Modeling Diagrams for OO Programs (Chapter 14)

Questions:
1. Let’s say that your company, Hotorola, has decided to launch a $100 Million Dollar project to develop a new line of cell phones with brand new state-of-the-art features.

It is estimated that the software development part of the new cell phone project will cost $80 Millions with roughly the following breakdown:

- Selection, testing, and fine-tuning of the embedded software: $10 Million
- Networking software for the telephony aspects of the cell phone and for the upgrade of the base station computer software to handle the new phone features: $30 Million
- Software simulation tools for testing the cell phone at the device level and as a part of a network: $20 Million
- Accounting and billing software for the new line of phones: $10 Million
- Parts inventory management software: $5 Million
The plan is to use Java for the embedded software and C++ for the rest.

The software development will be carried out partly in house and partly through out-sourcing to other corporations that are particularly strong in certain specialities such as testing/simulation software, billing/accounting software, etc.

a team in Chicago

a team in Munich, Germany

a team in Bangalore, India

Assume that you are overall in-charge of software development for this project.

What do you think is your biggest challenge?
2. What are the different aspects of OO design?
3. What are the commonly used diagrams from UML?
4. What is a use case?
5. What aspect of an OO program is represented by a use-case diagram?
6. What relationships can be shown in a use-case diagram and what do these relationships mean?
Use cases and use case diagrams must at some point be translated into classes for eventual implementation.

In UML, a class is represented by a rectangular box which in its most detailed representation is divided into three parts vertically.
In its most common usage, a class diagram shows two relationships between different classes:

**generalization**

and

**association.**
Other types of relationships between classes that can be depicted in a class diagram are *aggregation* and *composition*. 
In the OO literature, one also commonly sees mention of IsA and HasA relationships between classes.

IsA represents a generalization-specialization sort of a relationship and the HasA an association, an aggregation, or a composition.
The name IsA is supposed to capture relationships such as

A Manager IsAn Employee
A CorporateCustomer IsA Customer

In each such statement, what comes after IsA is a generalization or a super-type of what comes before.
On the other hand, a statement like

An Order HasA Customer
An Orchestra HasA Player
A Window HasA Slider

expresses a containment, in the form of an association, an aggregation, or a composition.
A class diagram may be drawn using three different perspectives:

(i) *conceptual*

(ii) *specification*

(iii) *implementation*
At the conceptual level, for each class you include only the bare minimum information needed to get an overall sense of the main concepts of a problem domain.

This will most frequently be the diagram you would draw when you are just getting started with the design of an OO program.

However, even after you have fully developed an OO system, a conceptual level diagram can be useful for communicating to others a coarse-level description of the system.

At the specification level, you want to show the interfaces of each class. At this level you’d want to make explicit the class responsibilities, as embodied in the public operations for each class.

At the implementation level, you want to show more precisely how a class was (or needs to be) implemented in code. Now you’d include the private and the protected attributes and operations as well.
Association as a Relationship Between Classes

An example of a more elaborate representation of an association:

An \textbf{Employee} has a data member called \texttt{employedBy} of type \texttt{Corporation}; this data member is shown as a label at the head of the arrowed association link from \texttt{Employee} to \texttt{Corporation}.

We can talk about the label \texttt{employedBy} as the \textit{role} played by a \texttt{Corporation} in an object of type \texttt{Employee}. 
The arrowhead on the association link from Employee to Corporation is referred to as the navigability arrow.

The arrow tells us as to which of the two objects implements the association.

In the example shown, the association with the rolename employedBy is implemented in the Employee class and therefore “belongs” to objects of type Employee.

The label ‘0..1’ at the Corporation end of the association is referred to as the multiplicity of the association, which specifies how many objects of type Corporation in role employedBy may associate with a single object of type Employee.

The multiplicity of ‘0..1’ means that an Employee is employed by no more than one Corporation.
The two association links on the previous slide can also be shown as a single line between the two classes.

If we were to do so for our example, the line would show navigability arrows, rolenames, and multiplicity symbols at both ends.

An association with no navigability arrows is considered bidirectional.

The association that was shown above is a binary association connecting two different classes.
A binary association is also allowed to connect the same class to itself.

