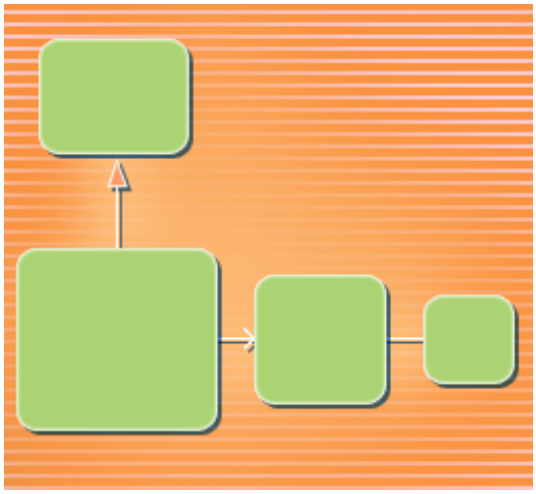
UML basics

Part III: The class diagram

by **Donald Bell**
IBM Global Services



In June 2003, I began a series of articles titled "UML Basics," designed as an introduction to the Unified Modeling Language. The first article in this series provided high-level introductions to the most widely used diagrams in the UML; the second article offered an in-depth look at the activity diagram.

In this third article, I will focus on the class diagram. Although almost every UML-knowledgeable person claims to understand this diagram, very few actually know the diagram's proper notation set and consequently do not know how to use the diagram. The discussion that follows should enable you to understand and draw a proper class diagram using the UML v1.4 notation set.

This article assumes you have a rudimentary understanding of object-oriented design. For those of you who need a little assistance with OO concepts, you might try the Sun brief tutorial about Object Oriented Programming at http://java.sun.com/docs/books/tutorial/java/concepts/. Reading the sections "What Is a Class?" and "What Is Inheritance?" should give you enough understanding to read this article. In addition, David Taylor's book, Object-oriented Technologies: A Manager's Guide, *offers an excellent, high-level explanation of object-oriented design without requiring an in-depth understanding of computer programming.*

[Editor's note: In Ben Lieberman's article "The art of modeling, Part II," also published in this issue (November 2003), Lieberman uses the word element *to refer to "the things that make up a model's contents." Donald Bell uses the word* entity *to refer to the same things. In editing their*

respective articles, I saw no reason to insist that both writers use the same word.]

The class diagram's purpose

The purpose of the class diagram is to show the static structure of the system being modeled. The diagram specifically shows the entities in the system -- and I literally mean *entities*, as in "discrete things," not to be confused with "database entities" -- along with each entity's internal structure and relationships with other entities in the system. Because class diagrams only model the static structure of a system, only types of entities are shown on a class diagram; specific instances are not shown. For example, a class diagram would show an Employee class, but would not show actual employee instances such as Donald Bell, Mike Perrow, or Jimmy Buffett.

Developers typically think of the class diagram as a diagram specifically meant for them, because they can use it to find out details about the system's coded classes or soon-to-be-coded classes, along with each class's attributes and methods.

Class diagrams are particularly useful for business modeling, too. Business analysts can use class diagrams to model a business's current assets and resources, such as account ledgers, products, or geographic hierarchy.

The notation

As mentioned earlier, a class diagram models the static structure of entities. But for the next section of this article, *The basics*, you can think of *entities* as the same thing as *classes*. That's an oversimplification, but for now it will make things easier to understand. When you get to the *Beyond the basics* section, you will be ready for the fuller, more complex explanation of entities.

The basics

Modeling the static structure of classes, the class diagram shows each class's internal structure along with the relationship that the class has to other classes.

The UML representation of a class -- a class diagram -- is a rectangle containing three compartments stacked vertically. The top compartment shows the class's name. The middle compartment lists the class's attributes. The bottom compartment lists the class's operations. Figure 1 offers a simple example of a class diagram.

