A PRIMER ON UML CLASS DIAGRAMS

Jeffrey S. Franzone, Assistant Professor
Engineering Technology Department
University of Memphis

Abstract

The Unified Modeling Language (UML) is currently the de-facto visual modeling standard for object-oriented design. The UML provides many modeling diagrams and constructs used to aid the design and development of objected-oriented systems. Each UML diagram presents a unique view of the object-oriented system under design.

The most common UML modeling diagram is the Class Diagram. Classes represent the modeling framework from which all object-oriented systems are designed. They are the "blueprints" of object-oriented systems, defining the attributes and behaviors of objects, which in turn, provide the functionality of object-oriented systems. A typical class diagram groups logically-related classes together to show the relationships of the classes to one another.

Since it is highly likely that programming students will encounter the UML in industry, it is imperative that computer science and computer engineering technology instructors begin to introduce students to the UML in their object-oriented programming courses. This tutorial is designed for just that purpose.

The tutorial introduces the UML Class Diagram-its syntax and constructs. Specifically, seven methods expressing the relationships between classes are examined through detailed examples and pictorials. Students will learn how to model class diagrams using associations, association classes, aggregations, compositions, generalizations, realizations, and dependencies. Using this tutorial, programming instructors can quickly teach the fundamentals of UML class diagrams to their students. Students can use the tutorial as reference material as they develop class diagrams for their own objected-oriented programs.

I. Introduction

There are seven basic ways to show the relationships between classes using the UML. They are:

- Associations - "knows about" or "aware of" relationships without instances of or references to other classes as data members. Does not indicate "whole/part" relationships.
- Association Classes - Defines classes that convert many-to-many relationships between two other classes into one-to-many relationships. Occasionally, an additional class association is required to show ownership of the association class.
- Aggregations - "comprised of" relationships with references to other classes, not instantiated in the aggregate class, as data members. Indicates "whole/part" relationships where the destruction of the "whole" does not destroy the "parts".
- Compositions - "has a" or "contains a" relationships with instances of other classes, or references to other classes instantiated within the composite class, as data members. Indicates "whole/part" relationships where the destruction of the "whole" destroys the "part".
- Generalizations - "is a" relationships where more specific classes (called derived classes) inherit behaviors and attributes from general classes (called base classes) while adding new behaviors and attributes. Allows classes to extend the functionality of already existing classes.
- Realizations - "is a" relationships where base classes specify a set of methods (an interface) that must be implemented by derived classes. Classes that describe interfaces contain no attributes.
- Dependencies - "uses a" or "depends on" relationships where classes use or depend on the services of other classes in such a manner that a change in the provider class (the class offering the services) may affect the operation of the user class (the class using the services).

The following sections discuss each of the above UML class relationships. Examples are used to illustrate how each of these relationships are modeled as class diagrams in the UML. Class diagrams typically depict the structural connections (and implicit object interactions) of classes

used in a particular application. However, class diagrams can also depict class relationships that are independent of any application.

The example screenshots were taken from Rational Rose 2000 Enterprise Edition, a visual modeling tool for object-oriented design and analysis. Courtesy is given to Rational Software Company for their use of the software.

II. Classes

Classes represent the modeling framework from which all object-oriented systems are designed. As shown in Figure 1, the UML visually represents classes as a rectangular box with three compartments: *Name, Attributes, and Operations*.

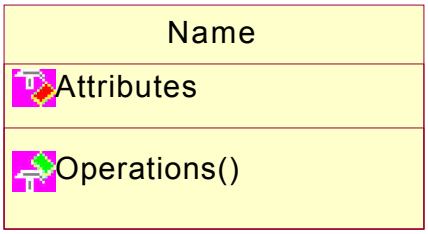

Figure 1. UML Class symbol.

The Name compartment includes the name of the class and an optional stereotype that distinguishes the nature of the class from other classes (see the section *Associations* for more information on stereotypes). All classes will have at least a Name compartment.

The Attributes compartment is an optional compartment that lists the attributes or states of a class. The complete format of an attribute is stated as:

<visibility> <<<stereotype>><name of attribute> : <type = initial value>

The visibility of an attribute refers to its access-specifier (public, private, protected, or package); the attribute's scope inside and outside of its class. A + represents public, - represents private, # represents protected, and ~ represents package. Some UML tools such as Rational Rose use icons instead of character symbols to denote visibility. Attributes that are const, static, or extern are typically expressed as stereotypes.

The Operations compartment is an optional compartment that lists the operations, services, or behaviors of a class. The complete format of an operation is stated as:

<visibility> <<<stereotype>>> <name of operation(args list (name : type)> : <return type>

Methods that are const, static, virtual, or friends are typically expressed as stereotypes.

Figure 2 shows a typical example of a Class modeled with the UML.

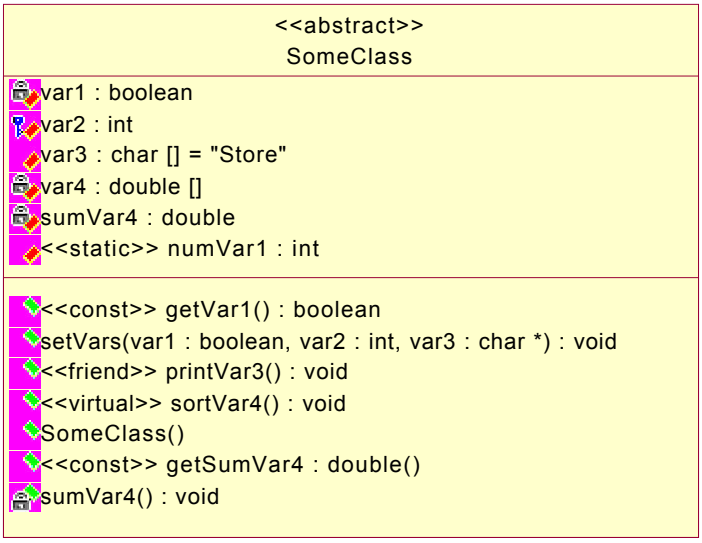

Figure 2. Modeling a Class with the UML.

III. Associations

Associations define structural links between different, but related, classes. More specifically, associations define "knows about" relationships between classes without needing instances of or references to other classes as data members. Class relationships of this type usually communicate by sending messages via class or object references. A bi-directional association means that the classes involved in the association know about each other (each class is passed a reference to or an instance of the other class). A unidirectional association means that only one class in the association knows about the other class.

Figure 3 illustrates a bidirectional association between two generic classes, Customer and Order. A bidirectional association is depicted with a solid darkened line connected between two classes. You can select a descriptive name for the association that describes the purpose of the association (placed near the middle of the association line) or provide "roles" at each end of the association. Roles specify the function each class presents to the other class in the association. Roles are often preferred over association names because good role names generally convey more information about the association (see Figure 4).

Figure 3. A bidirectional association with multiplicity values.

Association names may be attached to a small arrow indicating the direction the association should be read (although the grammatical tense and context of the association name usually

implies the direction). This is only necessary for bidirectional associations since unidirectional associations are marked with feathered arrows. This notation is sometimes useful when you want to emphasize the direction of an association even though structurally both classes know about each other.

Most associations use multiplicity values to indicate how many objects of one class participate with objects of the other class. Table 1 lists the multiplicity notation used in Class diagrams. Absence of multiplicity values could either indicate one-to-one relationships or many-to-many relationships between classes. If multiplicity values are absent from an association, the context of the classes participating in the association usually determines the implied multiplicity.

MULTIPLICITY	NUMBER OF OBJECTS
0	Zero.
1	One.
m	m.
0..1	Zero or one.
m..n	At least m, but not more than n.
*	Any number.
0..*	Zero or more.
1..*	One or more.

Table 1. Multiplicity notation and meanings.