Such an association link may connect two different objects from the same class, or one object to itself.

In the latter case, the association would be called reflexive.
This association would represent an Employee being allowed to supervise between 1 and 10 other employees.
Aggregation and Composition as Relationships Between Classes

The objects participating in an association will often have independent, and, in some sense, equal existences of their own.

But that is not true of all interclass relationships.

In other relationships, especially those that relate a “whole” to its “parts,” there can be lifetime dependencies between the whole and its parts.

When the whole ceases to exist, the parts may get destroyed at the same time. A composition represents such a tight linkage between a whole and its parts.
When you have a whole–part relationship in which the parts can have lifetimes independent of the whole, you have an aggregation.
Representing Attributes

The class data members — known as attributes in UML — are shown in a separate partition below the classname partition of the class box.

The UML convention for displaying an attribute is:

\[
\text{visibility} \quad \text{name [N]} : \text{type} = \text{initialValue} \quad \{\text{property-string}\}
\]

where the visibility is one of

+ for public visibility

# for protected visibility

- for private visibility
The keywords *public*, *protected*, and *private* can also be used for denoting visibility.

The absence of a visibility marker indicates that the visibility is not shown (not that it is undefined or public).
The name of the attribute is the string `name`.

The symbol `N` inside square brackets denotes the multiplicity allowed for the attribute.

A language-dependent specification of the implementation type of the attribute is denoted by `type`.

The string `initialValue` is a language-dependent expression for the default value of the attribute for a newly created instance of the class.

`property-string` a string for expressing those traits of the attribute that are not captured by the rest of the syntax.

For example, for an attribute that is read-only (such as a `const` in C++ or a `final` in Java), the `property-string` would be set to `frozen`.
visibility name [N] : type = initialValue {property-string}

---------------

The convention for expressing multiplicity is the same as for an association.

For example, if an attribute is allowed to take two or more values, the multiplicity symbol N would be replaced by ‘2..*’.

The absence of multiplicity designation means that exactly one value is allowed for the attribute.

The underscore, shown under name and type, if used, signifies that the attribute has class scope, which means the same thing that it is static or one per-class, as opposed to one per object. Except for the name, all other elements of the syntax specification are optional.
Employee

address
age
name

getName
setAddress
Representing Operations

The third partition from the top, when it exists, of a class box shows its operations, meaning the member functions of the class.

When a class is drawn at the specification level, only the public operations of the class are displayed.

However, at the implementation level, you’d also want to show the private and the protected operations.

The full UML syntax for an operation is

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

`visibility` and `name` mean the same as for the case of attributes.
The parameter-list is a comma-separated list of formal parameters, each specified using the syntax

\[
\text{kind name : type = defaultValue}
\]

where kind can be \textit{in}, \textit{out}, or \textit{inout}, where \textit{in} is for a parameter that passes a value to the operation, \textit{out} for a parameter that fetches a value from the operation, and \textit{inout} for a parameter that can play both roles. The symbols name, type, and defaultValue serve their usual roles.
Back to the syntax for an operation, the symbol **return-type** is an implementation dependent language type of the value returned by the operation.

The **property-string** can be used to express such traits as to whether an operation is abstract, which is the case when only the header is defined for the class and no implementation code is provided.

Finally, operations that have class scope are underlined as shown above.

It is useful to make a distinction between two types of operations: **query** and **modifier**.

A query operation simply tries to get the value of some class attribute without changing the state of the object. On the other hand, a modifier operation will change the state of the object.
Stereotypes

UML also allows a **stereotype** to be specified for a class just above the class name. The stereotype indicates what ‘kind’ of a class it is. The stereotype is enclosed in guillemots or the pair ‘<< >>,’ as in Figure ??.

```
 « financial instruments »

 **MarginAccount**

brokerLoan: double
maintenanceMargin: double
margin: double
marketValueOfSecurities: double

buyOnMargin(): void
checkSecurityMarginable(): bool
estimateRisk(): string
```
An *abstract class* is represented in the same way as a regular class, except that the name of the class is in bold italics.

If it is necessary to show specific objects pictorially, the notation used is the same as for a class, except that now the object name is followed by the class name after a colon, the whole construct underlined.

