5. UML Class Diagrams

UML class diagrams show the classes of a system, their interrelationships, and the operations and attributes of the classes. They are used to

- explore domain concepts in the form of a domain model,
- analyze requirements in the form of a conceptual/analysis model,
- depict the detailed design of object-oriented or object-based software.

A class model comprises one or more class diagrams and the supporting specifications that describe model elements, including classes, relationships between classes, and interfaces.

5.1 General Guidelines

Because UML class diagrams are used for a variety of purposes—from understanding your requirements to describing your detailed design—you will need to apply a different style in each circumstance. This section describes style guidelines pertaining to different types of class diagrams.

87. Identify Responsibilities on Domain Class Models

When creating a domain class diagram, often as part of your requirements modeling efforts, focus on identifying

Table 3. Visibility Options on UML Class Diagrams

Visibility	Symbol	Accessible to
Public	+	All objects within your system
Protected	#	Instances of the implementing class and its subclasses
Private	-	Instances of the implementing class
Package	~	Instances of classes within the same package

responsibilities for classes instead of on specific attributes or operations. For example, the *Invoice* class is responsible for providing its total, but whether it maintains this as an attribute or simply calculates it at request time is a design decision that you'll make later.

There is some disagreement about this guideline because it implies that you should be taking a responsibility-driven approach to development. Craig Larman (2002) suggests a data-driven approach, where you start domain models by identifying only data attributes, resulting in a model that is little different from a logical data model. If you need to create a logical data model, then do so, following AM's practice, *Apply the Right Artifact(s)* (Chapter 17). However, if you want to create a UML class diagram, then you should consider the whole picture and identify responsibilities.

88. Indicate Visibility Only on Design Models

The visibility of an operation or attribute defines the level of access that objects have to it, and the UML supports four types of visibility that are summarized in Table 3. Visibility is an important design issue. On detailed design models, you should always indicate the visibility of attributes and operations, an issue that typically is not pertinent to domain or conceptual models. Visibility on an analysis or domain model will always be public (+), and so there is little value in indicating this.

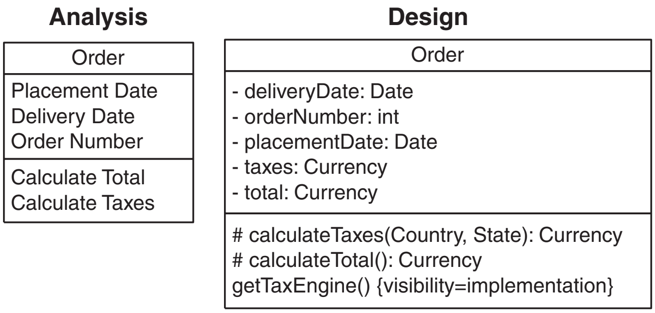

Figure 18. Analysis and design versions of a class.

89. Indicate Language-Dependent Visibility with Property Strings

If your implementation language includes non-UML-supported visibilities, such as C++'s implementation visibility, then a property string should be used, as you can see in Figure 18.

90. Indicate Types on Analysis Models Only When the Type Is an Actual Requirement

Sometimes the specific type of an attribute is a requirement. For example, your organization may have a standard definition for customer numbers that requires that they be nine-digit numbers. Perhaps existing systems, such as a legacy database or a predefined data feed, constrain some data elements to a specific type or size. If this is the case, you should indicate this information on your domain class model(s).

91. Be Consistent with Attribute Names and Types

It would not be consistent for an attribute named *customerNumber* to be a string, although it would make sense for it to be an integer. However, it would be consistent for the name *customerID* to be a string or an integer.

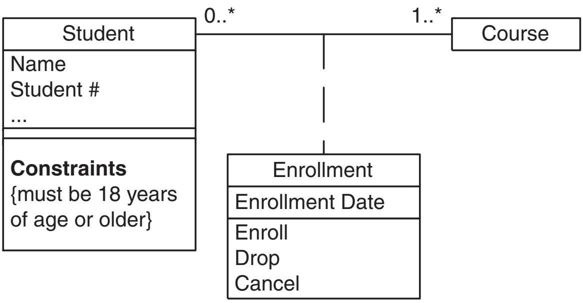

Figure 19. Modeling association classes.

