, these terms are not identical and there is a logical difference among their definitions. Here it goes:

Pattern:

A pattern is a way of doing something, or a way of pursuing intent. This idea applies to cooking, making fireworks, developing software, and to any other craft. It is a solution to a problem in a context. Patterns are classified into 1) Design Pattern, 2) Architectural Pattern, 3) Macro – Architecture, 4) Micro – Architecture, 5) Idioms or Coding Patterns, 6) Language Paradigms etc.

Design Pattern:

Design pattern is a category of patterns that deals with object oriented software. They represent solutions to problems that arise when developing software within a particular context. Design pattern captures the static and dynamic structure and collaboration among key participants in software designs. They can be used across different domains.

Framework:

Framework is made up of group of concrete classes which can be directly implemented on an existing platform. Frameworks are written in programming languages. It is a large entity comprising of several design patterns. Frameworks are concerned with specific application domain e.g. database, web application etc.

Above definition very much clarifies the difference among three. A design pattern is a type of pattern and is more like a concept, whereas a framework is something already coded to be used repetitively.

Patterns support reuse of software architecture and design
– Patterns capture the static and dynamic structures and collaborations of successful solutions to problems that arise when building applications in a particular domain
Frameworks support reuse of detailed design and code
– A framework is an integrated set of components that collaborate to provide a reusable architecture for a family of related application
Together, design patterns and frameworks help to improve software quality and reduce development time
– e.g., reuse, extensibility, modularity, performance