![Diagram of Joe Shmoe as a Person]

```
joeShmoe : Person

name: "Joe Shmoe"
address: "main street"
```
Interaction Diagram

An interaction diagram shows how objects collaborate in achieving a use case or a small set of related use cases.

Interaction diagrams are drawn at an early stage in a design process as they shed further light on the role of each class and give a more concrete focus to the responsibilities of each class.

Since in an interaction diagram you make explicit how a class interacts with other classes, you get a better sense of what methods to endow a class with.
There are two types of interaction diagrams:

*sequence diagrams*

and

*collaboration diagrams.*

Modern OO design software can switch automatically between the two.
Sequence Diagram

A sequence diagram shows in a time-sequenced manner the collaboration carried out by a group of objects to achieve a use case or a small set of related use cases.

Each object is assigned a lifeline that hangs below it in the diagram.

The time sequencing of the activities related to an object is made evident by placing the activities at different points on the lifeline.

Time increases downwards along a lifeline.
In a sequence diagram, an object interacts with other objects by either sending messages to them or by receiving messages from them.

A C++ or Java implementation of these interactions will most commonly consist of one object invoking a method on another object — the argument object — for either eliciting some behavior from the argument object, or for ascertaining the value or status of a data member of the argument object.
an Object:

message (synchronous)

message (asynchronous)

message (may be synchronous or asynchronous)

time

lifeline
The various components of a sequence diagram are:

1. Object icons

2. Lifelines

3. Arrowed lines for interaction messages between two different objects

4. Activation icons

5. Message to self lines
A message can be one of the following six different kinds:

1. Status message, like the value of a boolean variable, that an object transmits to another object.

2. Name of a method; this is the method that one object invokes on another object.

3. *[iteration basis] method*, where ‘*’ is the iteration marker; the named method is invoked on multiple instances of the target object; as to which instances specifically is controlled by the expression inside the square brackets.

4. *flag := method*, where flag is set to TRUE or FALSE depending on the outcome of the specified method that is invoked on the receiver of the message.

5. *[condition] method*, where the specified method is executed on the receiver object only when the given condition is satisfied.

6. The special symbol *new*, which means to create a new instance of the receiver object.
To illustrate the various components of a sequence diagram and the different types of messages in such diagrams, consider the following closely related set of use cases for the internet auction example:

The buyer first examines the max bid posted so far on all the items of interest to him/her. The buyer selects one item and posts his/her bid on that item. If the newly posted bid equals or exceeds the seller’s minimum acceptable bid, the seller is notified. If seller is satisfied with the bid, he/she so notifies the auction site and prepares a sales document for the buyer.
We can implement this set of use cases with the following objects:

A Buyer object

An AuctionList object, this is simply a list of all the items available for auction

An AuctionItem object (this object constantly compares the latest highest posted bid with the minimum highest bid acceptable to the seller)

A Seller object

A SalesDocument object
a Buyer:

AuctionList:

an AuctionItem:

a Seller:

*for all items in AuctionList*
getCurrentMaxBid()

selectItem()

postBid()

minAcceptBidExceeded := check()

notifySeller()

bidAcceptable = check()

new SalesDoc:

bidAcceptable := check()

[bidAcceptable]
The first message shown in this diagram is of the form

\*[ iteration-basis ] method

The presence of the iteration marker symbol ‘*’ means that the method is to be applied to multiple instances of the target object; as to which target objects exactly, that is controlled by the expression inside the brackets.

In this case, we wish to apply the method \texttt{getCurrentMaxBid()} to all the \texttt{AuctionItem} objects in the \texttt{AuctionList} object.

The second and the third messages from the top are straightforward invocations of the designated methods on the target objects.

In the fourth from the top message, of form

\texttt{flag := method}

we want to execute the method \texttt{check()} in order to set the boolean value of the flag \texttt{minAcceptBidExceeded}.
Going down the lifelines, the next message, of form

\[ \text{[condition] method} \]

is executed only after the boolean variable \text{minAcceptBidAccepted} is set to TRUE.

The next message seeks to set the value of the boolean variable \text{bidAcceptable} by running a method \text{check()} on self, in this case a \text{Seller} object.