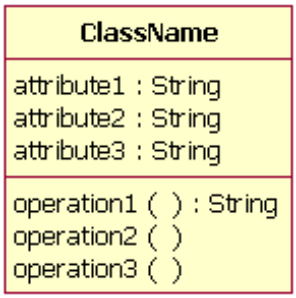


Figure 1: A generic class diagram showing a single class

Now let's look at a real-world example from the commercial airline industry. Figure 2 shows an airline flight modeled as a UML class on a class diagram. As we can see, the name is *Flight*, and in the middle compartment we see that the Flight class has three attributes: flightNumber, departureTime, and flightDuration. In the bottom section we see that the Flight class has two operations: delayFlight and getArrivalTime.

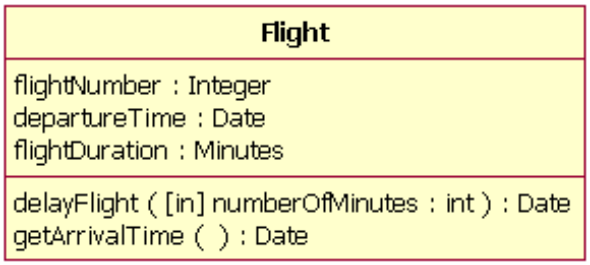


Figure 2: Class diagram of the airline class *Flight*

The class's attribute list

The attribute section of a class (the middle compartment) lists each of the class's attributes on a separate line. Each attribute line uses the following format:

name : attribute type

For example,

flightNumber : Integer

Continuing with our Flight class example, we can describe the class's attributes with attribute type information, as shown in Table 1.

Table 1: The Flight class's attribute names with their associated types

Attribute Name	Attribute Type
flightNumber	Integer
departureTime	Date
flightDuration	Minutes

In business class diagrams, the attribute types usually correspond to units that make sense to likely readers of the diagram (i.e., minutes, dollars, etc.). However, a class diagram that will be used to generate code needs classes whose attribute types are limited to the types provided by the programming language, or types included in the model that will also be implemented in the system.

Sometimes it is useful to show on a class diagram that a particular attribute has a default value. (In our Flight class example, an airline flight will rarely be under 60 minutes, because the "flight time" -- flightDuration -- includes time for the plane to prepare for departure, back out and proceed to the runway, take off, land, etc.) The UML specification allows for the identification of default values in the attribute list section by using the following notation:

name : attribute type = default value

For example:

flightDuration : Minutes = 60

Showing a default value for attributes is optional; Figure 3 shows a modified version of Figure 2, with the flightDuration attribute showing a default value of 60 minutes.

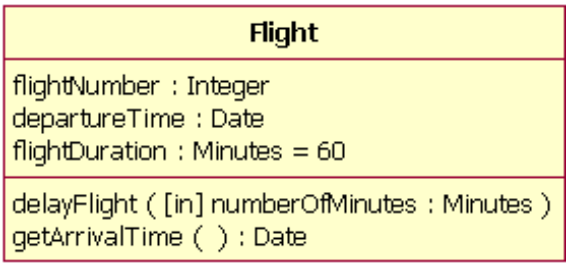


Figure 3: A Flight class diagram showing the flightDuration attribute's value defaulted to 60 minutes

The class's operations list

I mentioned earlier that the class's operations are documented in the third (lowest) compartment of the class diagram's rectangle. Like attributes, the operations of a class are displayed in a list format, with each operation on its own line. Operations are documented using the following notation:

name(parameter list) : type of value returned

The Flight class's operations are mapped in Table 2 below.

Table 2: Flight class's operations mapped from Figure 3

Operation Name	Parameters	Return Value Type
delayFlight	Name: numberOfMinutes; Type: Minutes	N/A
getArrivalTime	N/A	Date

Figure 3 shows that the delayFlight operation has one input parameter -- numberOfMinutes -- of the type Minutes. However, the delayFlight operation does not have a return value.[1] When an operation has parameters, they are put inside the operation's parentheses; each parameter uses the format "parameter name : parameter type", which can include an optional indicator to show whether or not the parameter is input to, or output of, the operation. This optional indicator appears as an "[in]" or "[out]" as seen in the delayFlight operation in Figure 3.