As shown in Figure 4, a Course object can be aware of 1 up to 25 Student objects. The Course object knows about the number and names of students enrolled because it is passed a reference to a Student object when the enrollStudent() method is invoked. The Course object stores a Student object's name in a private String array. Since the association is bidirectional, you can also say that a Student object can be aware of 0 up to 6 Course objects. The Student object knows about the number of courses it is enrolled in and the names of the courses when the enrollCourse() method is invoked because it is passed a reference to a Course object. The Student object stores a Course object's name in a private String array.

Note that both the Course and Student classes know about each other through object references passed into member methods. Nowhere in the Course class is an instance or reference to a Student object stored as a data member and vice versa. And yet, we can still say that a Course object "knows about" Students (at least one).

Another important property of associations is they do not indicate "whole/part" relationships. In other words, Course and Student objects can survive independently of each other. Although logically it might be correct to think that Students are a part of Courses or Courses contain Students, the particular structural connection between the classes in Figure 4 does not support these notions. If a Course dies, Students still survive (quite happily I might add). If Students of a Course die (an unfortunate predicament, to say the least), the Course still goes on (even without any students). However, using aggregation or composition relationships (discussed in later sections), we can express the notion that Courses die if no students are enrolled ("whole/part" relationships).

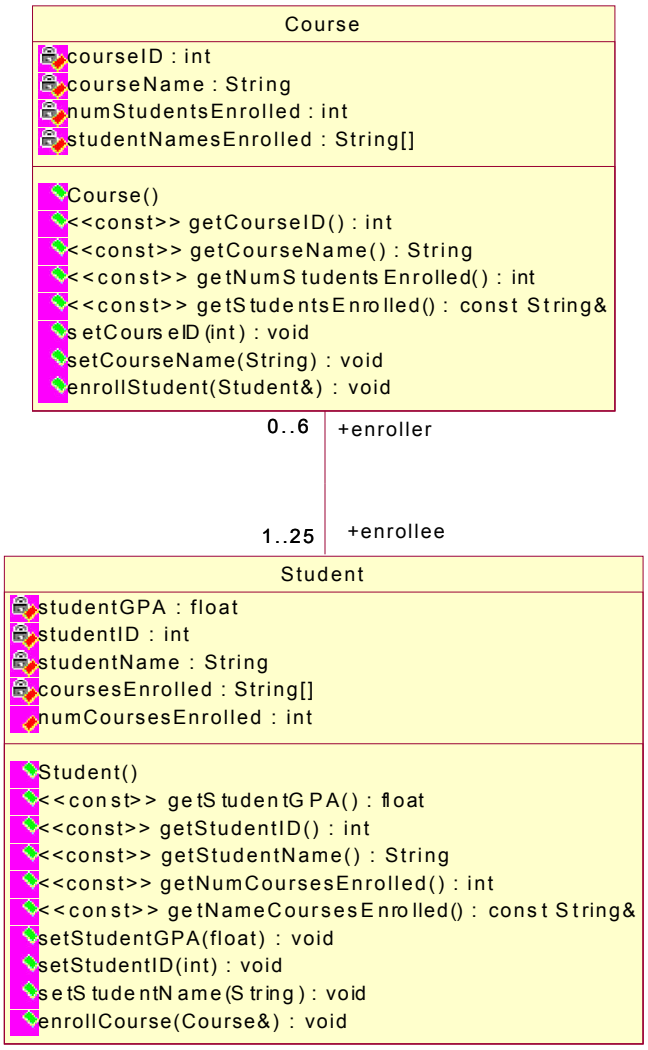

Figure 4. Bidirectional association between the Course and Student classes.

Figure 4 also illustrates the use of "stereotypes" to signify special properties of some class member functions. Stereotypes represent user-defined modeling elements that extend the basic modeling vocabulary of the UML. Stereotypes are enclosed within guillemets << >>. Stereotypes help you to distinguish and differentiate unique purposes or uses of classes, member functions, data members, associations, aggregations, compositions, dependencies, realizations, and generalizations based on the context in which these modeling elements are used. The UML has many predefined stereotypes that convey special meaning to a modeling element. For example, member functions that do not modify class data are typically declared as 'const' functions. The stereotype <<const>> before a function signature clearly signifies that the function is a 'const' function. You will see more examples of stereotypes throughout this tutorial.

Figure 5 shows a variation of the Course/Student relationship as a unidirectional association. Unidirectional associations are shown with feathered arrows pointing to the class that the other class knows about.

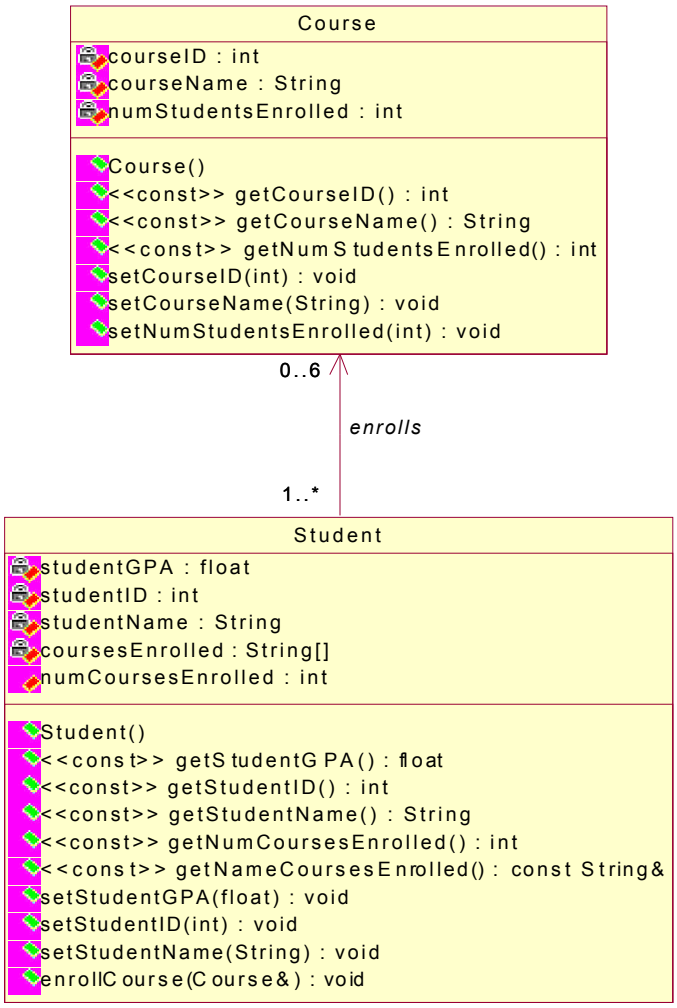

Figure 5. Unidirectional association between the Course and Student classes.

As seen in Figure 5, Students know about the Courses they are enrolled in but Courses do not know implicitly how many students are enrolled (enrollment information is entered manually). Also notice that the roles are replaced by the “enrolls” association indicating that a Student object enrolls in a Course object.

Some associations require rules or conditions to be applied. These rules or conditions are referred to as *constraints*. Figure 6 is an example of a Sort class that contains a method that performs the bubble sort algorithm on a list of integers stored in a Container object. The constraint, shown enclosed within curly braces, signifies that the Sort class performs sorting in ascending order only.

Figure 7 shows an “xor” constraint between three classes. Here, an instantiated object of the Container class can either be a linked list “or” an array, but not both.

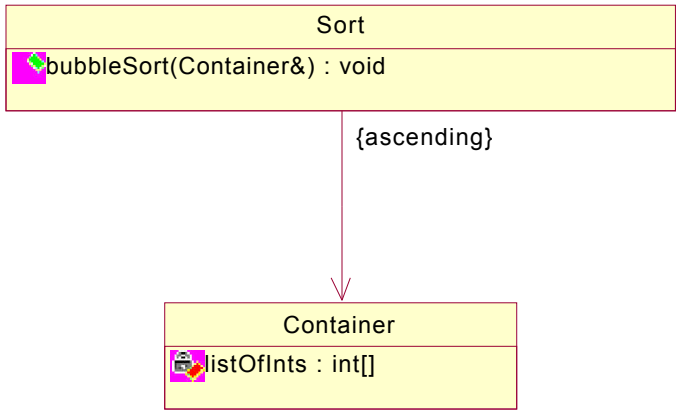

Figure 6. A unidirectional association containing a constraint.