92. Model Association Classes on Analysis Diagrams

Association classes, also called link classes, are used to model associations that have methods and attributes. Figure 19 shows that association classes are depicted as classes attached via dashed lines to associations—the association line, the class, and the dashed line are considered to be one symbol in the UML. Association classes typically are modeled during analysis and then refactored during design (either by hand or automatically by your CASE tool) because mainstream programming languages do not (yet) have native support for this concept.

93. Do Not Name Associations That Have Association Classes

The name of the association class should adequately describe it. Therefore, as you can see in Figure 19, the association does not need an additional adornment indicating its name.

94. Center the Dashed Line of an Association Class

The dashed line connecting the class to the association path should be clearly connected to the path and not to either class or to any adornments of the association, so that your meaning is clear. As you can see in Figure 19, the easiest way to accomplish this is to center the dashed line on the association path.

5.2 Class Style Guidelines

A class is effectively a template from which objects are created (instantiated). Classes define attributes, information that is pertinent to their instances, and operations—functionality that the objects support. Classes also realize interfaces (more on this later). Note that you may need to soften some of these naming guidelines to reflect your implementation language or any purchased or adopted software.

95. Use Common Terminology for Class Names

Class names should be based on commonly accepted terminology to make them easier for others to understand. For business classes, this would include names based on domain terminology such as *Customer*, *OrderItem*, and *Shipment* and, for technical classes, names based on technical terminology such as *MessageQueue*, *ErrorLogger*, and *PersistenceBroker*.

96. Prefer Complete Singular Nouns for Class Names

Names such as *Customer* and *PersistenceBroker* are preferable to *Cust* and *PBroker*, respectively, because they are more descriptive and thus easier to understand. Furthermore, it is common practice to name classes as singular nouns such as *Customer* instead of *Customers*. Even if you have a class that does represent several objects, such as an iterator (Gamma, Helm, Johnson, and Vlissides 1995) over a collection of customer objects, a name such as *CustomerIterator* would be appropriate.

97. Name Operations with Strong Verbs

Operations implement the functionality of an object; therefore, they should be named in a manner that effectively communicates that functionality. Table 4 lists operation names for analysis class diagrams as well as for design class diagrams—the assumption being that your implementation language follows Java naming conventions (Vermeulen et al. 2000)—indicating how the operation name has been improved in each case.

98. Name Attributes with Domain-Based Nouns

As with classes and operations, you should use full descriptions to name your attribute so that it is obvious what the attribute represents. Table 5 suggests a Java-based naming convention for analysis names that are in the *Attribute Name* format, although *attribute name* and *Attribute name* formats are also fine if applied consistently. Table 5 also suggests design names that take an *attributeName* format, although the *attribute_name* format is just as popular depending on your implementation language.

99. Do Not Model Scaffolding Code

Scaffolding code includes the attributes and operations required to implement basic functionality within your classes, such as the code required to implement relationships with other classes. Scaffolding code also includes getters and setters, also known as accessors and mutators, such as *getItem()* and *setItem()* in Figure 20, which get and set the value of attributes. You can simplify your class diagrams by assuming that scaffolding code will be created (many CASE tools can generate it for you automatically) and not model it. Figure 20 depicts the difference between the *OrderItem* class without scaffolding code and with it—including the constructor, the common static operation *findAllInstances()* that all business classes implement (in this system), and the attributes *item* and *order* and their corresponding getters and setters to maintain its relationships with the *Order* class and *Item* class, respectively.

Table 4. Example Names for Operations

Initial Name	Good Analysis Name	Good Design Name	Issue
Open Acc	Open Account	openAccount()	An abbreviation was replaced with the full word to make it clear what is meant.
Mailing Label Print	Print Mailing Label	printMailingLabel()	The verb was moved to the beginning of the name to make it active.
purchaseparkingpass()	Purchase Parking Pass	purchaseParkingPass()	Mixed case was applied to increase the readability of the design-level name.
Save the Object	Save	save()	The name was shortened because the term “TheObject” did not add any value.