Figure 4 shows an example of a Flight class that has an operation called assignFlightCrew, and the assignFlightCrew operation takes two input parameters of pilot and coPilot.

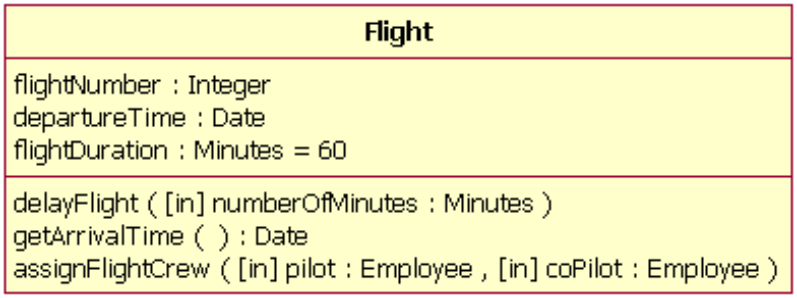


Figure 4: The Flight class has an operation called assignFlightCrew, which takes in two parameters.

Inheritance

A very important concept in object-oriented design, *inheritance*, refers to the ability of one class (child class) to *inherit* the identical functionality of another class (super class), and then add new functionality of its own. (Imagine that I inherited my mother's general musical abilities, but in my family I'm the only one who plays electric guitar.) To model inheritance on a class diagram, a solid line is drawn from the child class (the class inheriting the behavior) with a closed arrowhead (or triangle) pointing to the super class. Consider types of bank accounts: Figure 5 shows how both CheckingAccount and SavingsAccount classes inherit from the BankAccount class.

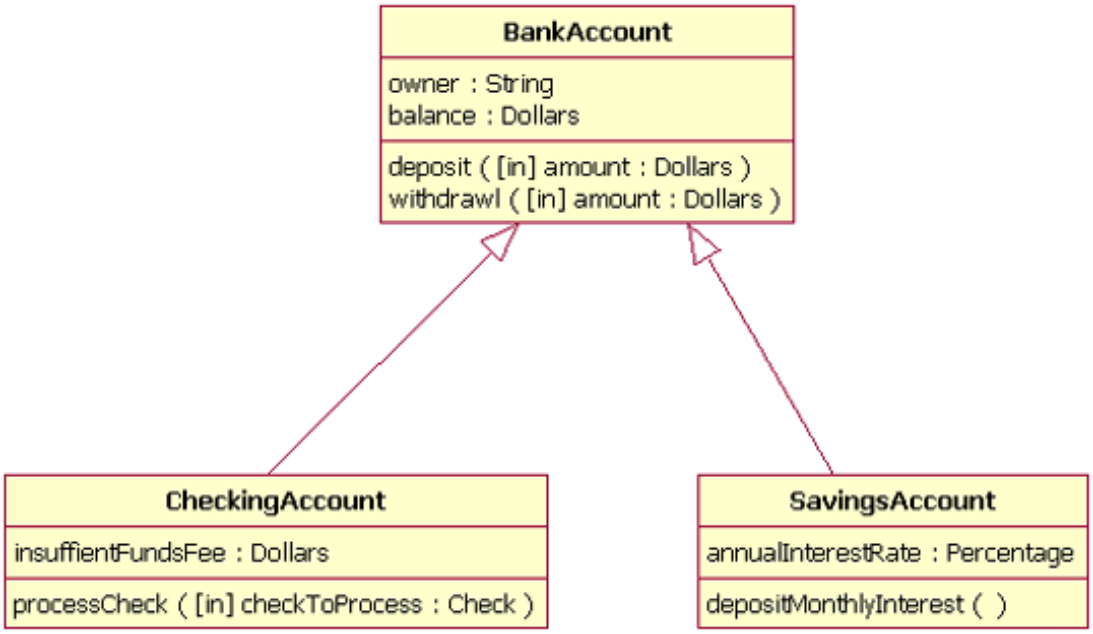


Figure 5: Inheritance is indicated by a solid line with a closed arrowhead pointing at the super class.