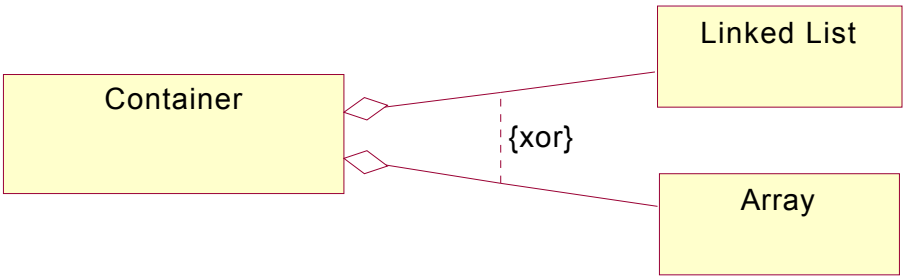

Figure 7. An “xor” constraint.

IV. Association Classes

Association classes are used to express one-to-many relationships between two classes that inherently have many-to-many relationships. Take for example, the Course and Student classes previously discussed. Typically, there are many-to-many relationships between a Course and Student because a Student may be enrolled in more than one Course and a Course may have more than one Student. Using a Classroom association class, as shown in Figure 8, we can associate a particular Student with a particular Course by the Classroom in which the Course is taught. In other words, one Classroom will be assigned to a particular Course in which a particular Student has enrolled. The Classroom association class presents a unifying association between one Course and one Student.

Typically, an association class implicitly knows about both classes so additional associations between the two classes and the association class are usually not needed. However, occasionally, an association class may only know about one of the classes. Consequently, an additional class relationship must be provided to indicate ownership of the association class. As

seen in Figure 8, a Course is assigned a Classroom and thus has ownership of the Classroom (we don't want Students to decide where to teach the Course, do we?).

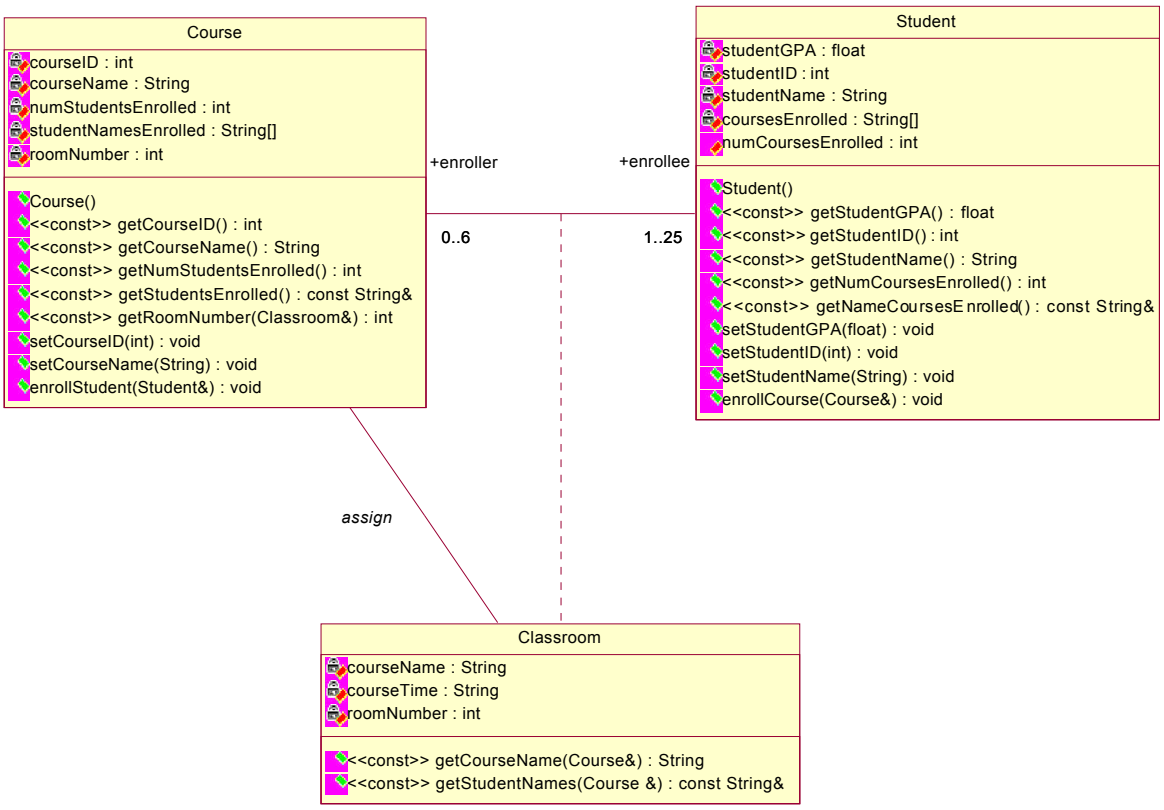

Figure 8. Association class relationship with ownership.

V. Aggregations

Aggregations are class associations that indicate "whole/part" relationships where the "parts" comprise the "whole" (implying the "parts" are logically independent of the "whole"). It differs from a generic association in that one class (the whole) contains object references or pointers to one or more classes (the parts) as data members. References to other class types stored as data members in the aggregate (or whole) class implies that the parts exist outside of the aggregate class. However, any instances of the aggregate class must be initialized with (or passed) valid references to the parts if the aggregate class is to function properly. Aggregation defines a "loosely-coupled" dependency on the parts to the whole. If the aggregate instance is destroyed, its parts still remain alive since their instances are declared outside of the aggregate class. However, the whole cannot exist without its parts.

To represent an aggregate relationship between classes, you use a solid darkened line between the classes with a hollow diamond attached to the aggregate class. Figure 9 shows another variation of the Course and Student classes. The Course class defines an array of Student pointers as data members. Under this scenario, Course objects cannot perform certain operations

(such as calculating the mean GPA of all enrolled students) without valid references to enrolled Student objects. The aggregate property of the Course class to the Student class implies a closer structural connection than what a regular association provides. If a Course object is destroyed (the whole), its Students (the parts) are not. And yet, a Course object cannot meaningfully perform its full capabilities without its Students.

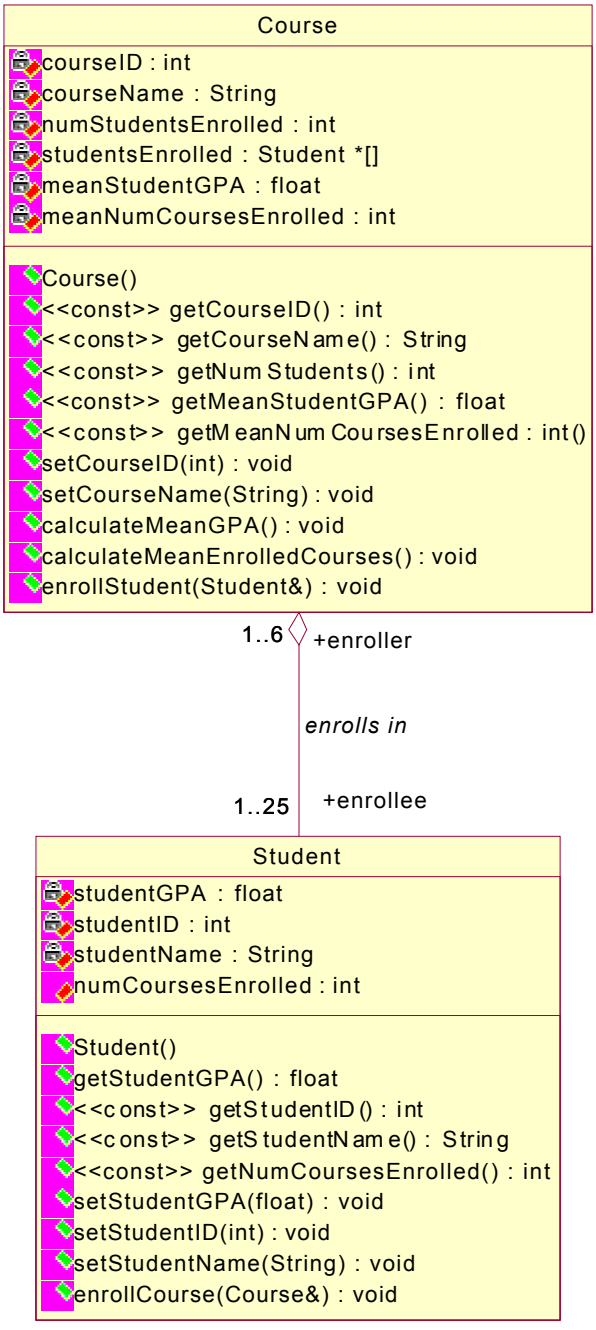

Figure 9. Aggregate association.