Table 5. Example Names for Attributes

Initial Name	Good Analysis Name	Good Design Name	Issue
fName	First Name	firstName	Do not use abbreviations in attribute names.
firstname	First Name	firstName	Capitalizing the second word of the design name makes the attribute name easier to read.
personFirstName	First Name	firstName	This depends on the context of the attribute, but if this is an attribute of the “Person” class, then including “person” merely lengthens the name without providing any value.
nameLast	Last Name	lastName	The name “nameLast” was not consistent with “firstName” (and it sounded strange anyway).
hTTPConnection	HTTP Connection	httpConnection	The abbreviation for the design name should be in all lowercase.
firstNameString	First Name	firstName	Indicating the type of the attribute, in this case “string,” couples the attribute name to its type. If the type changes, perhaps because you decide to reimplement this attribute as an instance of the class “NameString,” then you will need to rename the attribute.
OrderItemCollection	Order Items	orderItems	The second version of the design name is shorter and easier to understand.

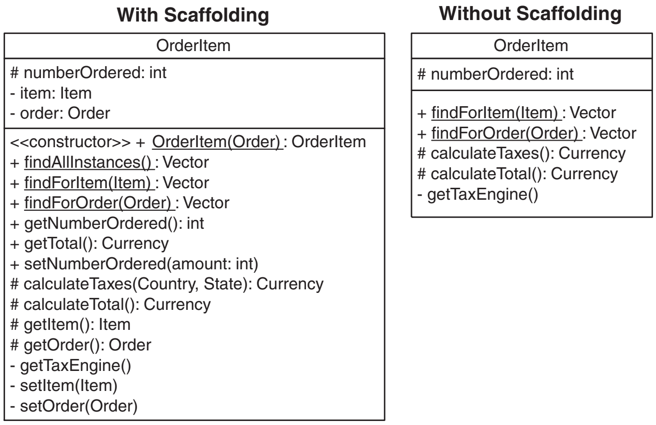
With Scaffolding
OrderItem
# numberOrdered: int
- item: Item
- order: Order
<<constructor>> + OrderItem(Order) : OrderItem
+ findAllInstances() : Vector
+ findForItem(Item) : Vector
+ findForOrder(Order) : Vector
+ getNumberOrdered(): int
+ getTotal(): Currency
+ setNumberOrdered(amount: int)
# calculateTaxes(Country, State): Currency
# calculateTotal(): Currency
# getItem(): Item
# getOrder(): Order
- getTaxEngine()
- setItem(Item)
- setOrder(Order)
Without Scaffolding
OrderItem
# numberOrdered: int
+ findForItem(Item) : Vector
+ findForOrder(Order) : Vector
# calculateTaxes(): Currency
# calculateTotal(): Currency
- getTaxEngine()

Figure 20. OrderItem class with and without scaffolding code.

100. Center Class Names

As you have seen in the figures in this chapter, it is common to center the names of classes.

101. Left-Justify Attribute Operation and Names

It is common practice, as you see in Figure 20 to left-justify both attribute and operation names within class boxes.

102. Do Not Model Keys

A key is a unique identifier of a data entity or table. Unless you are using a UML class diagram to model the logical or physical schema of a database (Ambler 2003), you should not model keys in your class. Keys are a data concept, not an object-oriented concept. A particularly common mistake of novice developers is to model foreign keys in classes, the data attributes needed to identify other rows of data within the database. A UML profile for physical data modeling is maintained at www.agiledata.org/essays/umlDataModelingProfile.html.

103. Never Show Classes with Just Two Compartments

It is allowable within the UML to have a class with one or more compartments. Although compartments may appear in any order, traditionally the topmost compartment indicates the name of the class and any information pertinent to the class as a whole (such as a stereotype); the second optional compartment typically lists the attributes; and the third optional compartment typically lists the operations. Other "nonstandard" compartments may be added to the class to provide information such as lists of exceptions thrown or notes pertaining to the class. Because naming conventions for attributes and operations are similar, and because people new to object development may confuse the two concepts, it isn't advisable to have classes with just two compartments (one for the class name and one listing either attributes or operations). If necessary, include a blank compartment as a placeholder, as you can see with the *Student* class in Figure 19.