Associations

When you model a system, certain objects will be related to each other, and these relationships themselves need to be modeled for clarity. There are five types of associations. We will discuss bi-directional and uni-directional associations in this section and the remaining three association types in the *Beyond the basics* section. Please note that a detailed discussion of when to use each type of association is beyond the scope of this article. Instead, I will focus on the purpose of each association type and show how the association is drawn on a class diagram.

Bi-directional (standard) association

An association is a linkage between two classes. Associations are assumed to be bi-directional -- in other words, both classes are aware of their relationship and of the other class -- unless you qualify the association as some other type of association. Going back to our Flight example, Figure 6 shows a standard kind of association between the Flight class and the Plane class.

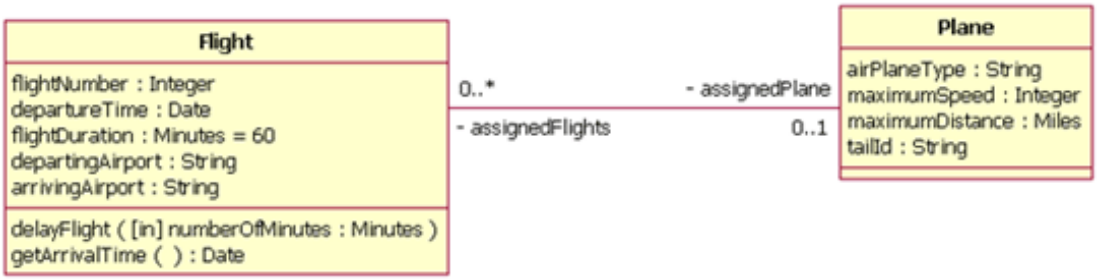


Figure 6: An example of a bi-directional association between a Flight class and a Plane class

A bi-directional association is indicated by a solid line between the two classes. At either end of the line, we place a role name and a multiplicity

value. Figure 6 shows that the Flight is associated with a specific Plane, and the Flight class knows about this association. The Plane takes on the role of "assignedPlane" in this association because the role name next to the Plane class says so. The multiplicity value next to the Plane class of 0..1 means that when an instance of a Flight exists, it can either have one instance of a Plane associated with it or no Planes associated with it (i.e., maybe a plane has not yet been assigned). Figure 6 also shows that a Plane knows about its association with the Flight class. In this association, the Flight takes on the role of "assignedFlights"; the diagram in Figure 6 tells us that the Plane instance can be associated either with no flights (e.g., it's a brand new plane) or with up to an infinite number of flights (e.g., the plane has been in commission for the last five years).

Uni-directional association

A uni-directional association shows that two classes are related, but only one class knows that the relationship exists. Figure 7 shows an example of an Overdrawn Accounts report with a uni-directional association.

Figure 7: An example of a uni-directional association: The OverdrawnAccountsReport class knows about the BankAccount class, but the BankAccount class does not know about the association.

A uni-directional association is drawn as a solid line with an open arrowhead (not the closed arrowhead, or triangle, used to indicate inheritance) pointing to the known class. Like standard associations, the uni-directional association includes a role name and a multiplicity value, but unlike the standard bi-directional association, the uni-directional association only contains the role name and multiplicity value for the known class. In our example in Figure 7, the OverdrawnAccountsReport knows about the BankAccount class, and the BankAccount class plays the role of "overdrawnAccounts." However, unlike a standard association, the BankAccount class has no idea that it is associated with the OverdrawnAccountsReport.[2]

At this point, I have covered the basics of the class diagram, but do not stop reading yet! In the following section I will address other useful parts of this diagram.