The next two messages are merely notification messages.

Finally, for the last message, notice how when the \text{bidAcceptable} variable checks out to be TRUE, the \text{seller} object creates a new object of type \text{SalesDoc}.
In the previous diagram, the time order in which the various interactions take place is strict, in the sense that a given interaction between any two objects takes place after some other known interaction and before some other interaction, also known.

And even more importantly, each method invocation is synchronous, meaning that after an object invokes a method, it waits for the method to return.

We can also say that the method call *blocks* until the method has finished execution.
But many OO programs are written today using multi-processing and/or multithreading.

In such programs, when an object invokes a method in a separate process or a separate thread, it does not necessarily have to wait for the method to finish execution.

Such method invocation is called *asynchronous*.

We will now see how an interaction diagram can represent a use case that calls for asynchronous interactions.
Consider the following set of related use cases for a robot engaged in autonomous navigation using its cameras to identify directions to landmarks in its environment:

When desiring to navigate from its current location to a target location, the navigator module must first get a precise fix on its current location. This it must do by locating a certain minimum number of landmarks in its environment and then by triangulating its position from the directions to those landmarks.
The MobileRobot asks its Navigator module to get a fix on its current location, which in turn creates a number (in our figure, two, but it could be any number) of LandmarkLocator objects, each of which is responsible for locating a particular landmark from a list of landmarks known to the robot.

Since the computational difficulty associated with the extraction and identification of a particular landmark would vary widely from landmark to landmark, it would be best to spawn the landmark identification processes asynchronously.

That would also make it easy to add additional landmark identification processes if necessary.
The concurrent processes here correspond to the different `LandmarkSeeker` threads.

And by these threads running asynchronously is meant that they do not block the caller, in this case the method of the `Navigator` object that spawns the threads.
The half-arrowhead messages are asynchronous.

When such methods are invoked, the caller does not block.

Ordinarily, as mentioned before, when a method A invokes method B, A waits for B to return.

But when B is launched asynchronously, A does not wait and continues doing what comes next after having launched B.

The thin vertical rectangles below the object boxes are called activations or Focus of Control (FOC).

To explain this notation, notice the three activations below the box for the Navigator object.

The first activation says that the object stays in operation as it is spawning asynchronously the different LandmarkSeeker objects.

The second activation takes effect when the return ”have landmarks” is received from the first LandmarkSeeker object.
During this activation, the *Navigator* object also checks whether a certain minimum number of the other *LandmarkSeeker* objects have returned “have landmark.” The third activation does the same for the second *LandmarkSeeker* object.

The X below each activation for asynchronously created objects means that the thread is supposed to get destroyed after it has finished its task.
A collaboration diagram is another way of showing the interaction between objects.

In this diagram, you do away with the lifelines of the sequence diagrams.

Messages go directly from object icons to object icons.

The temporal sequencing of the messages is displayed by assigning a sequence number to each message.
1. *[for all AuctionItems in AuctionList]: getCurrentMaxBid()

2. select Item()

3. postBid()

4. minAcceptBidExceeded := check()

5. [minAcceptBidExceeded]: notifySeller()

6. bidAcceptable := check()

7. [bidAcceptable]: notifyAuctionItem()

8. [bidAcceptable]: notifyBuyer()
Note the syntax used for objects:

\[
\text{objectName} : \text{className}
\]

where either the object name or the class name may be omitted. But if you omit the object name, you must retain the colon to make it clear that you are using a class name.
Package Diagram

The word “package” in UML means the same thing as it does in Java.

If you have organized your classes into packages, you may want to draw a package diagram to show the dependencies between packages.

One package depends on another package if any class in the former depends on any class in the latter.

Such inter-package dependencies come into existence any time a class in one package uses a class in another package.
A package is denoted by a rectangular box with a “tab” at the left end of its top.

\[\text{-----}
|     |
\[---------------------
|     |
| Swing|
|     |
\[---------------------\]
Statechart Diagram

Statechart diagrams are best for displaying the different states that an object can get into across several use cases.

They help to give a better understanding of the lifetime behavior of a single object in an OO program.

The name of the state is shown as the topmost entry in bold in a rounded rectangle.

