Unit 3
Elaboration

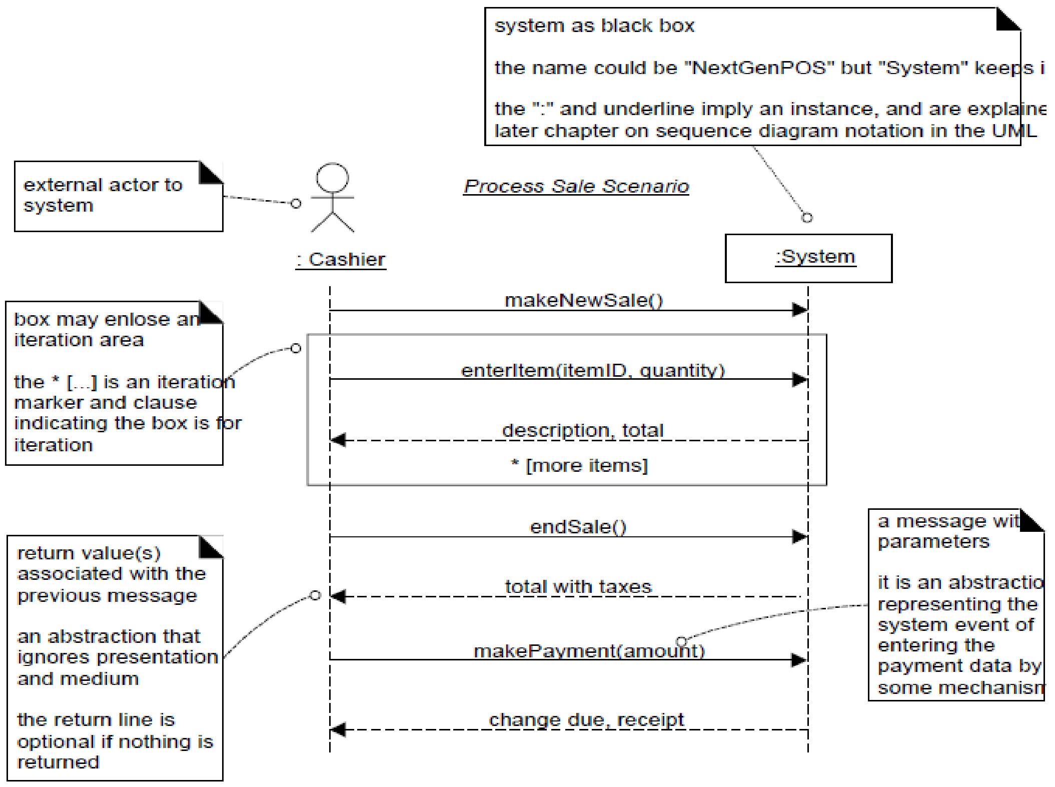
System sequence diagrams for use case model:

Use cases describe how external actors interact with the software system we are interested in creating. During this interaction an actor generates events to a system, usually requesting some operation in response. For example, when a cashier enters an item's ID, the cashier is requesting the POS system to record that item's sale. That request event initiates an operation upon the system. It is desirable to isolate and illustrate the operations that an external actor requests of a system, because they are an important part of understanding system behavior. The UML includes **sequence diagrams** as a notation that can illustrate actor interactions and the operations initiated by them.

A system sequence diagram (SSD) is a picture that shows, for a particular scenario of a use case, the events that external actors generate their order, and inter-system events. All systems are treated as a black box; the emphasis of the diagram is events that cross the system boundary from actors to systems.

Example of an SSD

An SSD shows, for a particular course of events within a use case, the external actors that interact directly with the system, the system (as a black box), and the system events that the actors generate. Time proceeds downward and the ordering of events should follow their order in the use case. System events may include parameters.


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

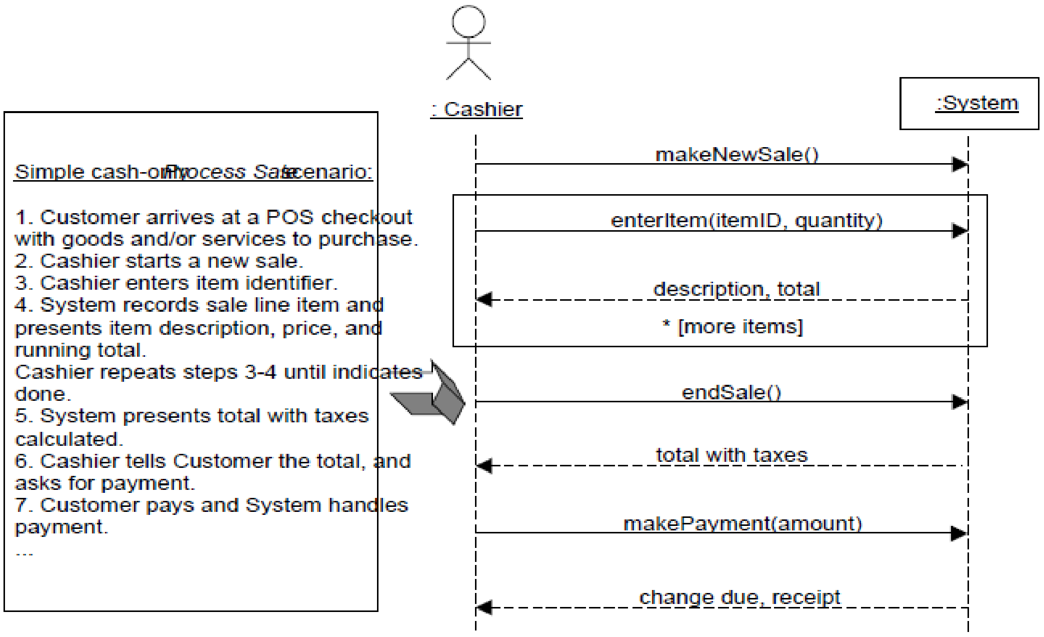
This example is for the main success scenario of the *Process Sale* use case. It indicates that the cashier generates *makeNewSale, enteritem, endSale,* and *make Payment* system events.

Inter-System SSDs:

SSDs can also be used to illustrate collaborations between systems, such as between the NextGen POS and the external credit payment authorizer. However, this is deferred until a later iteration in the case study, since this iteration does not include remote systems collaboration.

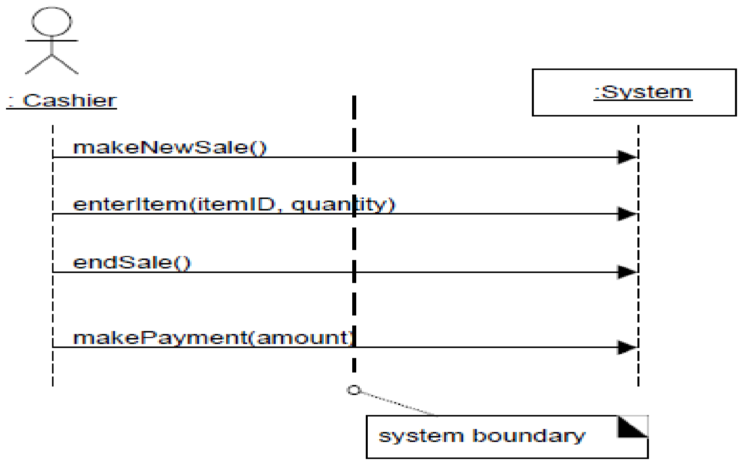
SSDs and Use Cases:

An SSD shows system events for a scenario of a use case therefore it is generated from inspection of a use case.


: Cashier
:System
Simple cash-only Process Sale scenario:
1. Customer arrives at a POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. System records sale line item and presents item description, price, and running total.
Cashier repeats steps 3-4 until indicates done.
5. System presents total with taxes calculated.
6. Cashier tells Customer the total, and asks for payment.
7. Customer pays and System handles payment.
...
makeNewSale()
enterItem(itemID, quantity)
description, total
* [more items]
endSale()
total with taxes
makePayment(amount)
change due, receipt

System Events and the System Boundary

To identify system events, it is necessary to be clear on the choice of system boundary. For the purposes of software development, the system boundary is usually chosen to be the software. (and possibly hardware) system itself; in this context, a system event is an external event that directly stimulates the software. Consider the *Process Sale* use case to identify system events. First, we must determine the actors that directly interact with the software system. The customer interacts with the cashier, but for this simple cash-only scenario, does not directly interact with the POS system—only the cashier does. Therefore, the customer is not a generator of system events; only the cashier is.

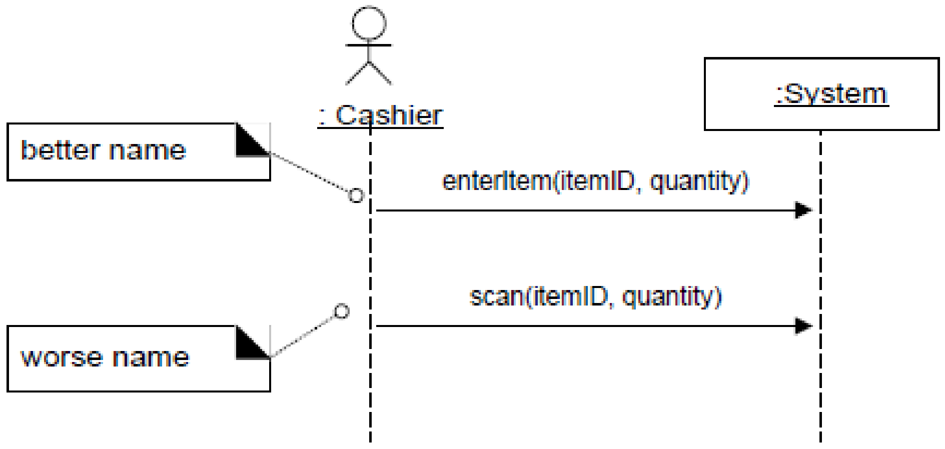


Naming System Events and Operations

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

It also improves clarity to start the name of a system event with a verb (add...,enter..., end..., make...), as in , since it emphasizes the command orientation of these events.

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

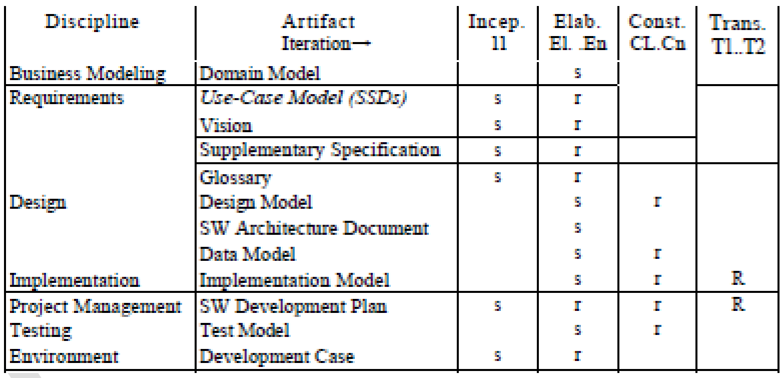


SSDs within the UP:

SSDs are part of the Use-Case Model—a visualization of the interactions implied in the use cases. SSDs were not explicitly mentioned in the original UP description, although the UP creators are aware of and understand the usefulness of such diagrams. SSDs are an example of the many possible skillful analysis and design artifacts or activities that the UP or RUP documents do not mention.

Inception—SSDs are not usually motivated in inception.

Elaboration—Most SSDs are created during elaboration, when it is useful to identify the details of the system events to clarify what major operations the system must be designed to handle, write system operation contracts , and possibly support estimation (for example, macro estimation with unadjusted function points and COCOMO II).