104. Label Uncommon Class Compartments

If you do intend to include a class compartment that isn't one of the standard three—class name, attribute list, operations list—then include a descriptive label such as Exceptions, Examples, or Constraints centered at the top of the compartment, as you can see with the *Student* class in Figure 19.

105. Include an Ellipsis (. . .) at the End of an Incomplete List

You know that the list of attributes of the *Student* class of Figure 19 is incomplete because of the ellipsis at the end of the list. Without the ellipsis, there would be no indication that there is more to the class than what is currently shown (Evitts 2000).

106. List Static Operations/Attributes Before Instance Operations/Attributes

Static operations and attributes typically deal with early aspects of a class's life cycle, such as the creation of objects or finding existing instances of the classes. In other words, when you are working with a class you often start with statics. Therefore it makes sense to list them first in their appropriate compartments, as you can see in Figure 20 (statics are underlined).

107. List Operations/Attributes in Order of Decreasing Visibility

The greater the visibility of an operation or attribute, the greater the chance that someone else will be interested in it. For example, because public operations are accessible to a greater audience than protected operations, there is a likelihood that greater interest exists in public operations. Therefore, list your attributes and operations in order of decreasing visibility so that they appear in order of importance. As you can see in Figure 20, the operations and attributes of the *OrderItem* class are then listed alphabetically for each level of visibility.

108. For Parameters That Are Objects, List Only Their Types

As you can see in Figure 20, operation signatures can become quite long, extending the size of the class symbol. To save space, you can forgo listing the types of objects that are passed as parameters to operations. For example, Figure 20 lists *calculateTaxes(Country, State)* instead of *calculateTaxes(country: Country, state: State)*, thus saving space.

109. Develop Consistent Operation and Attribute Signatures

Operation names should be consistent with one another. For example, in Figure 20, all finder operations start with the text *find*. Parameter names should also be consistent with one another. For example, parameter names such as *theFirstName*, *firstName*, and *firstNm* are not consistent with one another, nor are *firstName*, *aPhoneNumber*, and *theStudentNumber*. Pick one naming style for your parameters and stick to it. Similarly, be consistent also in the order of parameters. For example, the methods *doSomething(securityToken, startDate)* and *doSomethingElse(studentNumber, securityToken)* could be made more consistent by always passing *securityToken* as either the first or the last parameter.

110. Avoid Stereotypes Implied by Language Naming Conventions

The UML allows for stereotypes to be applied to operations. In Figure 20. I applied the stereotype <<constructor>> to the operation *OrderItem(Order)*, but that information is redundant because the name of the operation implies that it's a constructor, at least if the implementation language is Java or C++. Furthermore, you should avoid stereotypes such as <<getter>> and <<setter>> for similar reasons—the names *getAttributeName()* and *setAttributeName()* indicate the type of operations you're dealing with.

111. Indicate Exceptions in an Operation's Property String

Some languages, such as Java, allow operations to throw exceptions to indicate that an error condition has occurred. Exceptions can be indicated with UML property strings, an example of which is shown in Figure 21.

```
+ findAllInstances(): Vector
{exceptions=NetworkFailure, DatabaseError}
```

Figure 21. Indicating the exceptions thrown by an operation.

5.3 Relationship Guidelines

For ease of discussion the term relationships will include all UML concepts such as associations, aggregation, composition, dependencies, inheritance, and realizations. In other words, if it's a line on a UML class diagram, we'll consider it a relationship.

112. Model Relationships Horizontally

With the exception of inheritance, the common convention is to depict relationships horizontally. The more consistent you are in the manner in which you render your diagrams, the easier it will be to read them. In Figure 22, you can see that the dependencies are modeled horizontally, although the fulfilled via association is not. This sometimes happens.