VI. Compositions

Compositions are special types of aggregate associations that also indicate “whole/part” relationships between classes. It differs from aggregation in that one class (the whole) contains direct instances (or references of instances instantiated by the whole) of one or more class objects (the parts) as data members. This implies that the “parts” can only exist as part of the “whole” (the composite). If the “whole” is destroyed, so too are its parts. The “parts” cannot exist without the “whole”. Composition, unlike aggregation, defines a “tightly-coupled” dependency on the parts to the whole, a so-called “has a” or “contains a” relationship. More precisely, since composite classes have ownership over their parts, the lifetime of the “parts” are dependant on the lifetime of the “whole”.

To represent a composite relationship between classes, you use a solid darkened line between the classes with a filled diamond attached to the composite class. As shown in Figure 10, the composite class contains one instance of the Date class as a data member. The Date object ‘hireDate’ only exists when an instance of the Employee class is created. When the Employee class is destroyed, so too is the ‘hireDate’ object. Figure 10 also illustrates the use of an optional “stereotype” to name the composite relationship between the Employee and Date classes. For example, composite associations typically instantiate objects of other classes as data members so that those objects can be considered as part of the composite class. The predefined stereotype <<instantiate>> conveys the underlying nature of composite relationships. Along with stereotypes, you can also provide specific association names (or roles) that define logical descriptions between classes (see Figure 11).

Up to this point, all of the previous class diagram examples have neglected to show the composite relationship between the class String and its parts (other classes that use String). Now that you understand the concept of composition, Figures 12 and 13 show a more complete class diagram of the Employee/Date relationship and the Course/Student relationship, respectively. Class String is assumed to be a vendor-supplied class library.

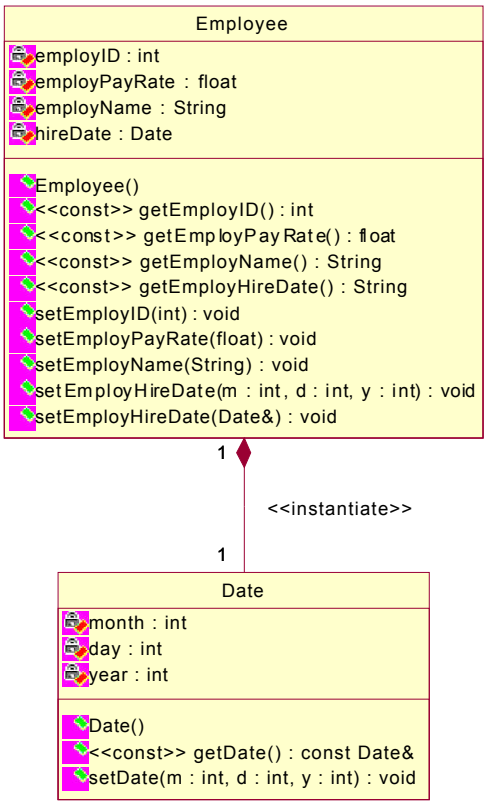

Figure 10. Composite association with stereotype.

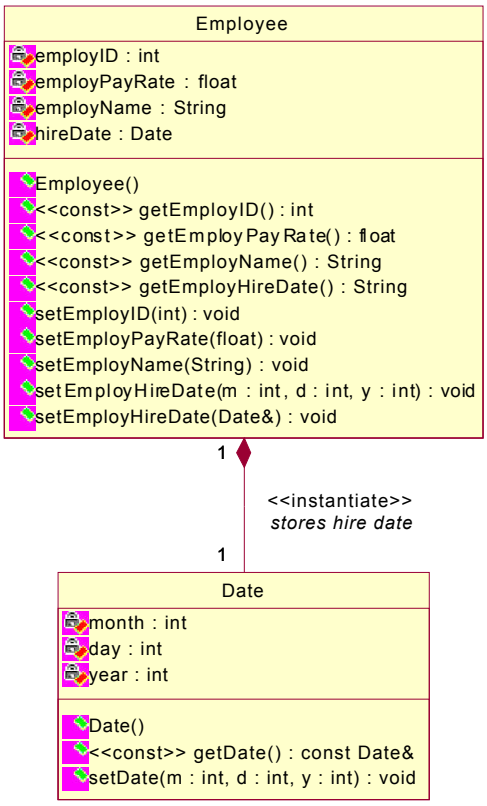

Figure 11. Composite association with stereotype and association name.

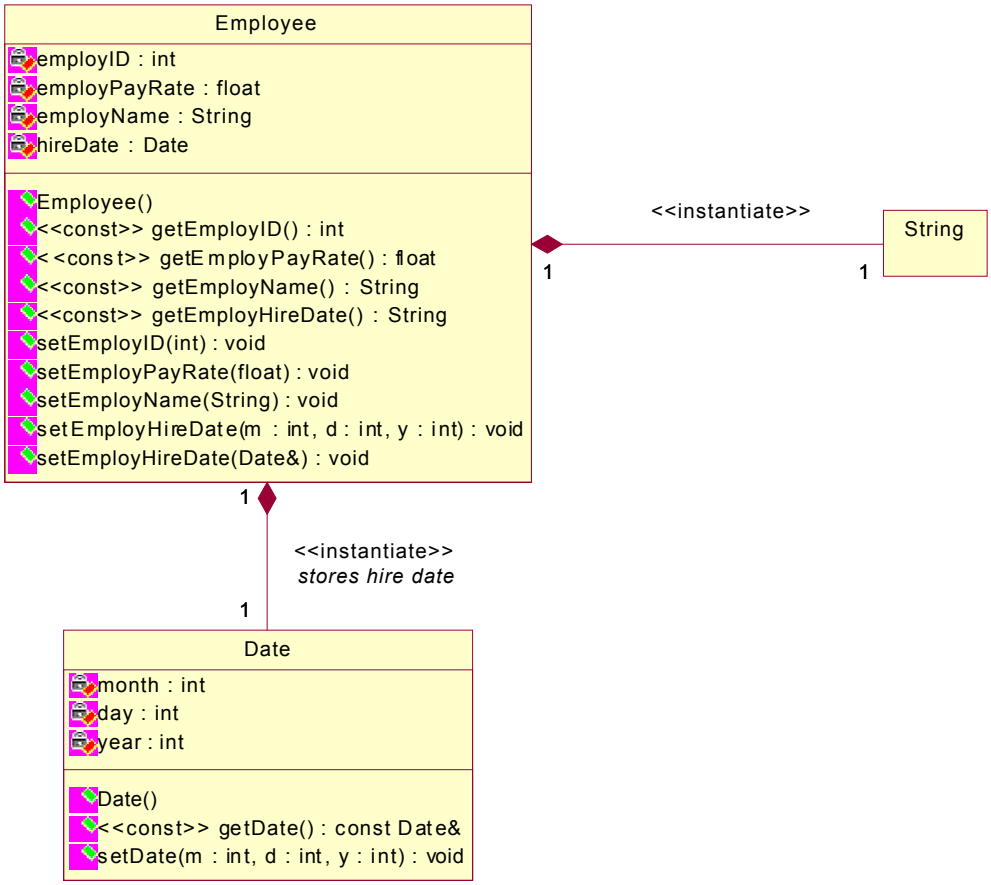

Figure 12. Employee/Date composite association with String class.