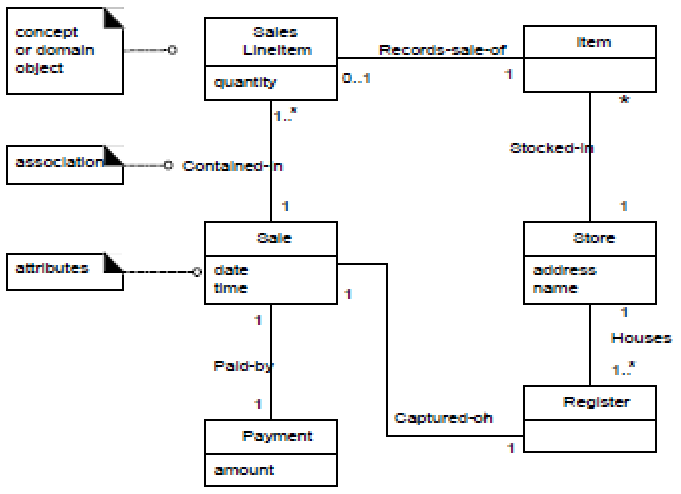


Discipline	Artifact Iteration→	Incep. I1	Elab. E1..En	Const. C1..Cn	Trans. T1..T2
Business Modeling	Domain Model		s		
Requirements	*Use-Case Model (SSDs)*	s	r		
	Vision	s	r		
	Supplementary Specification	s	r		
	Glossary	s	r		
Design	Design Model		s	r	
	SW Architecture Document		s		
	Data Model		s	r	
Implementation	Implementation Model		s	r	R
Project Management	SW Development Plan	s	r	r	R
Testing	Test Model		s	r	
Environment	Development Case	s	r		

Domain Model:

The quintessential object-oriented step in analysis or investigation is the decomposition of a domain of interest into individual conceptual classes or objects— the things we are aware of. A **domain model** is a *visual* representation of conceptual classes or real-world objects in a domain of interest. They have also been called **conceptual models**, **domain object models, and analysis object models.** The UP defines a Domain Model as one of the artifacts that may be created in the Business Modeling discipline. Using UML notation, a domain model is illustrated with a set of **class diagrams** in which no operations are defined. It may show:

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



Identifying conceptual Class:

Our goal is to create a domain model of interesting or meaningful conceptual classes in the domain of interest (sales). In this case, that means concepts related to the use case *Process Sale.* In iterative development, one incrementally builds a domain model over several iterations in the elaboration phase. In each, the domain model is limited to the prior and current scenarios under consideration, rather than a "big bang" model which early on attempts to capture all possible conceptual classes and relationships.

For example, this iteration is limited to a simplified cash-only *Process Sale* scenario; therefore, a partial domain model will be created to reflect just that—not more. The central task is therefore to identify conceptual classes related to the scenarios under design.

The following is a useful guideline in identifying conceptual classes:

It is better to over specify a domain model with lots of fine-grained conceptual classes than to underspecify it. Do not think that a domain model is better if it has fewer conceptual classes; quite the opposite tends to be true.

It is common to miss conceptual classes during the initial identification step, and to discover them later during the consideration of attributes or associations, or during design work. When found, they may be added to the domain model.

Do not exclude a conceptual class simply because the requirements do not indicate any obvious need to remember information about it (a criterion common in data modeling for relational database design, but not relevant to domain modeling), or because the conceptual class has no attributes. It is valid to have attribute less conceptual classes, or conceptual classes which have a purely behavioral role in the domain instead of an information role.

Strategies to Identify Conceptual Classes

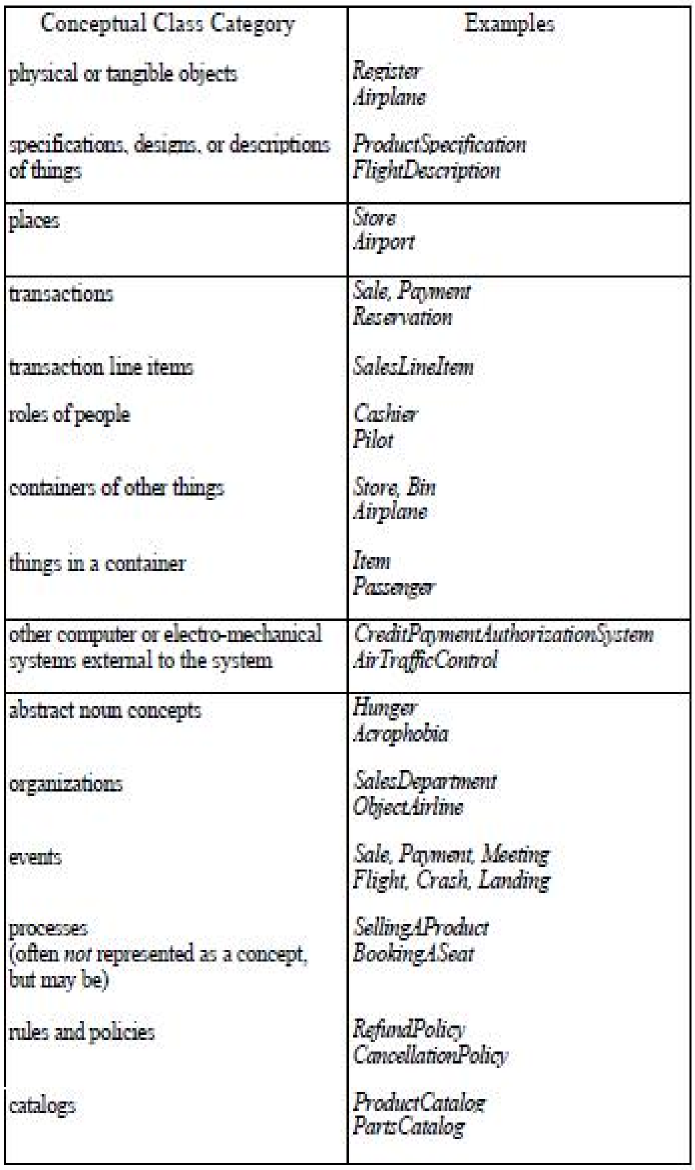
Two techniques are presented in the following sections:

1. Use a conceptual class category list.
2. Identify noun phrases.

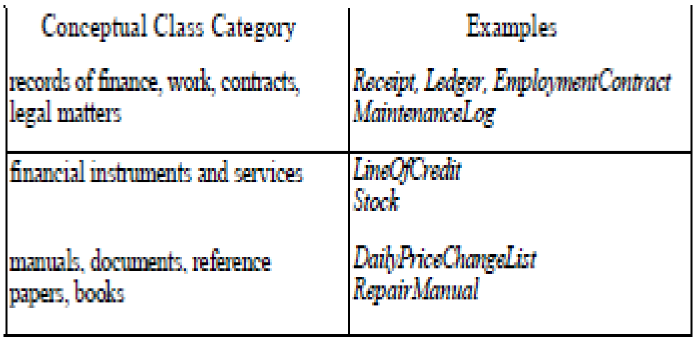
Another excellent technique for domain modeling is the use of **analysis patterns,** which are existing partial domain models created by experts, using published resources such as *Analysis Patterns* and *Data Model Patterns*.

Use a Conceptual Class Category List

Start the creation of a domain model by making a list of candidate conceptual classes. Table contains many common categories that are usually worth considering, though not in any particular order of importance. Examples are drawn from the store and airline reservation domains.



Conceptual Class Category	Examples
physical or tangible objects	*Register* *Airplane*
specifications, designs, or descriptions of things	*ProductSpecification* *FlightDescription*
places	*Store* *Airport*
transactions	*Sale, Payment* *Reservation*
transaction line items	*SalesLineItem*
roles of people	*Cashier* *Pilot*
containers of other things	*Store, Bin* *Airplane*
things in a container	*Item* *Passenger*
other computer or electro-mechanical systems external to the system	*CreditPaymentAuthorizationSystem* *AirTrafficControl*
abstract noun concepts	*Hunger* *Acrophobia*
organizations	*SalesDepartment* *ObjectAirline*
events	*Sale, Payment, Meeting* *Flight, Crash, Landing*
processes (often *not* represented as a concept, but may be)	*SellingAProduct* *BookingASeat*
rules and policies	*RefundPolicy* *CancellationPolicy*
catalogs	*ProductCatalog* *PartsCatalog*



Conceptual Class Category	Examples
records of finance, work, contracts, legal matters	*Receipt, Ledger, EmploymentContract* *MaintenanceLog*
financial instruments and services	*LineOfCredit* *Stock*
manuals, documents, reference papers, books	*DailyPriceChangeList* *RepairManual*

Finding Conceptual Classes with Noun Phrase Identification

Another useful technique (because of its simplicity) suggested in is linguistic analysis: identify the nouns and noun phrases in textual descriptions of a domain, and consider them as candidate conceptual classes or attributes. Care must be applied with this method; a mechanical noun-to-class mapping isn't possible, and words in natural languages are ambiguous.

Nevertheless, it is another source of inspiration. The fully dressed use cases are an excellent description to draw from for this analysis. For example, the current scenario of the *Process Sale* use case can be used.

Main Success Scenario (or Basic Flow):

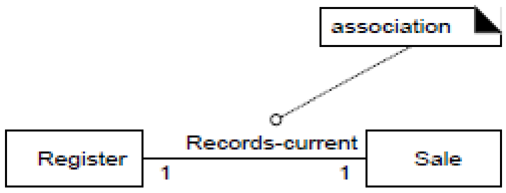
1. Customer arrives at a **POS checkout** with **goods** and/or **services** to purchase.

2. Cashier starts a new **sale.**
3. Cashier enters **item identifier.**
4. System records **sale line item** and presents **item description, price,** and running **total.** Price calculated from a set of price rules.
Cashier repeats steps 2-3 until indicates done.
5. System presents total with **taxes** calculated.
6. Cashier tells Customer the total, and asks for **payment.**
7. Customer pays and System handles payment.
8. System logs the completed **sale** and sends sale and payment information to the external **Accounting** (for accounting and **commissions)** and **Inventory** systems (to update inventory).
9. System presents **receipt.**
10. Customer leaves with receipt and goods (if any).
Extensions (or Alternative Flows):
7a. Paying by cash:
1. Cashier enters the cash **amount tendered.**
2. System presents the **balance due,** and releases the **cash drawer.**
3. Cashier deposits cash tendered and returns balance in cash to Customer.
4. System records the cash payment.
The domain model is a visualization of noteworthy domain concepts and vocabulary.
Where are those terms found?

In the use cases. Thus, they are a rich source to mine via noun phrase identification. Some of these noun phrases are candidate conceptual classes, some may refer to conceptual classes that are ignored in this iteration (for example, "Accounting" and "commissions"), and some may be attributes of conceptual classes.A weakness of this approach is the imprecision of natural language; different noun phrases may represent the same conceptual class or attribute, among other ambiguities. Nevertheless, it is recommended in combination with the *Conceptual Class Category List* technique.

Adding associations

An **association** is a relationship between types (or more specifically, instances of those types) that indicates some meaningful and interesting connection. In the UML associations are defined as "the semantic relationship between two or more classifiers that involve connections among their instances."



Criteria for Useful Associations

Associations worth noting usually imply knowledge of a relationship that needs to be preserved for some duration—it could be milliseconds or years, depending on context. In other words, between what objects do we need to have some memory of a relationship? For example, do we need to remember what *SalenLineItem* instances are associated with a *Sale* instance? Definitely, otherwise it would not be possible to reconstruct a sale, print a receipt, or calculate a sale total.
Consider including the following associations in a domain model:

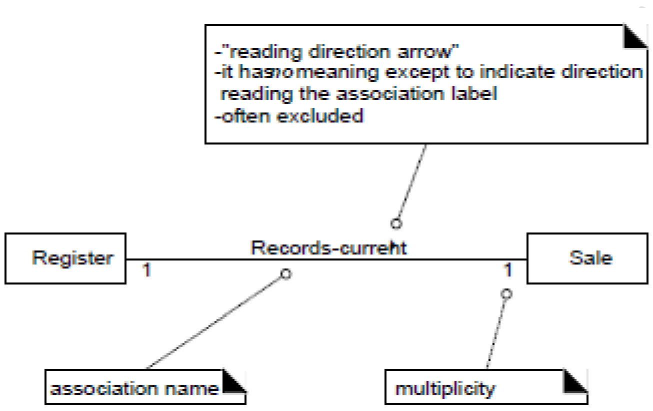
- Associations for which knowledge of the relationship needs to be preserved for some duration ("need-to-know" associations).
- Associations derived from the Common Associations List.

By contrast, do we need to have memory of a relationship between a current *Sale* and a *Manager?* No, the requirements do not suggest that any such relationship is needed. It is not wrong to show a relationship between *a Sale* and *Manager,* but it is not compelling or useful in the context of our requirements.

This is an important point. On a domain model with n different conceptual classes, there can be n-(n-l) associations to other conceptual classes—a potentially large number. Many lines on the diagram will add "visual noise" and make it less comprehensible. Therefore, be parsimonious about adding association lines.

The UML Association Notation

An association is represented as a line between classes with an association name. The association is inherently bidirectional, meaning that from instances of either class, logical traversal to the other is possible.