113. Draw Qualifier Rectangles Smaller than Classes

Qualifiers, in this case *itemNumber* in Figure 22, are considered part of the association and not the classifier itself in the UML (even though *itemNumber* would likely be an attribute of *Item*). If possible the qualifier rectangle should be smaller than the class it is attached to (Object Management Group 2004).

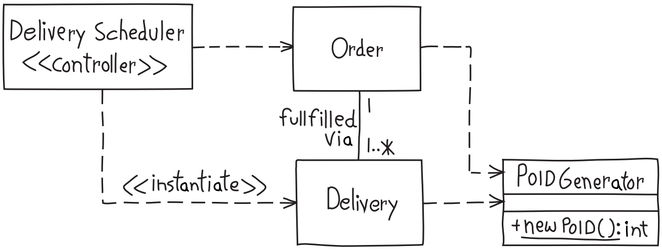

Figure 22. Shipping an order.

114. Model Collaboration Between Two Elements Only When They Have a Relationship

You need to have some sort of relationship between two model elements to enable them to collaborate. Furthermore, if two model elements do not collaborate with one another, then there is no need for a relationship between them.

115. Model a Dependency When the Relationship Is Transitory

Transitory relationships—relationships that are not persistent—occur when one or more of the items involved in a relationship is either itself transitory or a class. In Figure 22, you can see that there is a dependency between *DeliveryScheduler* and *Order*. *DeliveryScheduler* is a transitory class, one that does not need to persist in your database, and therefore, there is no need for any relationship between the scheduler and the order objects with which it interacts to persist. For the same reason, the relationship between *DeliveryScheduler* and *Delivery* is also a dependency, even though *DeliveryScheduler* creates *Delivery* objects.

In Figure 22, instances of *Delivery* interact with *OIDGenerator* to obtain a new integer value that acts as an object identifier (OID) to be used as a primary key value in a relational database. You know that *Delivery* objects are interacting with *OIDGenerator* and are not an instance of it because the operation is static. Therefore, there is no permanent relationship to be recorded, and so, a dependency is sufficient.

116. Tree-Route Similar Relationships to a Common Class

In Figure 22, you can see that both *Delivery* and *Order* have a dependency on *OIDGenerator*. Note how the two dependencies are drawn in combination in “tree configuration,” instead of as two separate lines, to reduce clutter in the diagram

(Evitts 2000). You can take this approach with any type of relationship. It is quite common with inheritance hierarchies (as you can see in Figure 25), as long as the relationship ends that you are combining are identical. For example, in Figure 23, you can see that *OrderItem* is involved in two separate relationships. Unfortunately, the *multiplicities* are different for each: one is 1..* and the other 0..*, so you can't combine the two into a tree structure. Had they been the same, you could have combined them, even though one relationship is aggregation and the other is association.

Note that there is a danger that you may be motivated to retain a relationship in order to preserve the tree arrangement, when you really should change it.

117. Always Indicate the Multiplicity

For each class involved in a relationship, there will always be a multiplicity. When the multiplicity is one and one only—for example, with aggregation and composition, it is often common for the part to be involved only with one whole—many modelers will not model the "1" beside the diamond. I believe that this is a mistake, and as you can see in Figure 23, I indicate the multiplicity in this case. If the multiplicity is "1," then indicate it as such so that your readers know that you've considered the multiplicity. Table 6 summarizes the multiplicity indicators that you will see on UML class diagrams.

118. Avoid a Multiplicity of ""*

You should avoid the use of "*" to indicate multiplicity on a UML class diagram because your reader can never be sure if you really mean "0..*" or "1..*" Although the UML specification (Object Management Group 2004) clearly states that "*" implies "0..*," the reality is that you simply can't trust that everyone has read and memorized the several-hundred-page specification.