If there is additional information inside the rectangle, it is separated from the name by a line.

An entry preceded by “do/” denotes an activity in that state.
The paths with arrowheads show how the object transitions from state to state. The syntax of the labels for the transitions is

\[ \text{Event[Guard]/Action} \]

all three parts of which are optional.

What this syntax means is that the occurrence of the Event will cause this state transition provided Guard condition evaluates to TRUE.
However, before the actual transition to the new state takes place, Action must be executed.

Since at any time only one transition can be taken out of a state, the guards must be mutually exclusive.
Grouping all the states together that have a transition to the same target state.

Call the grouped states a **superstate**.

Have a single transition from the superstate to the target state, as shown below:
A Buyer object would also be involved in use cases that deal with a check of his/her past record as an auction site participant, credit-worthiness before the acceptance of the bid, etc.
In a modern implementation, it is likely that the computations represented by the two previous state diagrams would be executed in two separate threads.

In this way, the background check and credit authorization could proceed in parallel with the Buyer’s bid interaction with the rest of the auction server.

This would, of course, imply that a Buyer object may simultaneously be in two states, one from the former state diagram and one from the state diagram above.

If we want to show the combined behavior of a Buyer object that would correspond to these two separate but concurrently running state diagrams, we can combine the two diagrams into what’s called a concurrent state diagram:
Activity Diagrams

What a class diagram does for getting an overall view of the main concepts of a problem domain and how they are related, an activity diagram does for getting an overall view of the main functionalities of a problem domain how these functionalities are to be achieved.

Interaction diagrams also portray functionality, but mostly at the level of a single or a small number of related use cases. Activity diagrams present functionality at a larger level.

An important issue in portraying activity at a level that is global with respect to the overall functionality of a software system is the identification of those sub-activities that can be executed simultaneously, through either multi-processing or multithreading, and those that must be executed serially.
An activity diagram gives us a representational tool called the *synchronization bar* to highlight those sub-activities that can be executed simultaneously.

A synchronization bar is shown by a double line with activities flowing into it and flowing out of it.
The figure shows a synchronization bar with multiple incoming and outgoing activities and with a condition \([condition]\) attached.

The meaning of the synchronization bar with respect to the incoming and the outgoing activities is different.

All of multiple incoming activities must be executed successfully before control is allowed to proceed beyond the synchronization bar.

The successful completion of all the incoming activities is subject to the condition attached to the bar.

In that sense, a synchronization bar is *conjunctive* with respect to the incoming activities.

With regard to the outgoing activities, all that a synchronization bar tells is that they can be executed in parallel, independently of each other.
SessionTerminated

state diagram for Buyer’s interaction with auction site

state diagram for background check and credit authorization

RejectionNotification

Activity1

Activity2

Activity3

synchronization bar

[condition]

activities to follow
An activity is *disjunctive* with respect to multiple incoming triggers.

What that means that if a trigger is received from any one of the incoming paths, the target activity would be enabled.
An activity diagram makes a special provision for representing a purely \textit{decision} activity, which in most cases will consist of testing the value of a boolean variable. This type of activity is represented by a diamond.
Activity1

trigger

Activity2

Activity1 Activity2 .... ActivityN

SpecialActivity
When a buyer checks in, we must ascertain whether or not he/she is an already registered user of the system. If not, we ask the buyer to fill out a form and supply the information needed. The buyer proceeds to browse through the list of items that he/she is interested in. The buyer is given the option of either bidding on the items individually, or bidding on groups of items together (what if the buyer wants two works of art together or none, or four Louis IV chairs together or none). When the buyer begins to start selecting items to bid on, we want to start the process of running a credit check on the buyer. If the bids placed by the buyer and the credit check are okay, we want to proceed to payment authorization before instructing the seller(s) to ship the item(s) to the buyer.
Is FOO true?

- no
- yes
Start

Buyer checks in

on file?

yes

Ask Buyer to fill form

no

Browse AuctionList

Select items

A group of items or a single item?

single

Record bid

fork

group

Initiate credit check

fork

Q

P

Record bid for each item

[for each item in the group]

Authorize payment to seller

Items shipped to Buyer

[bids accepted on all items selected singly or in groups and credit check ok]