-"reading direction arrow"
-it has no meaning except to indicate direction reading the association label
-often excluded
Register
Records-current
Sale
1
1
association name
multiplicity

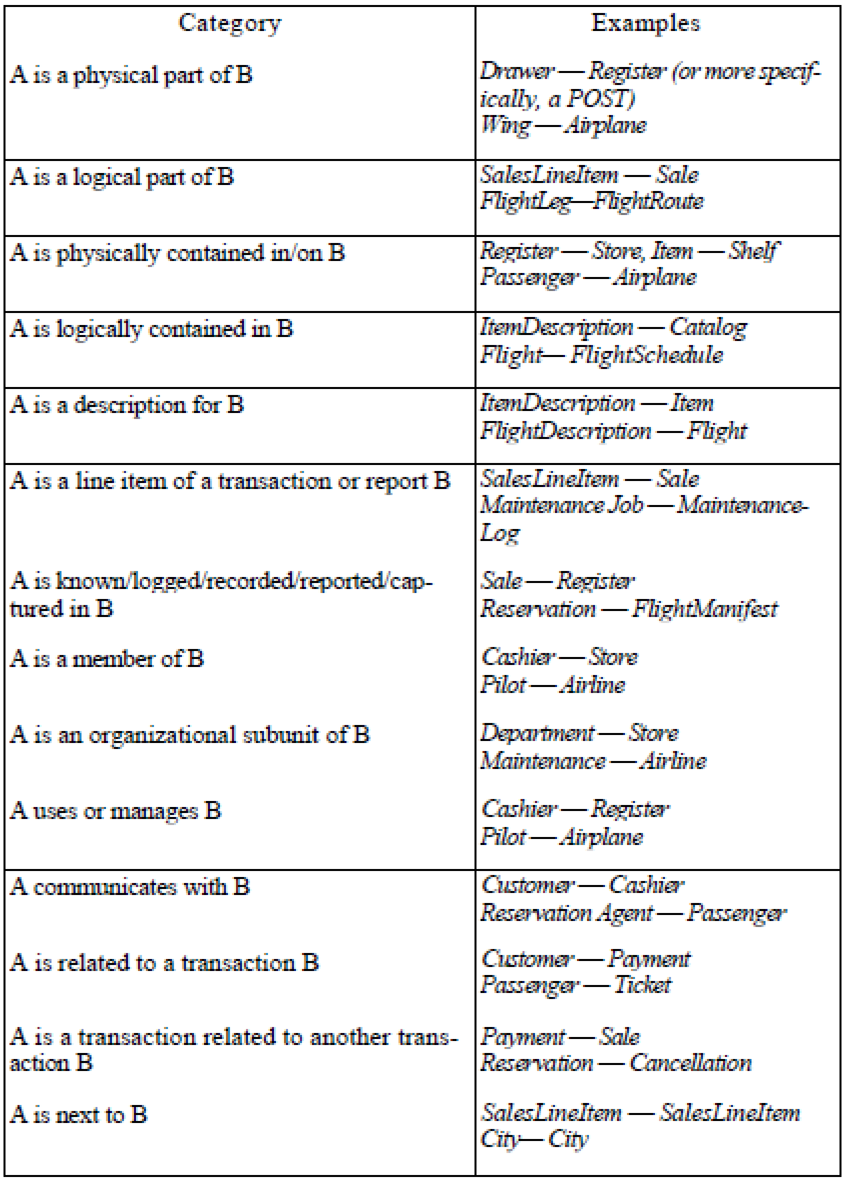
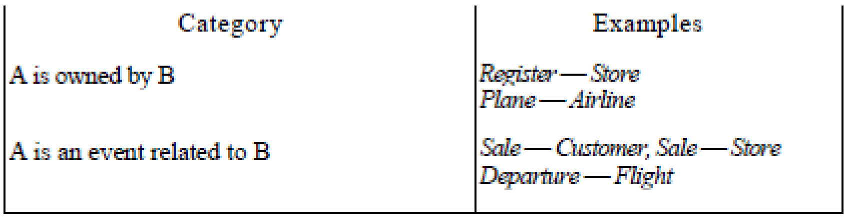
This traversal is purely abstract; it is *not a* statement about connections between software entities. The ends of an association may contain a multiplicity expression indicating the numerical relationship between instances of the classes.

An optional "reading direction arrow" indicates the direction to read the association name; it does not indicate direction of visibility or navigation.

If not present, it is conventional to read the association from left to right or top to bottom, although the UML does not make this a rule.

Finding Associations—Common Associations List

It contains common categories that are usually worth considering. Examples are drawn from the store and airline reservation domains.

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

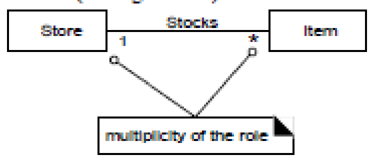
Roles

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

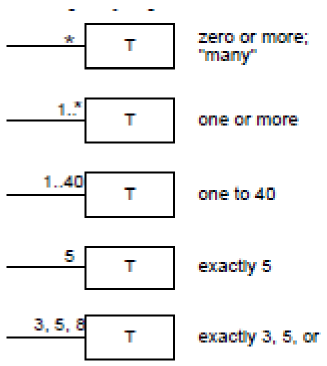
- name
- multiplicity expression
- navigability

Multiplicity

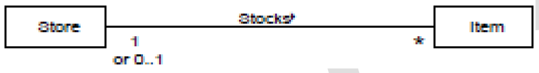
Multiplicity defines how many instances of a class *A* can be associated with one instance of a class *B*.



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



The multiplicity value communicates how many instances can be validly associated with another, at a particular moment, rather than over a span of time. For example, it is possible that a used car could be repeatedly sold back to used car dealers over time. But at any particular moment, the car is only *Stocked-by* one dealer. The car is not *Stocked-by* many dealers at any particular moment. Similarly, in countries with monogamy laws, a person can be *Married-to* only one other person at any particular moment, even though over a span of time, they may be married to many persons.
The multiplicity value is dependent on our interest as a modeler and software developer, because it communicates a domain constraint that will be (or could be) reflected in software.



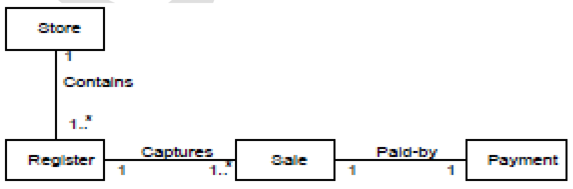
Rumbaugh gives another example *of Person* and *Company* in the *Works-for* association. Indicating if a *Person* instance works for one or many *Company* instances is dependent on the context of the model; the tax department is interested in *many;* a union probably only *one.* The choice usually practically depends on whom we are building the software for, and thus the valid multiplicities in an implementation.

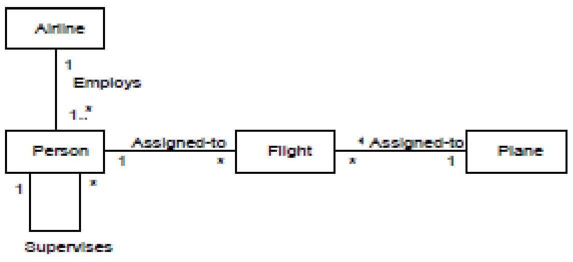
Naming Associations

Association names should start with a capital letter, since an association represents a classifier of links between instances; in the UML, classifiers should start with a capital letter. Two common and equally legal formats for a compound association name are:

• *Paid-by*
• *PaidBy*

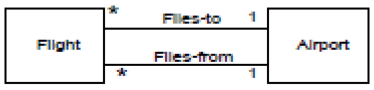
In Figure, the default direction to read an association name is left to right or top to bottom. This is not a UML default, but a common convention.





Multiple Associations between Two Types

Two types may have multiple associations between them; this is not uncommon. There is no outstanding example in our POS case study, but an example from the domain of the airline is the relationships between a *Flight* (or perhaps more precisely, a *FlightLeg)* and *an Airport*; the flying-to and flyingfrom associations are distinctly different relationships, which should be shown separately.



Associations and Implementation

During domain modeling, an association is *not* a statement about data flows, instance variables, or object connections in a software solution; it is a statement that a relationship is meaningful in a purely conceptual sense—in the real world. Practically speaking, many of these relationships will typically be implemented in software as paths of navigation and visibility (both in the Design Model and Data Model), but their presence in a conceptual (or essential) view of a domain model does not require their implementation.

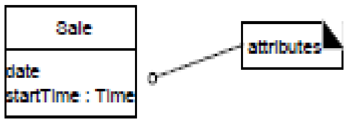
When creating a domain model, we may define associations that are not necessary during implementation. Conversely, we may discover associations that need to be implemented but were missed during domain modeling. In these cases, the domain model can be updated to reflect these discoveries. Later on we will discuss ways to implement associations in an object-oriented programming language (the most common is to use an attribute that references an instance of the associated class), but for now, it is valuable to think of them as purely conceptual expressions, *not* statements about a database or software solution. As always, deferring design considerations frees us from extraneous information and decisions while doing pure "analysis" investigations and maximizes our design options later on.

Adding Attributes

An attribute is a logical data value of an object. For example, a receipt (which reports the information of a sale) normally includes a date and time, and management wants to know the dates and times of sales for a variety of reasons. Consequently, the *Sale* conceptual class needs a *date* and *time* attribute.

UML Attribute Notation

Attributes are shown in the second compartment of the class box. Their type may optionally be shown.

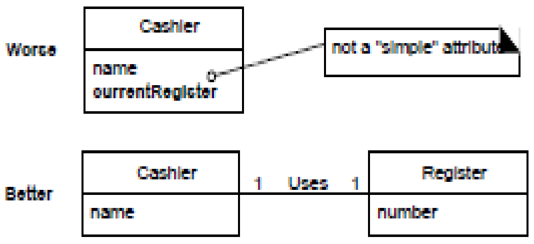


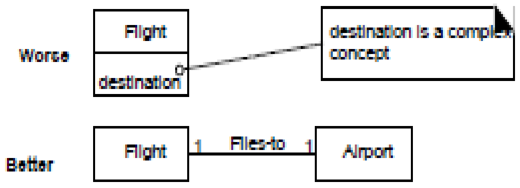
Valid Attribute Types

There are some things that should not be represented as attributes, but rather as associations. This section explores valid attributes.

Keep Attributes Simple

Intuitively, most simple attribute types are what are often thought of as primitive data types, such as numbers. The type of an attribute should not normally be a complex domain concept, such as a *Sale or Airport.* For example, the following *currentRegister* attribute in the *Cashier* class in Figure is undesirable because its type is meant to be a *Register,* which is not a simple attribute type (such as *Number* or *String).* The most useful way to express that a *Cashier* uses a *Register* is with an association, not with an attribute..





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

Conceptual vs. Implementation Perspectives

The restriction that attributes in the domain model be only of simple data types does *not* imply that C++ or Java attributes (data members, instance fields) must only be of simple, primitive data types. The domain model focuses on pure conceptual statements about a problem domain, not software components.

Later, during design and implementation work, it will be seen that the associations between objects expressed in the domain model will often be implemented as attributes that reference other complex software objects. However, this is but one of a number of possible design solutions to implement an association, and so the decision should be deferred during domain modeling.

Data Types

Attributes should generally be **data types.** This is a UML term that implies a set of values for which unique identity is not meaningful (in the context of our model or system). For example, it is not (usually) meaningful to distinguish between:

- Separate instances of the *Number* 5.
- Separate instances of the *String* 'cat'.
- Separate instances of *PhoneNumber* that contain the same number.
- Separate instances *of Address* that contain the same address.

By contrast, it *is* meaningful to distinguish (by identity) between two separate instances of a *Person* whose names are both "Jill Smith" because the two instances can represent separate individuals with the same name. In terms of software, there are few situations where one would compare the memory addresses of instances of *Number, String, PhoneNumber,* or *Address;* only value-based comparisons are relevant. By contrast, it is conceivable to compare the memory addresses *of Person* instances, and to distinguish them, even if they had the same attribute values, because their unique identity is important. Thus, all primitive types (number, string) are UML data types, but not all data types are primitives. For example, *PhoneNumber* is a non-primitive data type.

These data type values are also known as **value objects.** The notion of data types can get subtle. As a rule of thumb, stick to the basic test of "simple" attribute types: Make it an attribute if it is naturally thought of as number, string, boolean, date, or time (and so on); otherwise, represent it as a separate conceptual class.