Beyond the basics

Above, I discussed bi-directional and uni-directional associations. Now I will address the three remaining types of associations, and touch upon interfaces, visibility, and the creation of large class diagrams.

Association class

In modeling an association, there are times when you need to include another class because it includes valuable information about the relationship. For this you would use an *association class* that you tie to the primary association. An association class (also called a *drop class* by my former professor) is represented like a normal class. The difference is that the association line between the primary classes intersects a dotted line connected to the association class. Figure 8 shows an association class for our airline example.

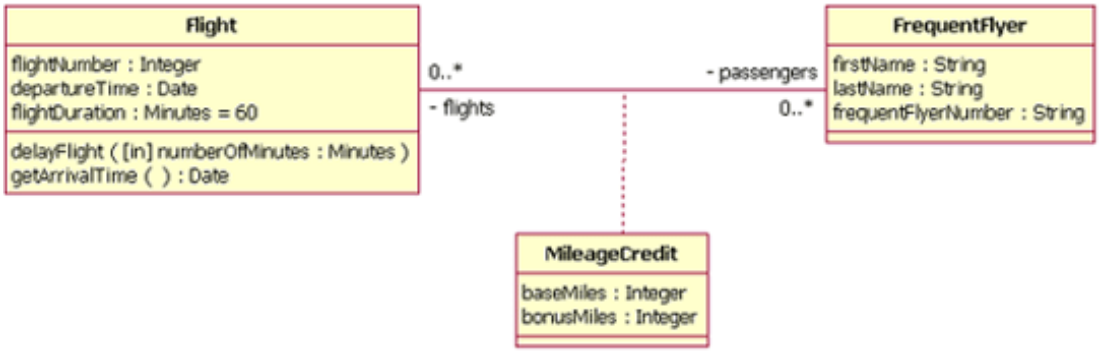


Figure 8: Adding the association class (a.k.a. drop class) MileageCredit

In the class diagram shown in Figure 8, the association between the Flight class and the
FrequentFlyer class results in an association class called MileageCredit. This means that when an instance of a Flight class is associated with an instance of a FrequentFlyer class, there will also be an instance of a MileageCredit class.

Aggregation

Aggregation is a special type of relationship used to model a "whole to its parts" relationship. In basic aggregation relationships, the lifecycle of a *part* class is independent from the *whole* class's lifecycle.

For example, we can think of *Car* as a whole entity and *Car Wheel* as part of the overall Car. The wheel can be created weeks ahead of time, and it can sit in a warehouse before being placed on a car during assembly. In this example, the Wheel class's instance clearly lives independently of the Car class's instance. However, there are times when the *part* class's lifecycle *is not* independent from that of the *whole* class -- this is called composition aggregation. Consider, for example, the relationship of a bank account to its transactions. Both *Bank Account* and *Transactions* are modeled as classes, and a deposit or withdrawal transaction cannot occur before a bank account exists. Here the

Potential multiplicity values

For those wondering what the potential multiplicity values are for the ends of associations, the table below lists the most frequently used multiplicity values along with their meanings.

Transactions class's instance is dependent upon the Bank Account class's instance.

Let's explore basic aggregation and composition aggregation further.

Basic aggregation

An association with an aggregation relationship indicates that one class is a subordinate class (or a part) of another class. In an aggregation relationship, the child class instance can outlive its parent class. To represent an aggregation relationship, draw a solid line from the parent class to the subordinate class, and draw an unfilled diamond shape on the parent class's association end. Figure 9 shows an example of an aggregation relationship between a Company and a Department.

Figure 9: Example of an aggregation association

Potential Multiplicity Values	
Indicator	**Meaning**
0..1	Zero or one
1	One only
0..*	Zero or more
1..*	One or more
N	Only *n* (where *n* > 1)
0..n	Zero to *n* (where *n* > 1)
1..n	One to *n* (where *n* > 1)

Composition aggregation

The composition aggregation relationship is just another form of the aggregation relationship, but the child class's instance lifecycle is dependent on the parent class's instance lifecycle. In Figure 10, which shows a composition relationship between a Student class and a Schedule class, notice that the composition relationship is drawn like the aggregation relationship, but this time the diamond shape is filled.