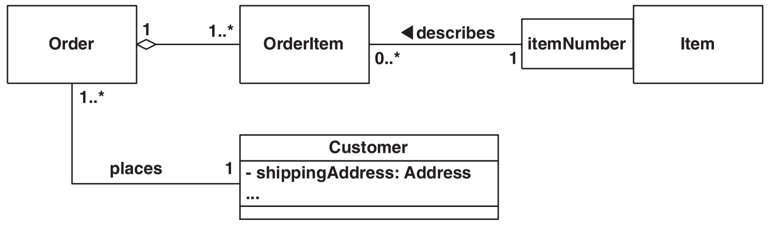

Figure 23. Modeling an order.

Table 6. UML Multiplicity Indicators

Indicator	Meaning
0..1	Zero or one
1	One only
0..*	Zero or more
1..*	One or more
n	Only *n* (where *n* > 1)
*	Many
0..*n*	Zero to *n* (where n > 1)
1..*n*	One to *n* (where *n* > 1)
n..*m*	Where *n* and *m* both > 1
n..*	*n* or more, where *n* > 1

119. Replace Relationship Lines with Attribute Types

In Figure 23, you can see that *Customer* has a *shippingAddress* attribute of type *Address*—part of the scaffolding code to maintain the association between customer objects and address objects. This simplifies the diagram because it visually replaces a class box and association, although it contradicts the *Do Not Model Scaffolding Code* guideline. You will need to judge which guideline to follow, the critical issue being which one will best improve your diagram given your situation.

A good rule of thumb is that if your audience is familiar with the class, then show it as a type. For example, if *Address* is a new concept within your domain then show it as a class; if it's been around for a while then showing it as a type should work well.

120. Do Not Model Implied Relationships

In Figure 23 there is an implied association between *Item* and *Order*—items appear on orders—but it was not modeled. A mistake? No; the association is implied through *OrderItem*. Orders are made up of order items, which in turn are described by items. If you model this implied association, not only do you clutter your diagram, but also you run the risk that somebody will develop the additional code to maintain it. If you don't intend to maintain the actual relationship—for example, you aren't going to write the scaffolding code—then don't model it.

121. Do Not Model Every Dependency

Model a dependency between classes only if doing so adds to the communication value of your diagram. As always, you should strive to follow AM's (Chapter 17) practice, *Depict Models Simply*.

5.4 Association Guidelines

122. Center Names on Associations

It is common convention to center the name of an association above an association path, as you can see in Figure 23 with the *describes* association between *Order* and *Item*, or beside the path, as with the *fulfilled via* association between *Order* and *Delivery* in Figure 22.

123. Write Concise Association Names in Active Voice

The name of an association, which is optional although highly recommended, is typically one or two descriptive words. If you find that an association name is wordy, think about it from the other direction; for example, the *places* name of Figure 23 is concise when read from right to left but would be wordy if written from the left-to-right perspective (e.g., "is placed by"). Furthermore, *places* is written in active voice instead of passive voice, making it clearer to the reader

124. Indicate Directionality to Clarify an Association Name

When it isn't clear in which direction the name of an association should be read, you can indicate the direction with a filled triangle, as you can see in Figure 23 between *OrderItem* and

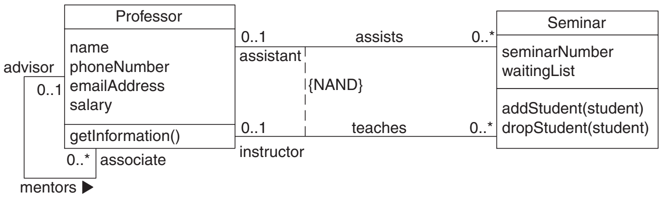

Figure 24. Professors and seminars.

Item. This marker indicates that the association should be read as “an item describes an order item” instead of “an order item describes an item.” It is also quite common to indicate the directionality on recursive associations, where the association starts and ends on the same class, such as *mentors* in Figure 24.

Better yet, when an association name isn’t clear, you should consider rewording it or maybe even renaming the classes.

125. Name Unidirectional Associations in the Same Direction

The reading direction of an association name should be the same as that of the unidirectional association. This is basically a consistency issue.

126. Word Association Names Left to Right

Because people in Western societies read from left to right, it is common practice to word association names so that they make sense when read from left to right. Had I followed this guideline with the *describes* association of Figure 23, I likely would not have needed to include the direction marker.