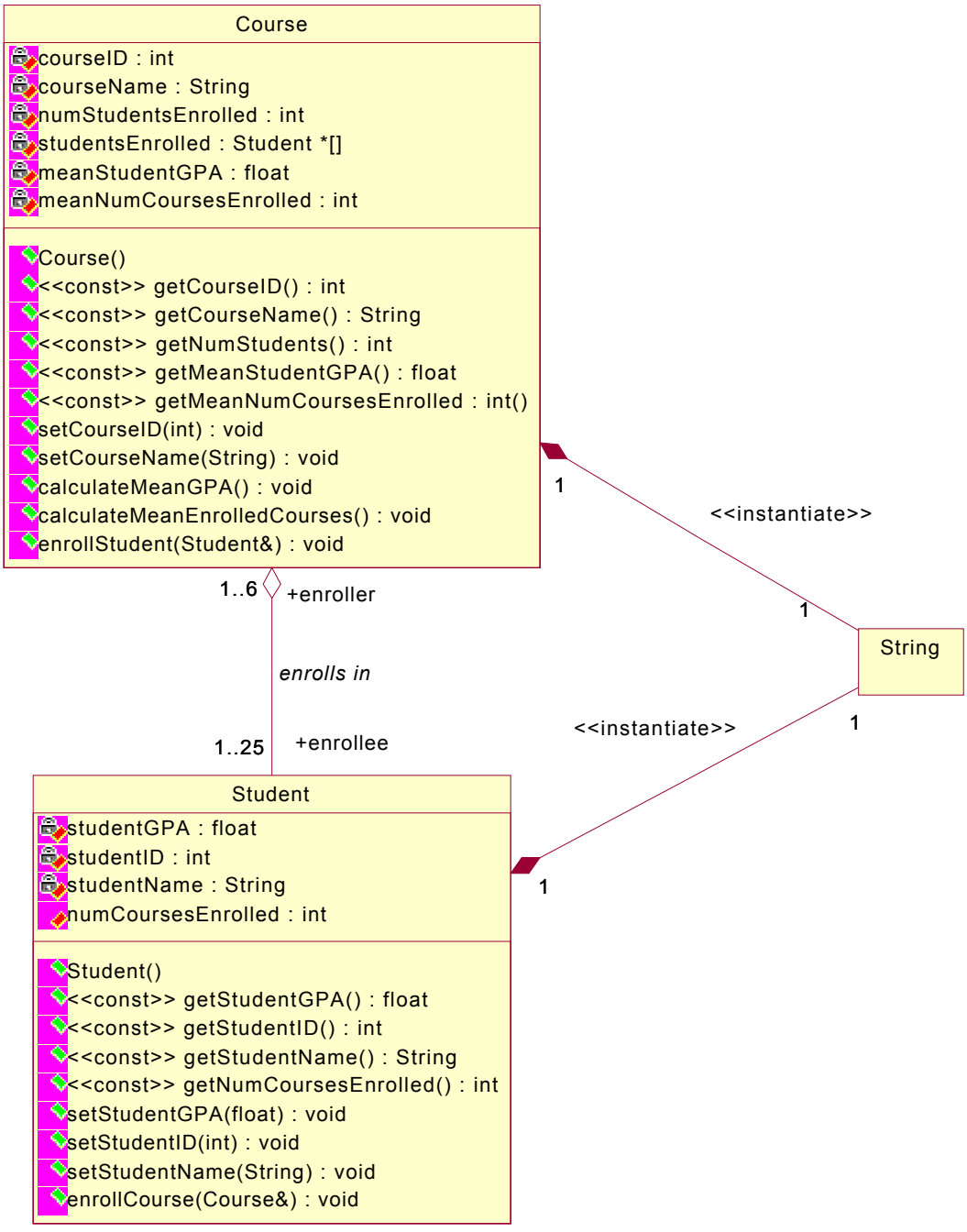

Figure 13. Course/Student composite associations with String class.

VII. Generalizations

Generalizations specify inherited relationships between more general classes (called base classes or superclasses) and more specific classes (called derived classes or subclasses). A derived class "inherits" common attributes and behaviors from its base class while defining additional attributes and behaviors unique to the class. For example, we might categorize common attributes and behaviors of animals in an Animal class. Some common attributes that all animals possess include size, weight, life-expectancy, and diet. Some common behaviors might include eating, sleeping, and mating. Although the Animal class contains common animal characteristics, different animals may possess additional, more specific characteristics and behaviors. For instance, we can derive a Dog class from the Animal class that includes all of the attributes and behaviors of the Animal class but adds additional attributes and behaviors common to dogs such as a tail, four legs, barking, and running. We can further derive the Dog class into specific types of dogs which include attributes such as intelligence, exercise habits, and disposition (see Figure 14).

Figure 14 illustrates how the UML expresses generalizations as applied to the Animal and Dog classes. Generalized relationships are shown with a solid darkened-line with a hollow triangle attached to one end that points to the more general class from the more specific class. Although the nature of a generalization can also be named or stereotyped to emphasize a logical idea or underlying connection between the classes involved, generalizations are often not labeled. Also, multiplicity values are seldom labeled because generalizations show inherited relationships among derived and base classes independent of any particular application. This implies multiplicity values of 0..* to 0..* between derived and base classes except for abstract base classes which have a multiplicity of 0..0 (objects cannot be instantiated for abstract classes).

From a logical perspective, generalizations depict one-way "is a" relationships between classes. For example, a Dog "is a" type of Animal but an Animal may not be a Dog (it could be a Tweetie Bird, for example). Similarly, a Dalmation "is a" Dog which is a type of Animal but a Dog is not necessarily a Dalmation (it could be other breeds as well). Contrast the "is a" relationship with the "has a" relationship. You wouldn't say that a Dog "has a" Dalmation (rather, a Dalmation "is a" Dog) or an Employee "is a" Date object (rather, an Employee "has a" Date object).

From a structural perspective, generalizations imply the creation of new "standalone-but-related" classes from parts of previously defined classes through the object-oriented property of inheritance. Contrast this with aggregate or composite class relationships where the aggregate or composite class depends on the creation of objects from other, possibly unrelated classes.