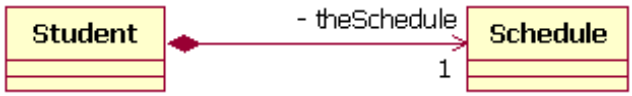


Figure 10: Example of a composition relationship

In the relationship modeled in Figure 10, a Student class instance will always have a Schedule class instance. Because the relationship is a composition relationship, when the Student instance is removed/destroyed, the Schedule instance is automatically removed/destroyed as well.

Reflexive associations

We have now discussed all of the association types. As you may have noticed, all our examples have shown a relationship between two different classes. However, a class can also be associated with itself, using

recursion. This may not make sense at first, but remember that classes are abstractions. Figure 11 shows how an Employee class could be related to itself through the manager/manages role. When a class is associated to itself, this does not mean that a class's instance is related to itself, but that an instance of the class is related to another instance of the class.

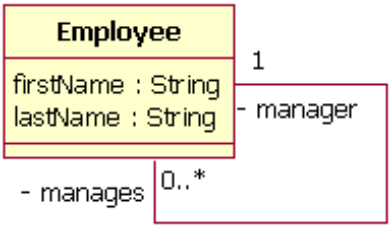


Figure 11: Example of a reflexive association relationship

The relationship drawn in Figure 11 means that an instance of Employee can be the manager of another Employee instance. However, because the relationship role of "manages" has a multiplicity of 0..*, an Employee might not have any other Employees to manage.

Interfaces

Earlier in this article, I suggested that you think of *entities* as classes for the time being but promised a fuller, more complex explanation of entities later on. Now we are ready for that explanation. The fact is, classes are entities, but the term *entities* refers to more things then just classes. Entities also include things like data types and interfaces.

A complete discussion of when and how to use data types and interfaces effectively in a system's class diagram is beyond the scope of this article. So why do I mention data types and interfaces here? There are times when you might want to model these entity types on a class diagram, and it is important to use the proper notation for doing so. Drawing these entity types incorrectly will likely confuse readers of your class diagram, and the ensuing code will probably not meet requirements.

A class and an interface differ: A class can have an actual instance of its type, whereas an interface must have at least one class to implement it. An interface is drawn just like a class, but the top compartment of the rectangle also has the text "<<interface>>", as shown in Figure 12.[3]

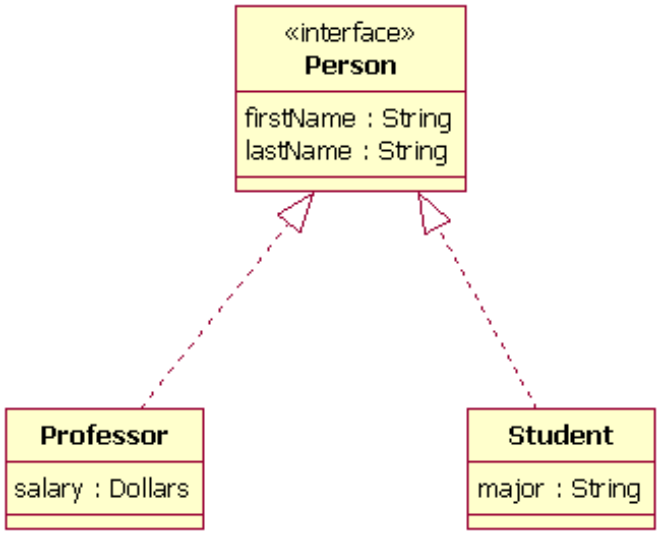


Figure 12: Example of a class diagram in which the Professor and Student classes implement the Person interface