Non-primitive Data Type Classes

The type of an attribute may be expressed as a non-primitive class in its own right in a domain model. For example, in the POS system there is an item identifier. It is typically viewed as just a number. So should it be represented as a non-primitive class? Apply this guideline:

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

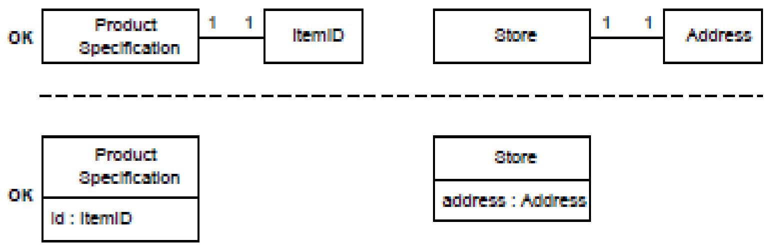
- It is composed of separate sections.
    - phone number, name of person
- There are operations usually associated with it, such as parsing or validation.
    - social security number
- It has other attributes.
    - promotional price could have a start (effective) date and end

- date
- It is a quantity with a unit.
    - payment amount has a unit of currency
- It is an abstraction of one or more types with some of these qualities.
    - item identifier in the sales domain is a generalization of types
- such as Universal Product Code (UPC) or European Article
- Number (EAN)
- Applying these guidelines to the POS domain model attributes yields the following
- analysis:
- The item identifier is an abstraction of various common coding schemes,
- including UPC-A, UPC-E, and the family of EAN schemes. These numeric
- coding schemes have subparts identifying the manufacturer, product, country
- (for EAN), and a check-sum digit for validation. Therefore, there should
- be a non-primitive *ItemID* class, because it satisfies many of the guidelines
- above.
- The *price* and *amount* attributes should be non-primitive *Quantity* or *Money*
- classes because they are quantities in a unit of currency.
- The *address* attribute should be a non-primitive *Address* class because it
- has separate sections.

The classes *ItemID, Address,* and *Quantity* are data types (unique identity of instances is not meaningful) but they are worth considering as separate classes because of their qualities.

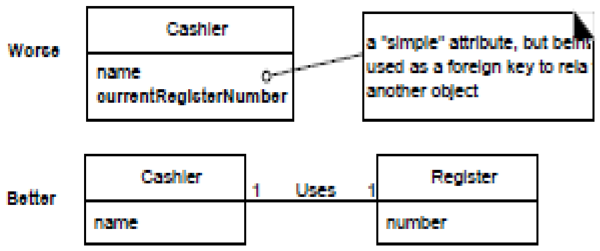
Where to Illustrate Data Type Classes?

Should the *ItemID* class be shown as a separate conceptual class in a domain model? It depends on what you want to emphasize in the diagram. Since *ItemID* is a *data type* (unique identity of instances is not important), it may be shown in the attribute compartment of the class box, as shown in Figure. But since it is a non-primitive class, with its own attributes and associations, it may be interesting to show it as a conceptual class in its own box. There is no correct answer; it depends on how the domain model is being used as a tool of communication, and the significance of the concept in the domain.



Design Creep: No Attributes as Foreign Keys

Attributes should not be used to relate conceptual classes in the domain model. The most common violation of this principle is to add a kind of **foreign key attribute,** as is typically done in relational database designs, in order to associate two types. For example, in Figure the *currentRegisterNumber* attribute in the *Cashier* class is undesirable because its purpose is to relate the *Cashier* to a *Register* object. The better way to express that a *Cashier* uses a *Register* is with an association, not with a foreign key attribute. Once again, relate types with an association, not with an attribute. There are many ways to relate objects—foreign keys being one—and we will defer how to implement the relation until design, in order to avoid **design creep.**


Worse
Cashier
name
currentRegisterNumber
a "simple" attribute, but being used as a foreign key to relate another object
Better
Cashier
name
1
Uses
1
Register
number

Interaction Diagrams

Sequence and Collaboration Diagrams

The term *interaction diagram*, is a generalization of two more specialized UML diagram types; both can be used to express similar message interactions:

- collaboration diagrams
- sequence diagrams

Throughout the book, both types will be used, to emphasize the flexibility in choice.

Collaboration diagrams illustrate object interactions in a graph or network format, in which objects can be placed anywhere on the diagram, as shown in Figure.

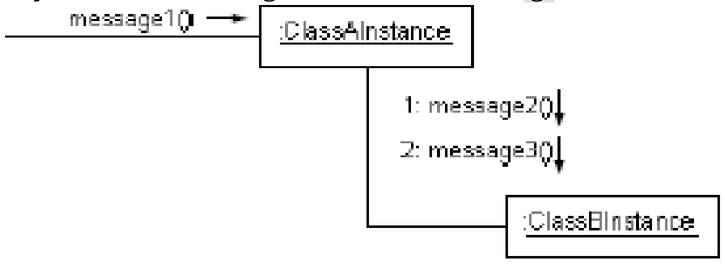
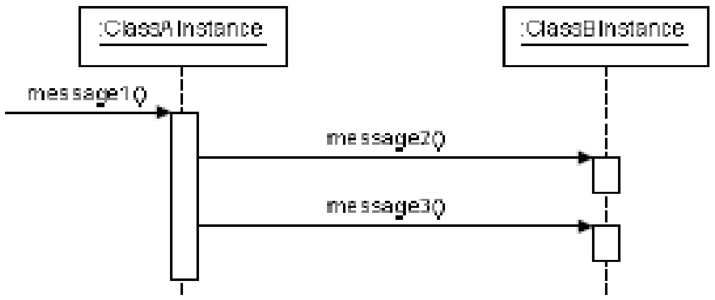
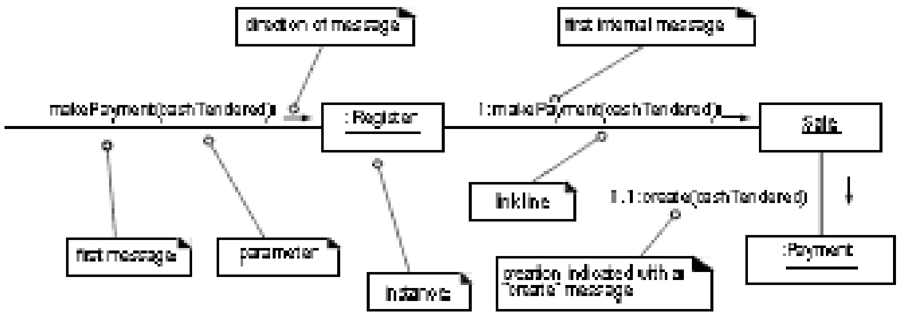


Figure 15.1 Collaboration diagram



Each type has strengths and weaknesses. When drawing diagrams to be published on pages of narrow width, collaboration diagrams have the advantage of allowing vertical expansion for new objects; additional objects in a sequence diagrams must extend to the right, which is limiting. On the other hand, collaboration diagram examples make it harder to easily see the sequence of messages. Most prefer sequence diagrams when using a CASE tool to reverse engineer source code into an interaction diagram, as they clearly illustrate the sequence of messages.

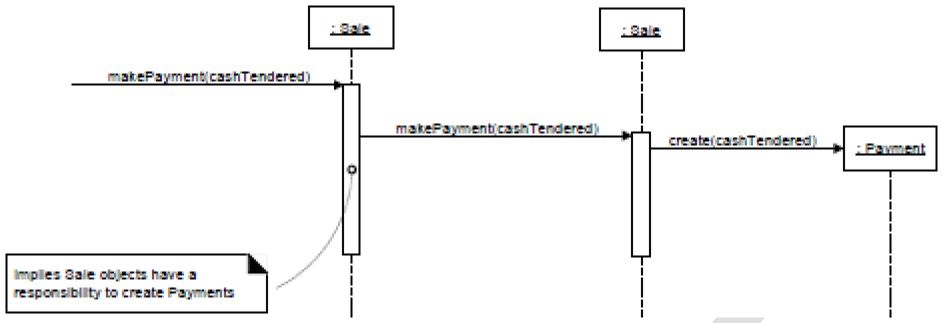
Example Collaboration Diagram: makePayment



The collaboration diagram shown in Figureis read as follows:

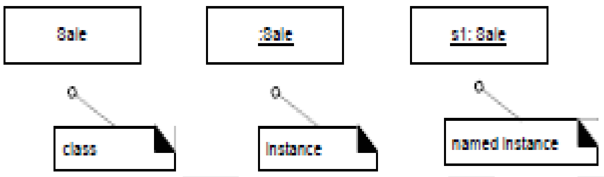
1. The message *makePayment is* sent to an instance of a *Register.* The sender is

not identified.
2. The *Register* instance sends the *makePayment* message to a *Sale* instance.
3. The *Sale* instance creates an instance of a *Payment.*
Example Sequence Diagram: makePayment

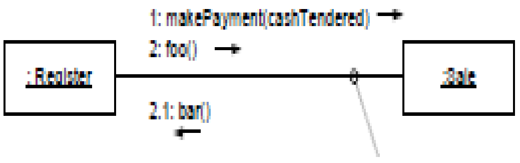


: Sale
: Sale
makePayment(cashTendered)
makePayment(cashTendered)
create(cashTendered)
: Payment
implies Sale objects have a responsibility to create Payments

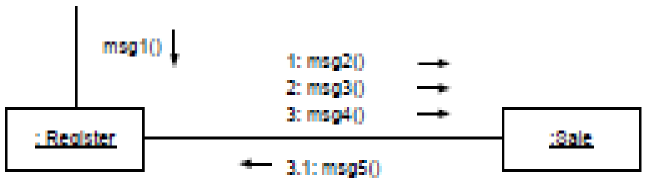
Common Interaction Diagram Notation
Illustrating Classes and Instances
The UML has adopted a simple and consistent approach to illustrate **instances** vs. classifiers:
For any kind of UML element (class, actor, ...), an instance uses the same graphic symbol as the type, but the designator string is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



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.

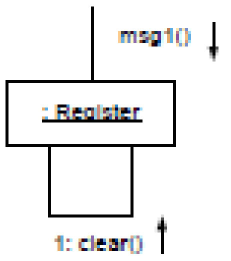


Messages
Each message between objects is represented with a message expression and small arrow indicating the direction of the message. Many messages may flow along this link. A sequence number is added to show the sequential order of messages in the current thread of control.



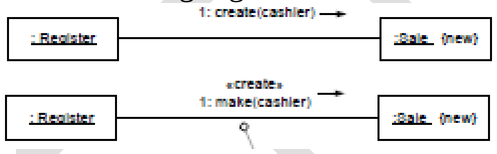
Messages to "self" or "this"

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



Creation of Instances

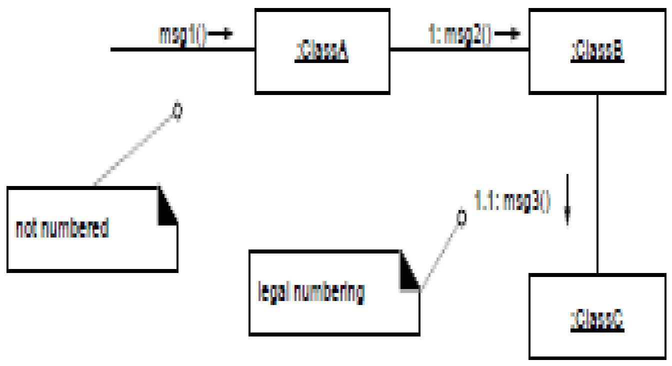
Any message can be used to create an instance, but there is a convention in the UML to use a message named *create* for this purpose. If another (perhaps less obvious) message name is used, the message may be annotated with a special feature called a UML stereotype, like so: *«create»*. The *create* message may include parameters, indicating the passing of initial values. This indicates, for example, a constructor call with parameters in Java. Furthermore, the UML property *{new}* may optionally be added to the instance box to highlight the creation.

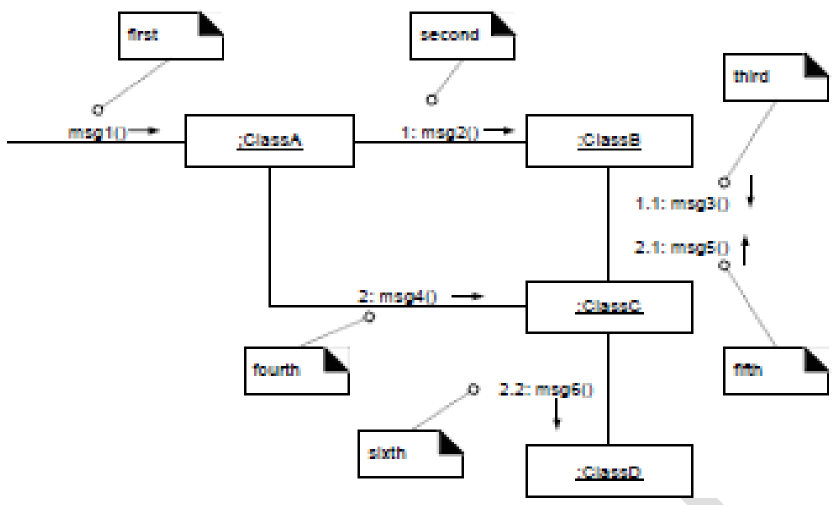


Message Number Sequencing

The order of messages is illustrated with **sequence numbers,** as shown in Figure. The numbering scheme is:

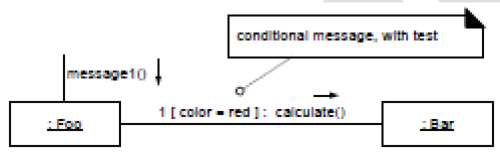
1. The first message is not numbered. Thus,*msg1()* is unnumbered.
2. The order and nesting of subsequent messages is shown with a legal numbering scheme in which nested messages have a number appended to them. Nesting is denoted by prepending the incoming message number to the out going message number.