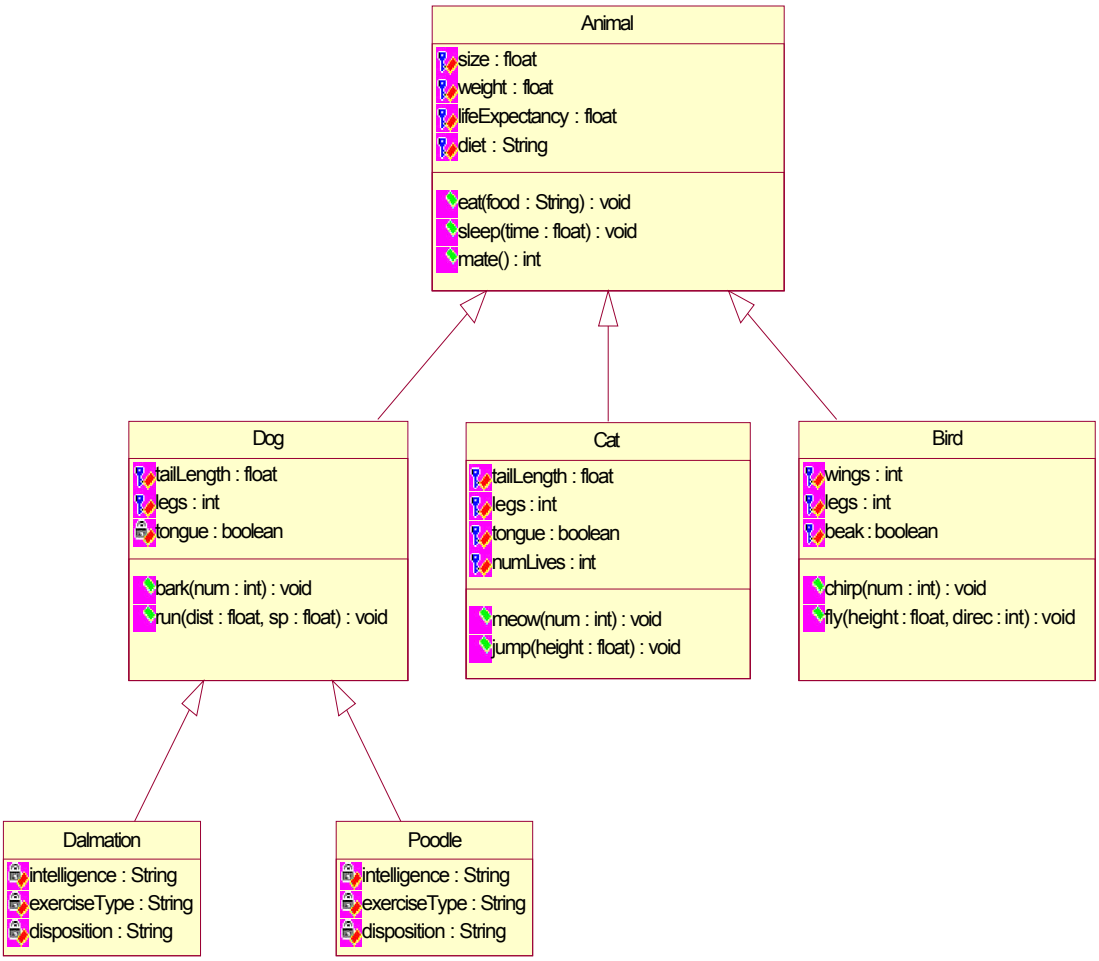

Figure 14. Generalizations convey the OO principle of inheritance.

VIII. Realizations

Realizations are special forms of generalizations which connect the services (or operations) described in interface classes to the implementation classes that actually provide the services. Interfaces merely describe services or operations that are not implemented and they contain no attributes. Instead, its derived classes must provide the actual implementations for these services. In other words, interface classes provide a predefined contract that state what services must be implemented by its derived classes. Realization associations simply connect implementation classes to their interface classes.

Java directly supports the notion of interfaces; in C++, an abstract class that only contains pure virtual functions (no attributes) is called an interface. As shown in Figure 15, realizations are shown as a dashed arrow with a hollow triangle attached to the end that points to the interface class from the implementation class. It is common to use the <<interface>> stereotype in the name compartment of the interface class to help distinguish this type of class. Multiplicity

values between interface and implementation classes are assumed to be 0..0 (for the interface class) to 0..* (for the implementation class).

Figure 15 also shows another UML modeling element called a *Note*. Notes are useful when you want to clarify or provide additional information about your classes.

Figure 16 depicts a typical scenario where a user is operating an email application and selects the 'Send Message' button. The button the user selects is a special type of button called a *SendButton*. The SendButton class is derived from an interface called Button which represents a generic button type. The Button interface describes a method called 'actionPerformed()' that all its derived classes must implement. The SendButton class inherits the 'actionPerformed()' method interface and must implement the specific actions taken when the user selects the instance of the SendButton (called sendMessageButton, in this example). When the user selects the 'sendMessageButton', its 'actionPerformed()' method calls the email applications' 'sendMessage()' method which actually handles the details of sending the message to another user. Here, the SendButton class implements a specific version of the Button class tailored to the application it is used in. Other button types can be derived from the same Button class while implementing unique behaviors and attributes suitable for the application.

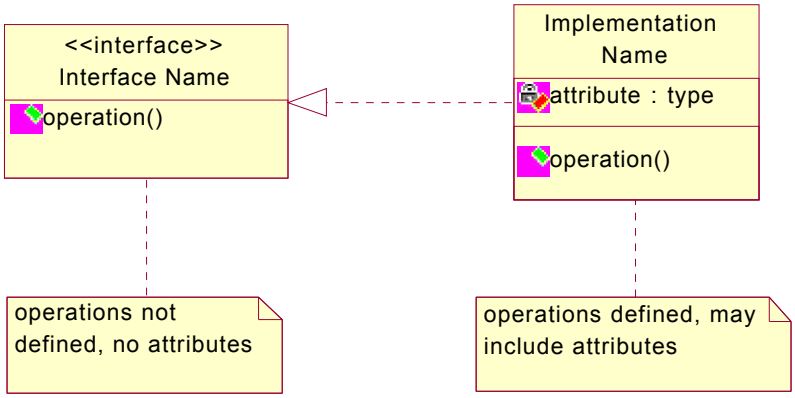

Figure 15. Realization notation.

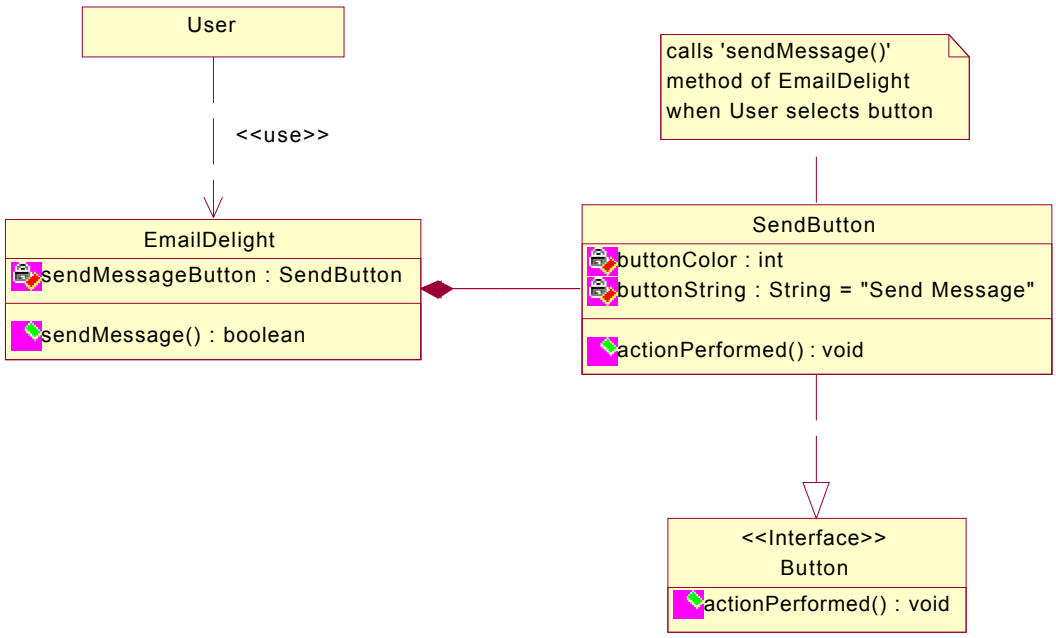

Figure 16. The SendButton class "realizes" the Button interface class.

The UML also provides an interface icon symbol called a *Pin* that is often used to simplify class diagrams containing interfaces. Figure 17 redraws the class diagram of Figure 16 using the interface icon symbol.