In the diagram shown in Figure 12, both the Professor and Student classes implement the Person interface and do not inherit from it. We know this for two reasons: 1) the Person object is defined as an interface -- it has the "<<interface>>" text in the object's name area, and we see that the Professor and Student objects are *class* objects because they are labeled according to the rules for drawing a class object (there is no additional classification text in their name area); 2) we know inheritance is not being shown here, because the line with the arrow is dotted and not solid. As shown in Figure 12, a *dotted* line with a closed, unfilled arrow means realization (or implementation); as we saw in Figure 5, a *solid* arrow line with a closed, unfilled arrow means inheritance.

Visibility

In object-oriented design, there is a notation of visibility for attributes and operations. UML identifies four types of visibility: public, protected, private, and package.

The UML specification does not require attributes and operations visibility to be displayed on the class diagram, but it does require that it be defined for each attribute or operation. To display visibility on the class diagram, you place the visibility mark in front of the attribute's or operation's name. Though UML specifies four visibility types, an actual programming language may add additional visibilities, or it may not support the UML-defined visibilities. Table 3 displays the different marks for the UML-supported visibility types.

Table 3: Marks for UML-supported visibility types

Mark	Visibility type
+	Public
#	Protected
-	Private
~	Package

Now, let's take a look at a class that shows the visibility types indicated for its attributes and operations. In Figure 13, all the attributes and operations are public, with the exception of the updateBalance operation. The updateBalance operation is protected.

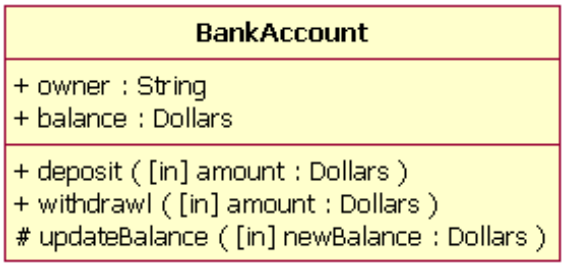


Figure 13: A BankAccount class that shows the visibility of its attributes and operations

Large class diagrams

Inevitably, if you are modeling a large system or a large business area, there will be numerous entities you must consider. Instead of modeling every entity and its relationships on a single class diagram, it is better to use multiple class diagrams. Dividing a system into multiple class diagrams makes the system easier to understand, especially if each diagram is a graphical representation of a specific part of the system.

Placing notes and comments on class diagrams

Typically, class diagrams are accompanied by comments or notes about each class, and/or its relationships, attributes, and operations. These notes can be placed directly on the class diagram, or, better yet, associated with each level (e.g. class, attribute, etc.). IBM Rational Rose® and IBM Rational XDE® allow notes to be associated with each level, as needed, and IBM Rational SoDA® can generate a report that neatly includes this documentation in a Word document.

Conclusion

The class diagram is important because it shows the static structure of the entities in a system. Developers may think that class diagrams are created specially for them, but business analysts can also use class diagrams to model business systems. As we will see in other articles in this series on UML basics, other diagrams -- including the activity, sequence, and statechart diagrams -- refer to the entities modeled and documented on

the class diagram.

Next in this series on "UML basics": the sequence diagram.

Notes

[1]The delayFlight does not have a return value because I made a design decision not to have one. One could argue that the delay operation should return the new arrival time, and if this were the case, the operation signature would appear as `delayFlight(numberOfMinutes : Minutes) : Date`.

[2]It may seem strange that the BankAccount class does not know about the OverdrawnAccountsReport class. The modeling allows report classes to know about the business class they report about, but the business classes do not know they are on reports. The reason for doing this is that it loosens the coupling of the objects and therefore makes the system more adaptive to changes.

[3]When drawing a class diagram it is completely within UML specification to put <<class>> in the top compartment of the rectangle, as you would with <<interface>>; however, the UML specification says that placing the <<class>> text in this compartment is optional, and it should be assumed if <<class>> is not displayed.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!