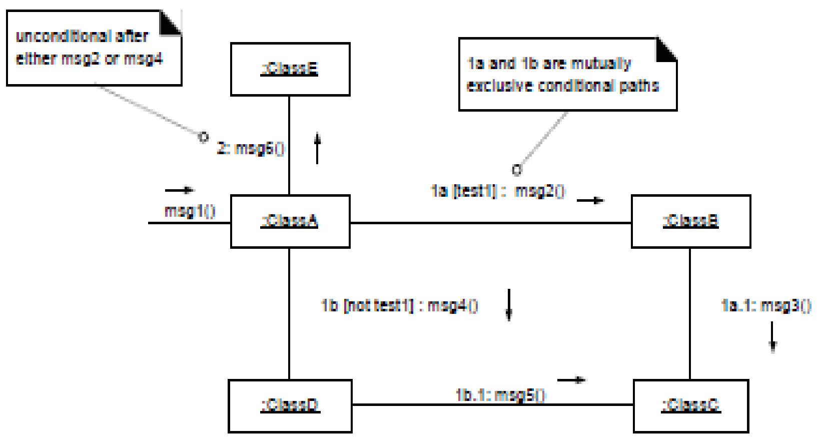
Conditional Messages

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

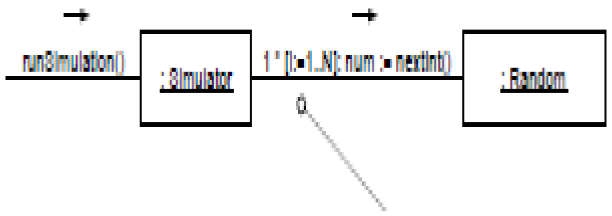
Mutually Exclusive Conditional Paths

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



Iteration or Looping

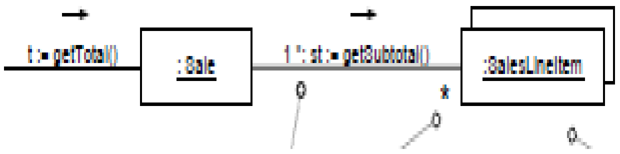
Iteration notation is shown in Figure. If the details of the iteration clause are not important to the modeler, a simple ’*’ can be used



Iteration Over a Collection (Multiobject)

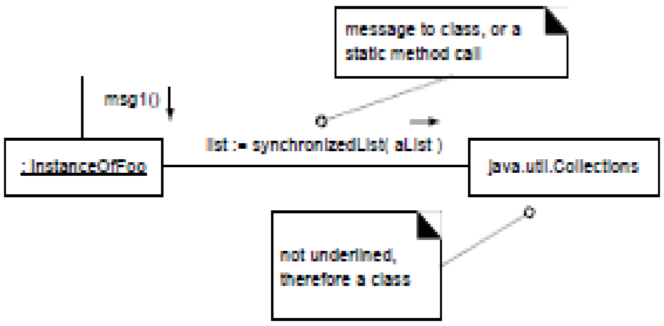
A common algorithm is to iterate over all members of a collection (such as a list or map), sending a message to each. Often, some kind of iterator object is ultimately used, such as an implementation of *java.util.Iterator* or a C++ standard library iterator. In the UML, the term

multiobject is used to denote a set of instances. a collection. In collaboration diagrams, this can be summarized as shown in Figure.



Messages 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 to class, or a static method call
msg1()
list := synchronizedList(aList)
: InstanceOfFoo
java.util.Collections
not underlined, therefore a class

Consequently, it is important to be consistent in underlining your instance names when an instance is intended, otherwise messages to instances versus classes may be incorrectly interpreted.

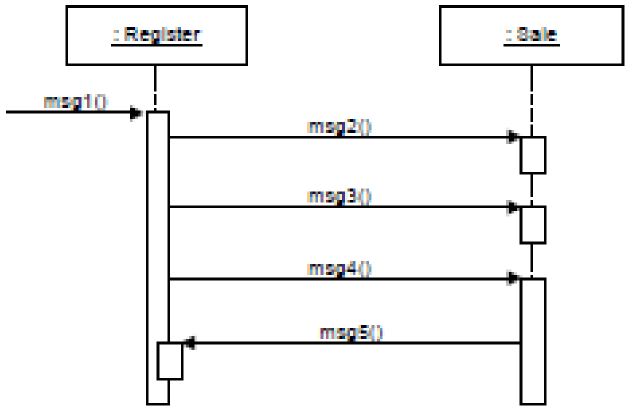
Basic Sequence Diagram Notation

Links

Unlike collaboration diagrams, sequence diagrams do not show links.

Messages

Each message between objects is represented with a message expression on an arrowed line between the objects. The time ordering is organized from top to bottom.

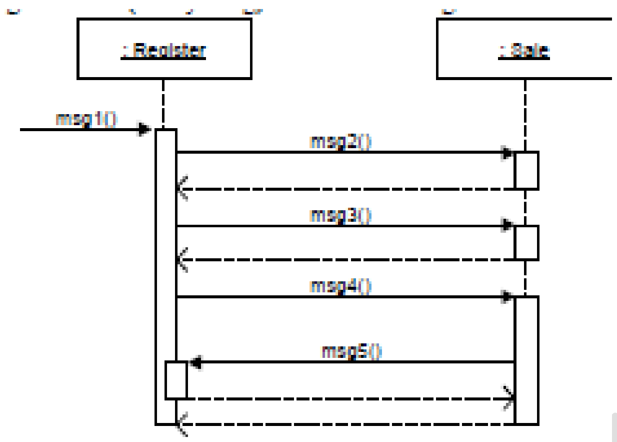


Focus of Control and Activation Boxes

As illustrated in Figure, sequence diagrams may also show the focus of control (that is, in a regular blocking call, the operation is on the call stack) using an **activation box.** The box is optional, but commonly used by UML practitioners.

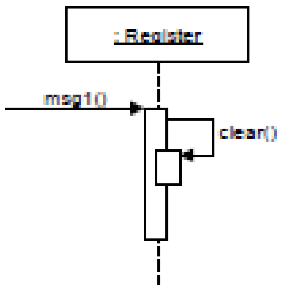
Illustrating Returns

A sequence diagram may optionally show the return from a message as a dashed open-arrowed line at the end of an activation box. Many practitioners exclude them. Some annotate the return line to describe what is being returned (if anything) from the message.

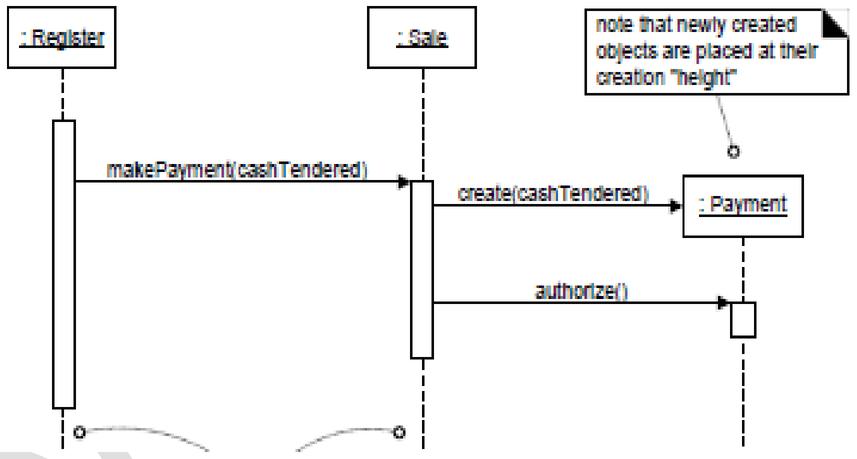


Messages to "self" or "this"

A message can be illustrated as being sent from an object to itself by using a nested activation box.



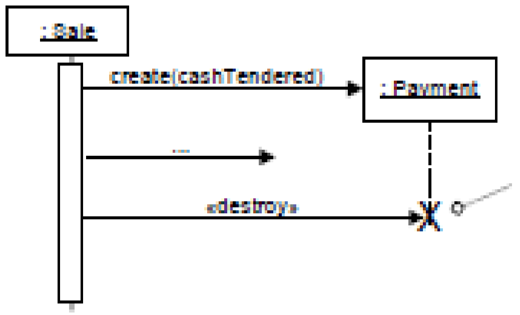
Creation of Instances



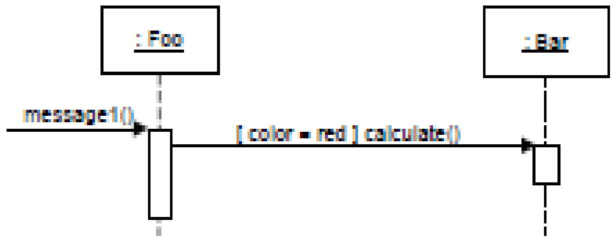
: Register
: Sale
note that newly created objects are placed at their creation "height"
makePayment(cashTendered)
create(cashTendered)
: Payment
authorize()

Object Lifelines and Object Destruction

Figure also illustrates **object lifelines**. The vertical dashed lines underneath the objects. These indicate the extent of the life of the object in the diagram. In some circumstances it is desirable to show explicit destruction of an object (as in C++, which does not have garbage collection); the UML lifeline notation provides a way to express this destruction.

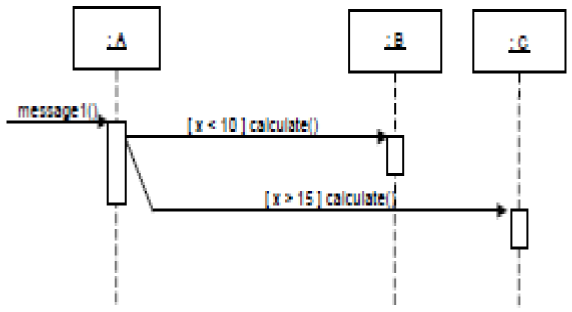


Conditional Messages



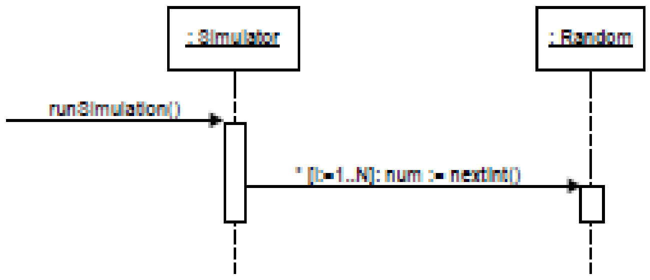
Mutually Exclusive Conditional Messages

The notation for this case is a kind of angled message line emerging from a common point, as illustrated in Figure.



Iteration for a Single Message

Iteration notation for one message is shown in Figure.



Iteration of a Series of Messages

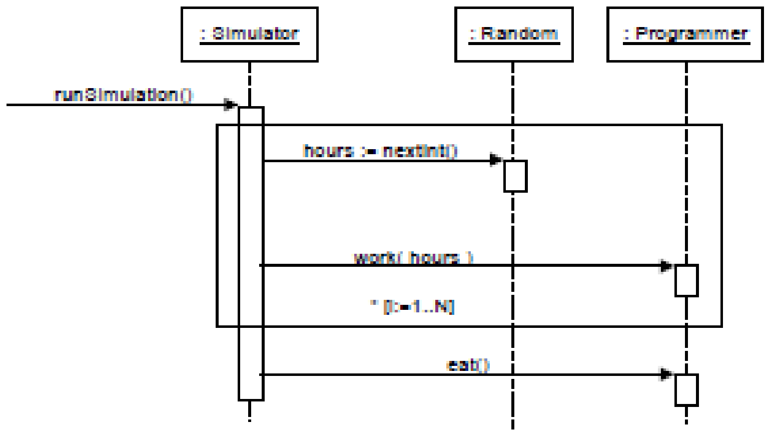
Notation to indicate iteration around a series of messages is shown in Figure.