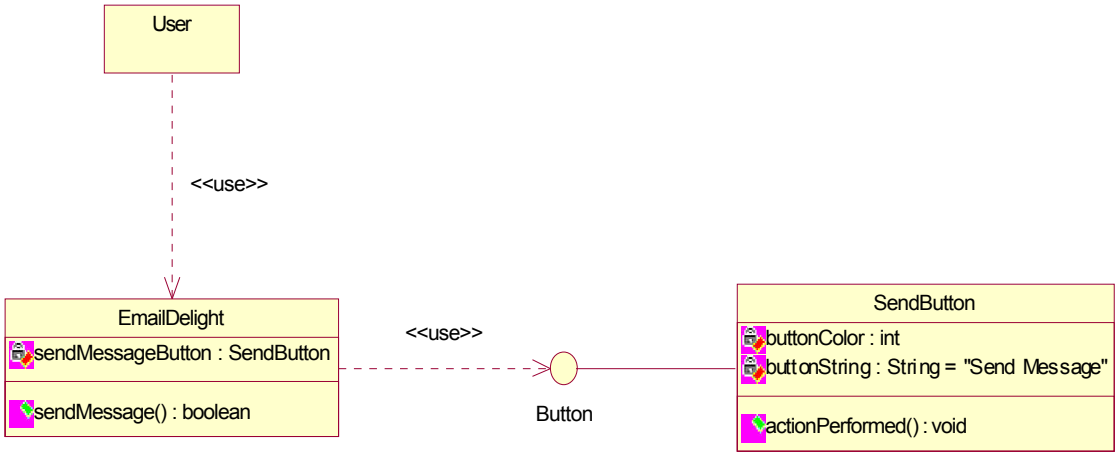

Figure 17. Figure 16 redrawn using the UML interface icon ("Pin" notation).

IX. Dependencies

You may have noticed from Figure 16 the dashed-line, feathered arrow pointing from the User class (the dependent class) to the EmailDelight class (the independent class). This symbol denotes a dependency relationship between the two classes in which a User class "uses" another class. The User depends on the email application to send email. If the email application should fail, so too are the attempts by the User to send email to another user. In a dependency relationship, the arrow always points to the independent class from the dependent class.

Dependencies emphasize "uses a" or "depends on" relationships between classes. Since stereotypes are often used to distinguish class differences, it is common to name dependency relationships with stereotypes. Dependencies typically involve classes that use the services of other classes without any structural connection between the classes. If the classes providing services happen to change their implementation, this could affect the operation of the classes using those services. Hence, the dependency of dependent classes using the services of independent classes.

Some commonly named dependencies include <<friend>>, <<instantiate>>, and <<include>> (although, these stereotypes could name other types of associations as well). A class might contain several friend functions that are granted access to the dependent classes' private data. For example, to help determine the amount of merit pay given to an Employee, a Human Resource class might contain a friend function that is granted access to private disciplinary actions taken against an Employee without being related to the Employee class. This information, along with the number of projects completed by the Employee, is then used to determine the amount of merit pay given to the Employee. This is shown in Figure 18 as a <<friend>> dependency. Friend classes are also possible which allow any of the member functions of a friend class access to the granting classes' private data.

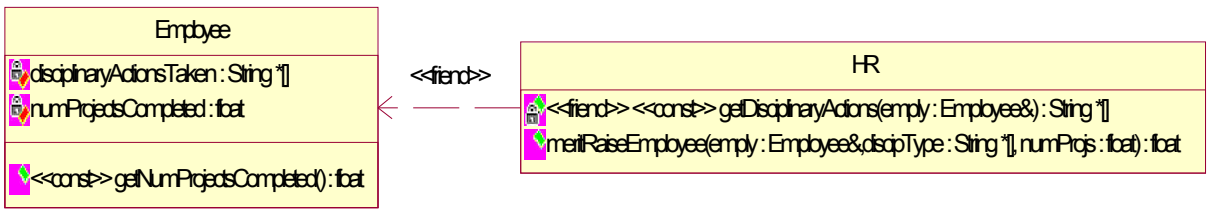

Figure 18. A <<friend>> dependency.

An example of an <<instantiate>> dependency is shown in Figure 19. Here, a Client class wishes to use a service provided by a Server class. The Client object connects to the Server class by instantiating a Server object within a Client method. After the Client method is finished using the services, the Client object disconnects from the Server object. If the Server object either changes the name of its service, the format of its String argument, or the type of its argument, the Client object would not be able to update the customer address successfully.

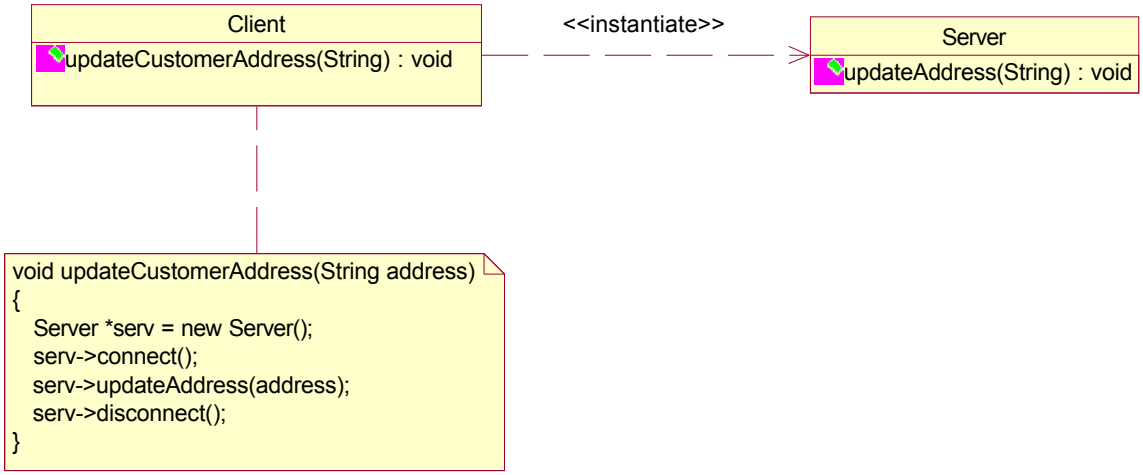

Figure 19. An <<instantiate>> dependency.

An <<include>> (or <<access>>) dependency is one in which classes in one package depend on classes of another package. In Java, packages are directory structures which store related classes (a logical library of classes, so to speak). C++ doesn't directly support packages but it does support namespaces. C++ namespaces are similar to the package concept where related classes are grouped together into logical units. Packages are grouped into logical units using directory structures while namespaces are grouped into logical units using programmatic language features. (Note: The terms "package" and "library" are often used synonymously. However, unlike packages, pure class libraries imply that related classes are grouped together and converted into a special format by a Librarian program and represented by a single filename (usually suffixed with a .lib extension). Typically, C++ namespaces actually include "library" files.) Figure 20 shows the UML symbol for a Package.

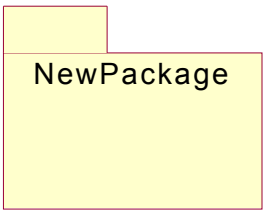

Figure 20. UML Package symbol.

Figure 21 shows an <<include>> dependency between the Time class and the C++ namespace 'std'. The Time class contains a method called 'printTime()' which uses the cout object to display the current time. The cout object is part of the iostream class and is contained within the 'std' namespace. By including the iostream header file and the 'std' namespace in the Time class definition file, Time objects can utilize the services of cout.