Iteration Over a Collection (Multiobject)

In sequence diagrams, iteration over a collection is shown in Figure. With collaboration diagrams the UML specifies a ’*’ multiplicity marker at the end of the role (next to the multiobject) to indicate sending a message to each element rather than repeatedly to the collection itself. However, the UML does not specify how to indicate this with sequence diagrams.

Messages to Class Objects

As in a collaboration diagram, class or static method calls are shown by not underlining the name of the classifier, which signifies a class object rather than an instance.



<u>Introduction to GRASP design pattern</u>

GRASP as a Methodical Approach to Learning Basic Object Design

It *is* possible to communicate the detailed principles and reasoning required to grasp basic object design, and to learn to apply these in a methodical approach that removes the magic and vagueness. The GRASP patterns are a learning aid to help one understand essential object design, and apply design reasoning in a methodical, rational, explainable way. This approach to understanding and using design principles is based on *patterns of assigning responsibilities.*

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:

- knowing
- doing

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, We may declare that "a *Sale* is responsible for creating *SalesLineltems"* (a doing), or "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. The responsibility to "provide access to relational databases" may involve dozens of classes and hundreds of methods, packaged in a subsystem. By contrast, the 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. For example, 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 *agetSubtotal* message to each *SalesLineltem* object asking for its subtotal.

Patterns

Experienced object-oriented 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 given a name, may be called **patterns.** For example, here is a sample pattern:
Pattern Name: Information Expert
Solution: Assign a responsibility to the class that has the information needed to fulfill it.
Problem It Solves: What is a basic principle by which to assign responsibilities to objects?

In object technology, a **pattern** is a named description of a problem and solution that can be applied to new contexts; ideally, it provides advice in how to apply it in varying circumstances, and considers the forces and trade-offs. Many patterns provide guidance for how responsibilities should be assigned to objects, given a specific category of problem.

Most simply, a **pattern** is a named problem/solution pair that can be applied in new context, with advice on how to apply it in novel situations and discussion of its trade-offs. "One person's pattern is another person's primitive building block" is an object technology adage illustrating the vagueness of what can be called a pattern. This treatment of patterns will bypass the issue of what is appropriate to label a pattern, and focus on the pragmatic value of using the pattern style as a vehicle for naming, presenting, learning, and remembering useful software engineering principles.

Repeating Patterns

New pattern could be considered an oxymoron, if it describes a new idea. The very term "pattern" is meant to suggest a repeating thing. The point of patterns is not to express new design ideas. Quite the opposite is true. Patterns attempt to codify *existing* tried-and-true knowledge, idioms, and principles; the more honed and widely used, the better.

Consequently, the GRASP patterns which will soon be introduced do not state new ideas; they are a codification of widely used basic principles. To an object expert, the GRASP patterns.by idea if not by name will appear very fundamental and familiar. That's the point!

Patterns Have Names

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

Naming Patterns Improves Communication

When a pattern is named, we can discuss with others a complex principle or design idea with a simple name. Consider the following discussion between two software designers, using a common vocabulary of patterns *(Creator, Factory,* and so on) to decide upon a design:

Fred: "Where do you think we should place the responsibility for creating a *SalesLineltem?* I think a *Factory."*
Wilma: "By *Creator,* I think *Sale* will be suitable."
Fred: "Oh, right.I agree."
Chunking design idioms and principles with commonly understood names facilitates communication and raises the level of inquiry to a higher degree of abstraction.

GRASP: Patterns of General Principles in Assigning Responsibilities

To summarize the preceding introduction:

- The skillful assignment of responsibilities is extremely important in object design.
- . Determining the assignment of responsibilities often occurs during the creation of interaction diagrams, and certainly during programming.

Patterns are named problem/solution pairs that codify good advice and principles often related to the assignment of responsibilities Understanding and being able to apply these principles during the creation of interaction diagrams is important because a software developer new to object technology needs to master these basic principles as quickly as possible; they form the foundation of how a system will be designed.

GRASP is an acronym that stands for General Responsibility Assignment Software Patterns.2 The name was chosen to suggest the importance of *grasp ing* these principles to successfully design object-oriented software.

How to Apply the GRASP Patterns

The following sections present the first five GRASP patterns:

- Information Expert
- Creator
- High Cohesion
- Low Coupling
- Controller

It is worthwhile mastering these five first because they address very basic, common questions and fundamental design issues. Please study the following patterns, note how they are used in the example interaction diagrams, and then apply them during the creation of new interaction diagrams. Start by *mastering Information Expert, Creator, Controller, High Cohesion,* and *Low Coupling.* Later, learn the remaining patterns.

Design Model: Use case realizations with GRASP patterns

Use-Case Realizations

To quote, "A use-case realization describes how a particular use case is realized within the design model, in terms of collaborating objects". More precisely, a designer can describe the design of one or more *scenarios* of a use case; each of these is called a use-case realization. Use-case realization is a UP term or concept used to remind us of the connection between the requirements expressed as use cases, and the object design that satisfies the requirements.

UML interaction diagrams are a common language to illustrate use-case realizations. There are principles and patterns of object design, such as Information Expert and Low Coupling that can be applied during this design work. To review, Figure illustrates the relationship between some UP artifacts:

- The use case suggests the system events that are explicitly shown in system sequence diagrams.
- Details of the effect of the system events in terms of changes to domain objects may optionally be described in system operation contracts.
- The system events represent messages that initiate interaction diagrams, which illustrate how objects interact to fulfill the required tasks—the use case realization.
- The interaction diagrams involve message interaction between software objects whose names are sometimes inspired by the names of conceptual classes in the Domain Model, plus other classes of objects.

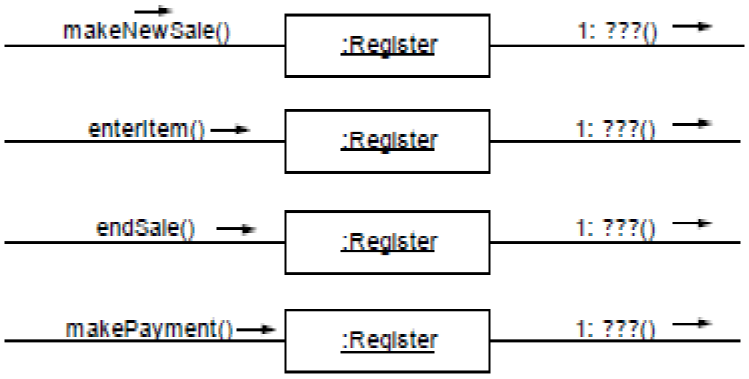
Artifact Comments

Interaction Diagrams and Use-Case Realizations

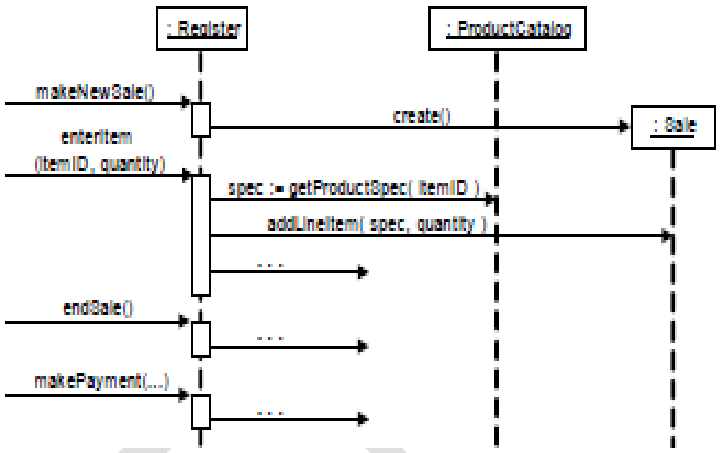
In the current iteration we are considering various scenarios and system events such as:

• *Process Scale: makeNewSale, enterItem, endSale, makePayment*

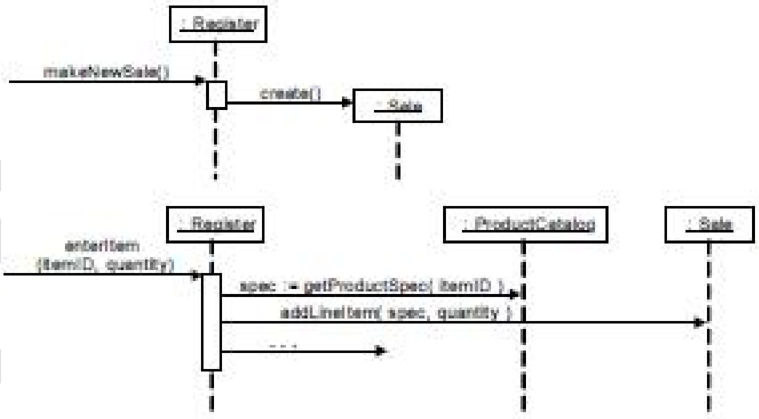
If collaboration diagrams are used to illustrate the use-case realizations, a different collaboration diagram will be required to show the handling of each system event message.



On the other hand, if sequence diagrams are used, it *may* be possible to fit all system event messages on the same diagram.



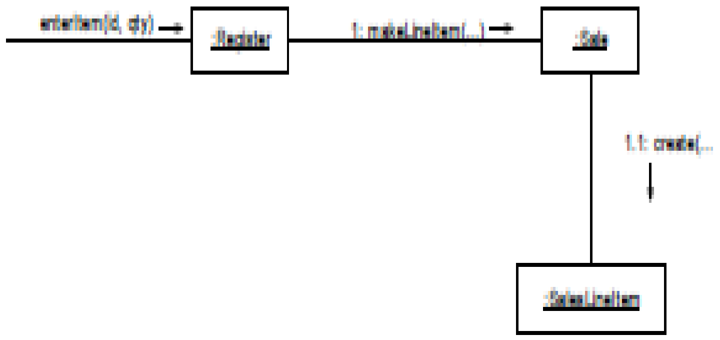
However, it is often the case that the sequence diagram is then too complex or long. It is legal, as with interaction diagrams, to use a sequence diagram for each system event message, as in Figure.



Contracts and Use-Case Realizations

To reiterate, it may be possible to design use-case realizations directly from the use case text. In addition, for some system operations, contracts may have been written that add greater detail or specificity.

In conjunction with contemplating the use case text, for each contract, we work through the postcondition state changes and design message interactions to satisfy the requirements. For example, given this partial *enterItem* system operation, a partial interaction diagram is shown in Figure that satisfies the state change of *SalesLineItem* instance creation.



Caution: The Requirements Are Not Perfect

It is useful to bear in mind that previously written use cases and contracts are only a guess of what must be achieved. The history of software development is one of invariably discovering that the requirements are not perfect, or have changed. This is not an excuse to ignore trying to do a good requirements job, but a recognition of the need to continuously engage customers and subject matter experts in review and feedback on the growing system's behavior.

An advantage of iterative development is that it naturally supports the discovery of new analysis and design results during design and implementation work. The spirit of iterative development is to capture a "reasonable" degree of information during requirements analysis, filling in details during design and implementation.

The Domain Model and Use-Case Realizations

Some of the software objects that interact via messages in the interaction diagrams are inspired from the Domain Model, such as a *Sale* conceptual class and *Sale* design class. The choice of appropriate responsibility placement using the GRASP patterns relies, in part, upon information in the Domain Model. As mentioned, the existing Domain Model is not likely to be perfect; errors and omissions are to be expected. You will discover new concepts that were previously missed, ignore concepts that were previously identified, and do likewise with associations and attributes.

Design Class diagrams in each MVC layer

With the completion of interaction diagrams for use-case realizations for the current iteration of the NextGen POS application, it is possible to identify the specification for the software classes (and interfaces) that participate in the software solution, and annotate them with design details, such as methods.

When to Create DCDs

Although this presentation of DCDs *follows* the creation of interaction diagrams, in practice they are usually created in parallel. Many classes, method names and relationships may be sketched out very early in design by applying responsibility assignment patterns, prior to the drawing of interaction diagrams.

It is possible and desirable to do a little interaction diagramming, then update the DCDs, then extend the interaction diagrams some more, and so on. These class diagrams may be used as an alternative, more graphical notation over CRC cards in order to record responsibilities and collaborators.

Example DCD

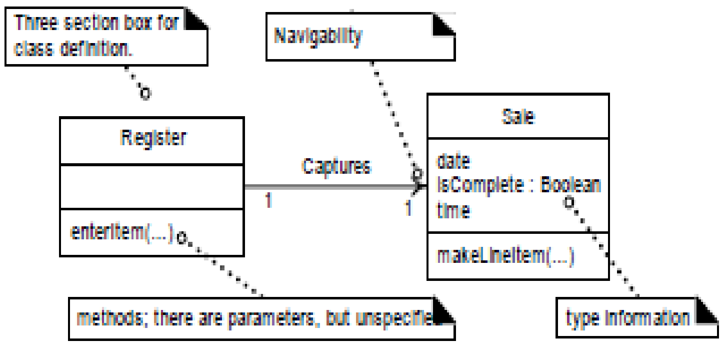
The DCD in Figure illustrates a partial software definition of the *Register* and *Sale* classes. In addition to basic associations and attributes, the diagram is extended to illustrate, for example, the methods of each class, attribute type information, and attribute visibility and navigation between objects.

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

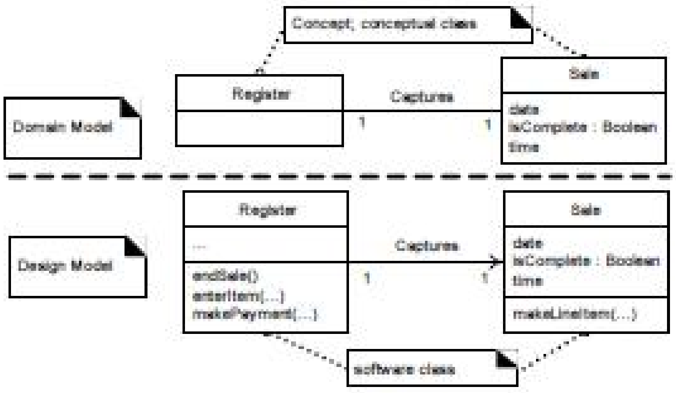
In contrast to conceptual classes in the Domain Model, design classes in the DCDs show definitions for software classes rather than real-world concepts.



The UP does not specifically define an artifact called a "design class diagram." The UP defines the Design Model, which contains several diagram types, including interaction, package, and class diagrams. The class diagrams in the UP Design Model contain "design classes" in UP terms. Hence, it is common to speak of "design class diagrams," that is shorter than, and implies, "class diagrams in the Design Model."

Domain Model vs. Design Model Classes

To reiterate, 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. In these diagrams, a *Sale* represents a software class.



Mapping Design to Code

Programming and the Development Process

The prior design work should not be taken to imply that there is no prototyping or design while programming; modern development tools provide an excellent environment to quickly explore alternate approaches, and some (or even lots) design-while-programming is usually worthwhile. However, some developers find that a little forethought with visual modeling before programming is helpful, especially those who are comfortable with visual thinking or diagrammatic languages.

The creation of code in an object-oriented programming language—such as Java or C#—is not part of OOA/D; it is an end goal. The artifacts created in the UP Design Model provide some of the information necessary to generate the code. A strength of OOA/D and OO programming—when used with the UP—is that they provide an end-to-end roadmap from requirements through to code. The various artifacts feed into later artifacts in a traceable and useful manner, ultimately culminating in a running application. This is not to suggest that the road will be smooth, or can simply be mechanically followed—there are too many variables. But having a roadmap provides a starting point for experimentation and discussion.

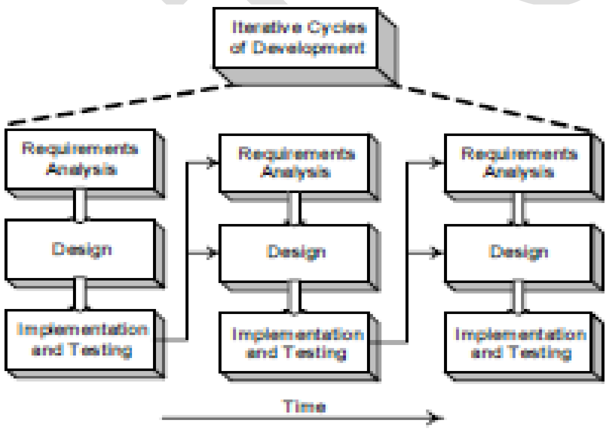
Creativity and Change During Implementation

Some decision-making and creative work was accomplished during design work. It will be seen during the following discussion that the generation of the code— in this example—is a relatively mechanical translation process. However, in general, the programming work is not a trivial code generation step—quite the opposite. Realistically, the results generated during design are an incomplete first step; during programming and testing, myriad changes will be made and detailed problems will be uncovered and resolved.

Done well, the design artifacts will provide a resilient core that scales up with elegance and robustness to meet the new problems encountered during programming. Consequently, expect and plan for change and deviation from the design during programming.

Code Changes and the Iterative Process

A strength of an iterative and incremental development process is that the results of a prior iteration can feed into the beginning of the next iteration. Thus, subsequent analysis and design results are continually being refined and enhanced from prior implementation work. For example, when the code in iteration N deviates from the design of iteration N (which it inevitably will), the final design based on the implementation can be input to the analysis and design models of iteration N+l.


Iterative Cycles of Development
Requirements Analysis
Design
Implementation and Testing
Requirements Analysis
Design
Implementation and Testing
Requirements Analysis
Design
Implementation and Testing
Time

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

Code Changes, CASE Tools, and Reverse-Engineering

It is desirable for the diagrams generated during design to be semi-automati-cally updated to reflect changes in the subsequent coding work. Ideally this should be done with a CASE tool that can read source code and automatically generate, for example, package, class, and sequence diagrams. This is an aspect of **reverse-engineering**—the activity of generating diagrams from source (or sometimes, executable) code.

Mapping Designs to Code

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

- class and interface definitions
- method definitions

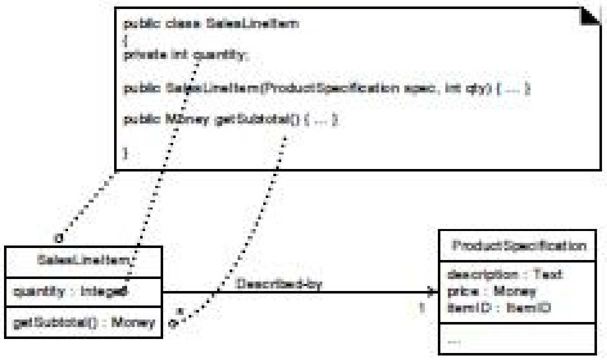
The following sections discuss their generation in Java (as a typical case).

Creating Class Definitions from DCDs

At the very least, DCDs depict the class or interface name, superclasses, method signatures, and simple attributes of a class. This is sufficient to create a basic class definition in an object-oriented programming language. Later discussion will explore the addition of interface and namespace (or package) information, among other details.

Defining a Class with Methods and Simple Attributes

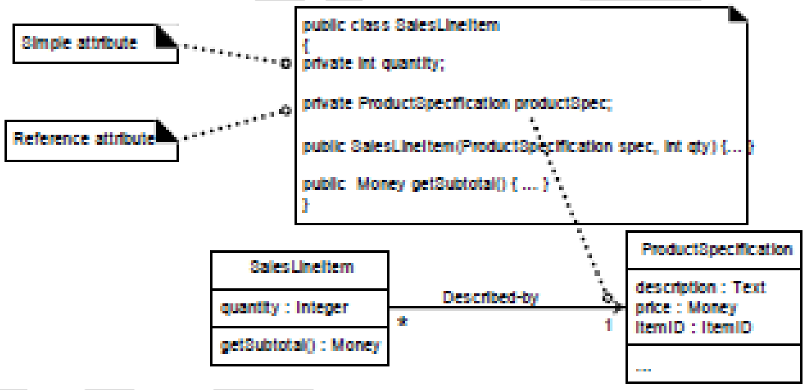
From the DCD, a mapping to the basic attribute definitions (simple Java instance fields) and method signatures for the Java definition of *SalesLineItem* is straightforward, as shown in Figure.



```
public class SalesLineItem
{
private int quantity;

public SalesLineItem(ProductSpecification spec, int qty) { ... }

public Money getSubtotal() { ... }

}
```

Adding Reference Attributes

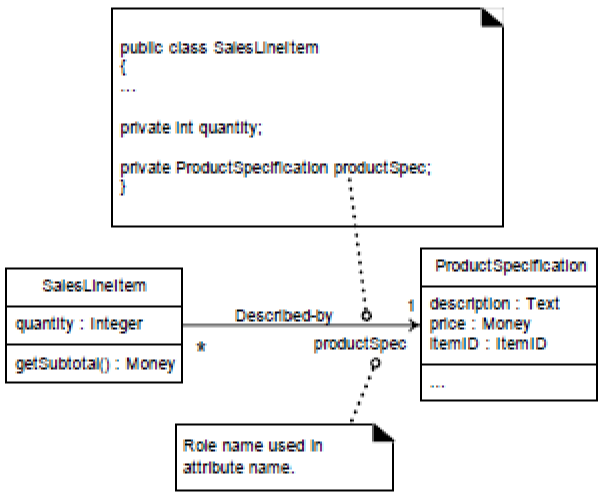
A reference attribute is an attribute that refers to another complex object, not to a primitive type such as a String, Number, and so on. For example, a *SalesLineItem* has an association to a *ProductSpecification*, with navigability to it. It is common to interpret this as a reference attribute in class *SalesLineItem* that refers to a *ProductSpecification* instance. In Java, this means that an instance field referring to a *ProductSpecification* instance is suggested.



```
public class SalesLineItem
{
private int quantity;

private ProductSpecification productSpec;

public SalesLineItem(ProductSpecification spec, int qty) {... }

public  Money getSubtotal() { ... }
}
```

Reference Attributes and Role Names

The next iteration will explore the concept of role names in static structure diagrams. Each end of an association is called a role. Briefly, a **role name** is a name that identifies the role and often provides some semantic context as to the nature of the role. If a role name is present in a class diagram, use it as the basis for the name of the reference attribute during code generation, as shown in Figure.

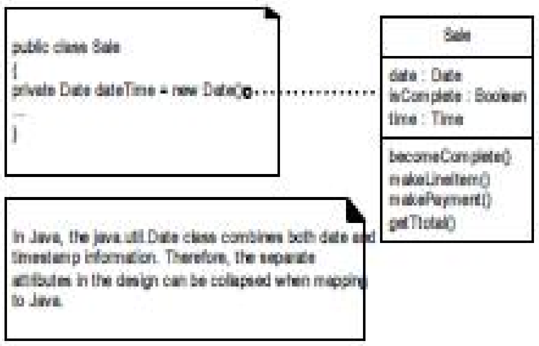


```
public class SalesLineItem
{
...

private int quantity;

private ProductSpecification productSpec;
}
```

Role name used in attribute name.

Mapping Attributes

The *Sale* class illustrates that in some cases one must consider the mapping of attributes from the design to the code in different languages. Figure illustrates the problem and its resolution.

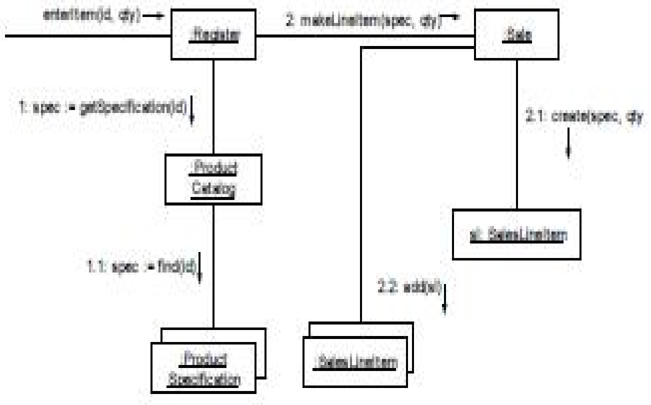


```
public class Sale
{
private Date dateTime = new Date();
...
}
```

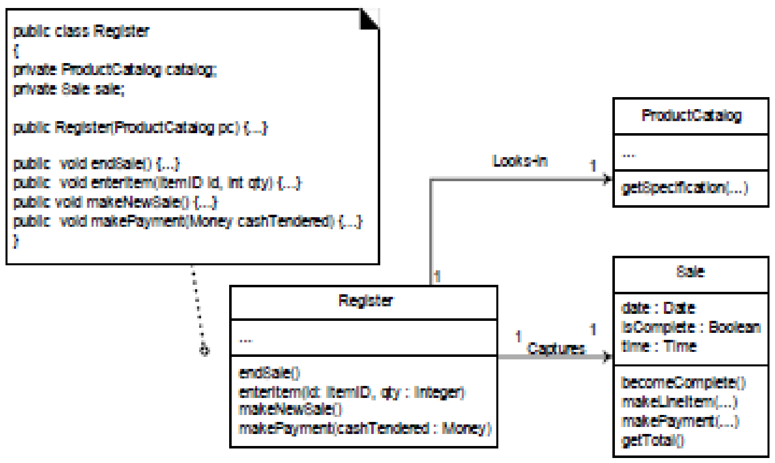
In Java, the java.util.Date class combines both date and timestamp information. Therefore, the separate attributes in the design can be collapsed when mapping to Java.