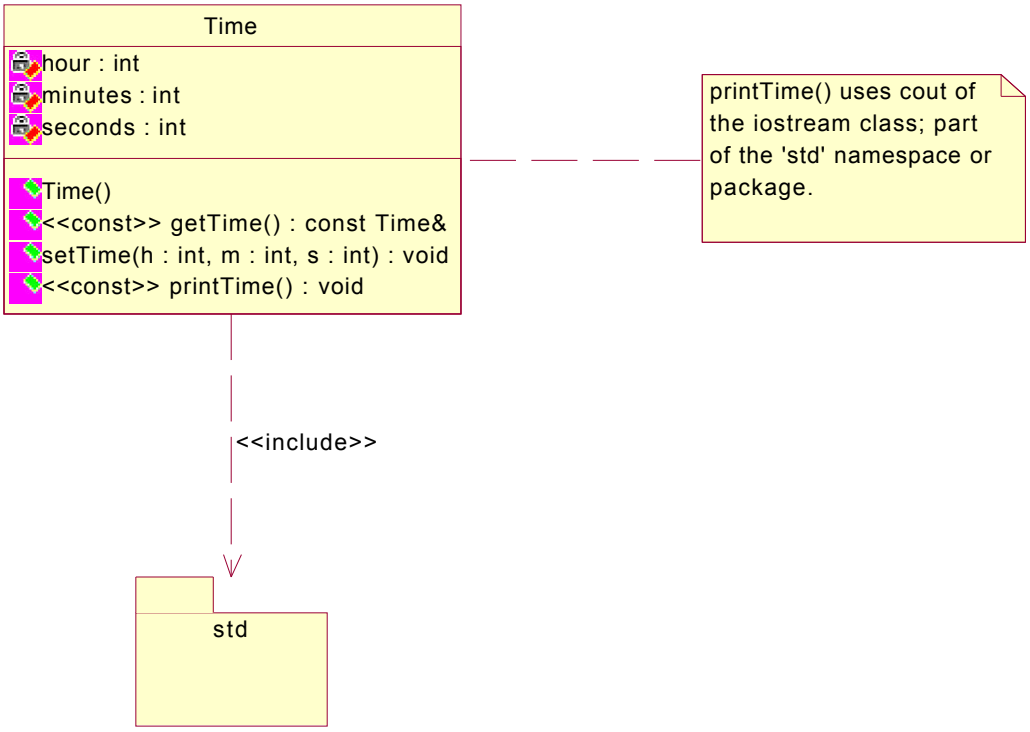

Figure 21. An <<include>> dependency.

X. Parameterized Classes

Typically, class development involves the definition of attributes and operations of classes. Specific data types are assigned to attributes, function arguments, and return types. Class methods often must modify or access class data members as well as define local variable types. Class implementations that are tied to specific data-types are said to be fixed in their implementation and instantiation of class objects. So, for example, if a Stack of integers class is defined, the implementation of the class is directly tied to the int type and any Stack objects instantiated from the class will always be integer Stacks. But you would also agree that a Stack of doubles, characters, or String types is also useful in some applications. Without parameterized classes, we would need to either provide many overloaded methods to handle the different data types or use inheritance to derive new Stack-type objects. These techniques aren't always practical or easy to implement.

Parameterized classes, sometimes referred to as Template classes or class Templates, are generic class definitions that are not tied to specific data-types until compile time. Instead, specific data-types for class implementation-dependent code is passed to the class template when objects of the class are instantiated. The compiler binds the passed type parameters to the class implementation-dependent code in the class template and generates a type-specific class from which class objects can be instantiated. Therefore, a generic Stack class, for example, can be written that is not tied directly to any particular data-type. Then at compile time, a type-specific Stack class can be specified that will bind the selected data type to a Stack implementation of

that type. In this manner, the same generic Stack code can be used to generate a Stack class of integers, characters, doubles, Strings, and other types. As you might guess, class templates are extremely useful in the development of generic class data structures whose operations are identical for a range of different data-types.

Figure 22 illustrates the basic notation for parameterized or template classes. The template class looks like a regular class type with attributes and operations listed in their respective compartments except for the added dashed-line rectangle located at the upper right corner of the class. Listed inside this rectangle are the formal type parameters used to indicate implementation-dependent code (or more specifically, implementation-dependent data types) within the class template definition. The bounding class is the same name as the template class but represents a type-specific version of the template class based on the passed type parameters. It is this generated type-specific version of the template class that type-specific class objects can be instantiated. The template class and the bounding class are connected with a dependency relationship with the arrow pointing from the bounded class to the template class. The dependency relationship is labeled with a <<bind>> stereotype and the actual parameter types that are bound to the template class are listed next to the stereotype.

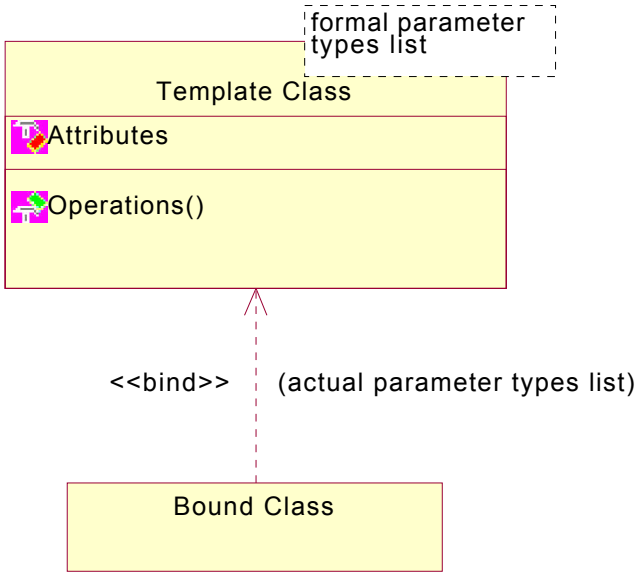

Figure 22. Template class notation.

Figure 23 shows an example of a simple Stack of integers class created from a Stack class template.

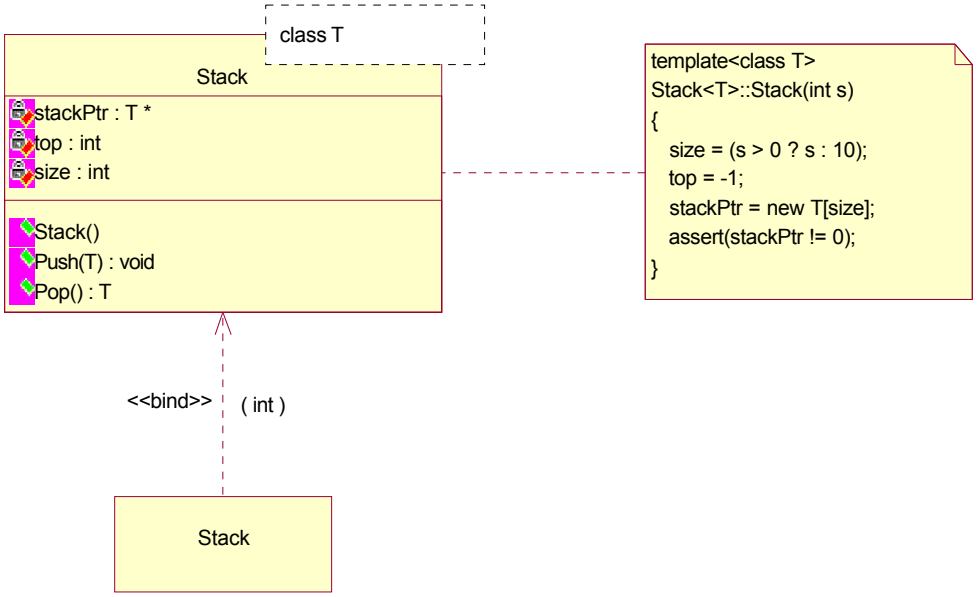

Figure 23. A Stack of integers class created from a Stack class template.

XI. Conclusion

Although there are many visual and notational elements in the UML, the previous sections have emphasized the more common elements you are most likely to work with when drawing and analyzing class diagrams. In particular, seven common class relationships and their associated notations were discussed that describe, distinguish, or differentiate the structural and/or logical connections among different classes. The class relationships described by associations, association classes, aggregations, compositions, generalizations, realizations, and dependencies and expressed in Class diagrams impart significant insight into the OO development and analysis of classes and the applications they are associated with.

Bibliography

1. Rational Rose Enterprise Edition 2000, Rational Software Corporation, 1991-2000.
2. UML Explained, Kendall Scott, Addison-Wesley, 2001.
3. OMG Unified Modeling Language Specification Version 1.4, September 2001.

JEFFREY FRANZONE

Jeffrey Franzone currently teaches in the Engineering Technology Department at the University of Memphis as an Assistant Professor. He teaches C, C++, Java, and microprocessor courses. He has 7 years industrial experience working as an engineering technologist both in hardware and software design and testing and 9 years teaching in Engineering Technology. Jeff received a Bachelor's degree in Electronics Engineering Technology at California State University at Long Beach in 1991 and received a Master's degree in Computer Technology at Arizona State University in 1996.