Creating Methods from Interaction Diagrams

An interaction diagram shows the messages that are sent in response to a method invocation. The sequence of these messages translates to a series of statements in the method definition. The *enterltem* interaction diagram in Figure illustrates the Java definition of the *enterltem* method. In this example, the *Register* class will be used. A Java definition is shown in Figure.



The Register-enterItem Method



```
public class Register
{
private ProductCatalog catalog;
private Sale sale;

public Register(ProductCatalog pc) {...}

public  void endSale() {...}
public  void enterItem(ItemID id, int qty) {...}
public void makeNewSale() {...}
public  void makePayment(Money cashTendered) {...}
}
```

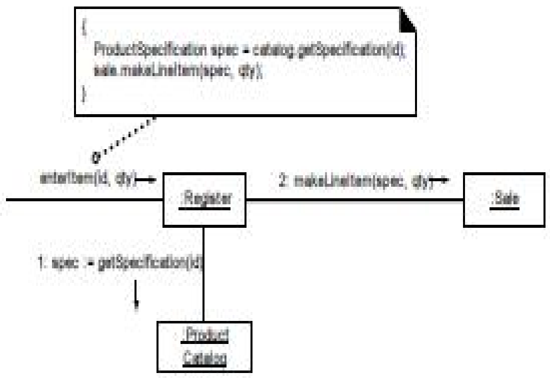
The *enterltem* message is sent to a *Register* instance; therefore, the *enterltem* method is defined in class *Register.* public void enterltem (ItemID itemID, int qty)

Message 1: *A getSpecification* message is sent to the *ProductCatalog* to retrieve a *ProductSpecification.*

ProductSpecif ication spec = catalog. getSpecif ication(itemID);

Message 2: The *makeLineltem* message is sent to the *Sale.* sale .makeLineltemf spec, qty);

In summary, each sequenced message within a method, as shown on the interaction diagram, is mapped to a statement in the Java method. The complete *enterltem* method and its relationship to the interaction diagram is shown in Figure.



```
{
  ProductSpecification spec = catalog.getSpecification(id);
  sale.makeLineItem(spec, qty);
}
```

Container/Collection Classes in Code

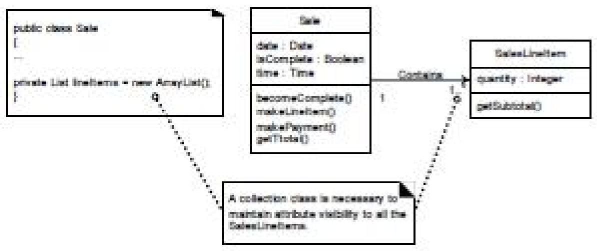
It is often necessary for an object to maintain visibility to a group of other objects; the need for this is usually evident from the multiplicity value in a class diagram—it may be greater than one. For example, a *Sale* must maintain visibility to a group of *SalesLineltem* instances, as shown in Figure.

In OO programming languages, these relationships are often implemented with the introduction of a intermediate container or collection. The one-side class defines a reference attribute pointing to a container/collection instance, which contains instances of the many-side class.

For example, the Java libraries contain collection classes such as *ArrayList* and *HashMap,* which implement the *List* and *Map* interfaces, respectively. Using *ArrayList,* the *Sale* class can define an attribute that maintains an ordered list *of SalesLineltem* instances. The choice of collection class is of course influenced by the requirements; key-based lookup requires the use of a *Map,* a growing ordered list requires a *List,* and so on.

Exceptions and Error Handling

Exception handling has been ignored so far in the development of a solution. This was intentional to focus on the basic questions of responsibility assignment and object design. However, in application development, it is wise to consider exception handling during design work, and certainly during implementation.



<u>Design class diagrams for case study and skeleton code</u>

Identify Software Classes and Illustrate Them

The first step in the creation of DCDs as part of the solution model is to identify those classes that participate in the software solution. These can be found by scanning all the interaction diagrams and listing the classes mentioned. For the POS application, these are:

- Register
- ProductCatalog
- Store Payment
- Sale
- ProductSpecification
- SalesLineItem

The next step is to draw a class diagram for these classes and include the attributes previously identified in the Domain Model that are also used in the design. Note that some of the concepts in the Domain Model, such as *Cashier,* are not present in the design. There is no need—f or the current iteration—to represent them in software. However, in later iterations, as new requirements and use cases are tackled, they may enter into the design. For example, when security and log-in requirements are implemented, it is likely that a software class named *Cashier* will be relevant.

Add Method Names

The methods of each class can be identified by analyzing the interaction diagrams. For example, if the message *makeLineItem* is sent to an instance of class *Sale,* then class *Sale* must define a *makeLineItem* method. In general, the set of all messages sent to a class X across all interaction diagrams indicates the majority of methods that class X must define. Inspection of all the interaction diagrams for the POS application yields the allocation of methods shown in Figure.

Method Name Issues

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

Method Names—create

The *create* message is a possible UML language independent form to indicate instantiation and initialization. When translating the design to an object-oriented programming language, it must be expressed in terms of its idioms for instantiation and initialization. There is no actual *create* method in C++, Java, or Smalltalk. For example, in C++, it implies automatic allocation, or free store allocation with the *new* operator, followed by a constructor call. In Java, it implies the invocation of the *new* operator, followed by a constructor call. Because of its multiple interpretations, and also because initialization is a very common activity, it is common to omit creation-related methods and constructors from a DCD.

Method Names—Accessing Methods

Accessing methods retrieve (accessor method) or set (mutator method) attributes. In some languages (such as Java) it is a common idiom to have an accessor and mutator for each attribute, and to declare all attributes private (to enforce data encapsulation). These methods are usually excluded from depiction in the class diagram because of the high noise-to-value ratio they generate; for n attributes, there are 2n uninteresting methods. For example, the *Product-Specification's getPrice* (or *price)* method is not shown, although present, because *getPrice* is a simple accessor method.

Method Names—Multiobjects

A message to a multiobject is interpreted as a message to the container/collection object itself. For example, the following *find* message to the multiobject is meant be interpreted as a message to the container/collection object, such as to a Java *Map*, a C++ *map* or a Smalltalk *Dictionary*. These container/collection interfaces or classes (such as the interface j*ava.util.Map)* are usually predefined library elements, and it is not useful to show these classes explicitly in the DCD, since they add noise, but little new information.

Method Names—Language-Dependent Syntax

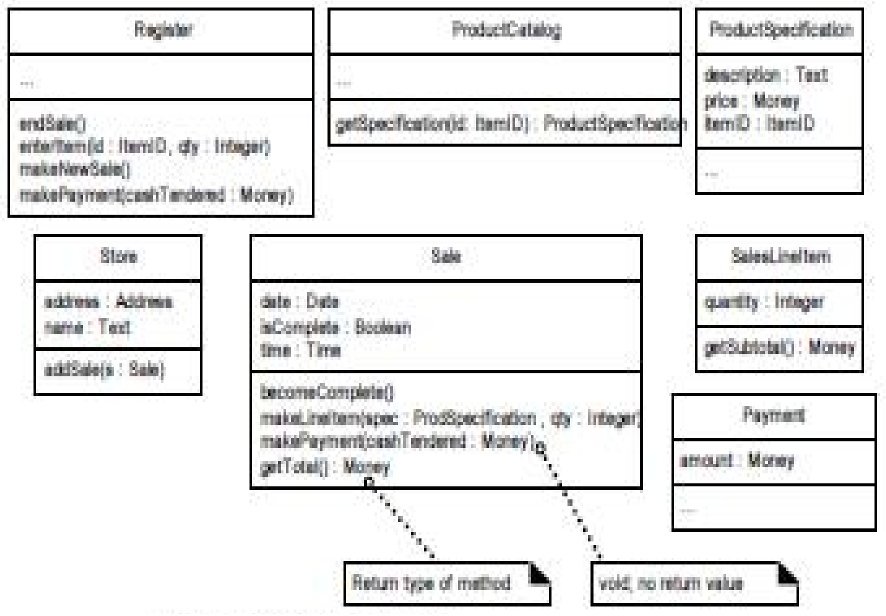
Some languages, such as Smalltalk, have a syntax that is very different from the basic UML format of *methodName(parameterList).* It is recommended that the basic UML format be used, even if the planned implementation language uses a different syntax. The translation should ideally take place during code generation time, instead of during the creation of the class diagrams. However, the UML does allow other syntax for method specification.

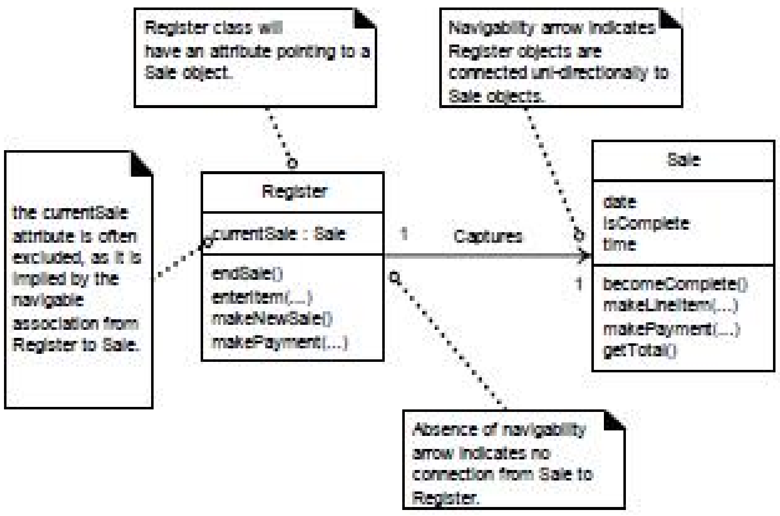
Adding More Type Information

The types of the attributes, method parameters, and method return values may all optionally be shown. The question as to whether to show this information or not should be considered in the following context:
For example, is it necessary to show all the parameters and their type information?
It depends on how obvious the information is to the intended audience.

Adding 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.




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.

Class Payment

```
public class Payment {
private Money amount;
public Payment( Money cashTendered ){ amount = cashTendered; }
public Money getAmount() { return amount; } }
```

Class ProductCatalog

```
public class ProductCatalog {
private Map productSpecifications = new HashMap();
public ProductCatalog() {
// sample data
ItemID idl = new ItemID( 100 );
ItemID id2 = new ItemID( 200 );
Money price = new Money( 3 );
ProductSpecification ps;
ps = new ProductSpecification( idl, price, "product 1" );
productSpecifications.put( idl, ps );
ps = new ProductSpecification( id2, price, "product 2" );
ProductSpecifications.put( id2, ps ); }
public ProductSpecification getSpecification( ItemID id ) {
return (ProductSpecification)productSpecifications.get( id );
}
}
```

Class Register

```
public class Register {
private ProductCatalog catalog;
private Sale sale;
public Register( ProductCatalog catalog ) {
this.catalog = catalog; }
public void endSaleO {
sale.becomeComplete();
}
public void enterltem( ItemID id, int quantity ) {
ProductSpecification spec = catalog.getSpecification( id );
sale.makeLineItem( spec, quantity ); }
public void makeNewSale() {
sale = new Sale(); }
```

```
public void makePayment( Money cashTendered ) {
sale.makePayment( cashTendered ); }
```

Class ProductSpecification

```
public class ProductSpecification {
private ItemID id;
private Money price;
private String description;
public ProductSpecification
( ItemID id. Money price. String description ) {
this.id = id;
this.price = price;
this.description = description; }
public ItemID getltemlDO { return id;}
public Money getPrice() { return price; }
public String getDescription() { return description; }
}
```

Class Sale

```
public class Sale
{
private List lineltems = new ArrayListO;
private Date date = new Date();
private boolean isComplete = false;
private Payment payment;
public Money getBalanceO {
return payment.getAmount().minus( getTotal() ); }
public void becomeComplete() { isComplete = true; }
public boolean isComplete() { return isComplete; }
public void makeLineltem
( ProductSpecification spec, int quantity ) {
lineltems.add( new SalesLineltem( spec, quantity ) ); }
public Money getTotal()
{
Money total = new MoneyO;
Iterator i = lineltems.iterator( ) ;
while ( i.hasNextO )
{
SalesLineltem sli = (SalesLineltem) i.nextO;
total.add( sli.getSubtotal() );
}
return total; }
```

***