127. Indicate Role Names When Multiple Associations Between Two Classes Exist

Role names are optionally indicated on association ends to indicate how a class is involved in the association. Although the name of an association should make the roles of the two classes

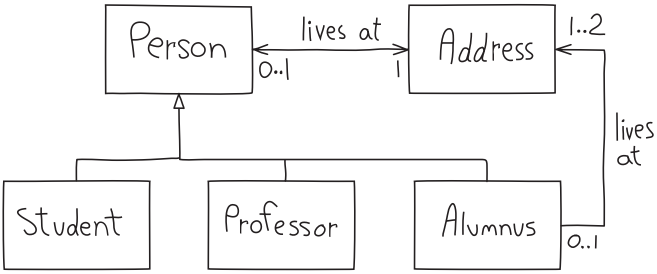

Figure 25. Modeling people in a university.

clear, it isn't always obvious when several associations exist between two classes. For example, in Figure 24, there are two associations between *Professor* and *Seminar*: *assists* and *teaches*. These two association names reflect common terminology at the university and cannot be changed. Therefore we opt to indicate the roles that professors play in each association to clarify them.

128. Indicate Role Names on Recursive Associations

Role names can be used to clarify recursive associations, ones that involve the same class on both ends, as you can see with the *mentors* association in Figure 24. The diagram clearly depicts the concept that an advisor mentors zero or more associate professors.

129. Make Associations Bidirectional Only When Collaboration Occurs in Both Directions

The *lives at* association between *Person* and *Address* in Figure 25 is unidirectional. A person object knows its address, but an address object does not know who lives at it. Within this domain, there is no requirement to traverse the association from *Address* to *Person*; therefore, the association does not need to be bidirectional (two-way). This reduces the code that needs to be written and tested within the address class because the scaffolding to maintain the association to *Person* isn't required.

130. Indicate Direction Only on Unidirectional Associations

In Figure 25 the *lives at* association between *Person* and *Address* is unidirectional; it can be traversed from *Person* to *Address* but not in the other direction. The arrowhead on the line indicates the direction in which the association can be traversed. In Figure 24 all of the associations are bidirectional, yet the associations do not include direction indicators—when an association does not indicate direction it is assumed that it is bidirectional. I could have indicated an arrow on each end but didn't because this information would have been superfluous.

131. Avoid Indicating Non-Navigability

The X on the *lives at* association in Figure 25 indicates that you cannot navigate the association from *Address* to *Person*. However, because the association indicates that you can navigate from *Person* to *Address* (that's what the arrowhead means) the assumption would have been that you cannot navigate in the other direction. Therefore the X is superfluous.

132. Redraw Inherited Associations Only When Something Changes

An interesting aspect of Figure 25 is the association between *Person* and *Address*. First, this association was pushed up to *Person* because *Professor*, *Student*, and *Alumnus* all had a *lives at* association with *Address*. Because associations are implemented by the combination of attributes and operations, both of which are inherited, the implication is that associations are inherited. If the nature of the association doesn't change—for example, both students and professors live at only one address—then we don't have any reason to redraw the association. However, because the association between *Alumnus* and *Address* is different, we have a requirement to track one or two addresses, and so we needed to redraw the association to reflect this.

133. Question Multiplicities Involving Minimums and Maximums

The problem with minimums and maximums is that they change over time. For example, today you may have a business rule that states that an alumnus has either one or two addresses that the university tracks, motivating you to model the multiplicity as 1..2, as depicted in Figure 25. However, if you build your system to reflect this rule, then when the rule changes you may find that you have significant rework to perform. In most object languages, it is easier to implement a 1..* multiplicity or, better yet, a 0..* multiplicity, because you don't have to check the size of the collection maintaining the association. Providing greater flexibility with less code seems good to me.

5.5 Inheritance Guidelines

Inheritance, also called generalization, models "is a" and "is like" relationships, enabling you to easily reuse existing data and code. When *A* inherits from *B*, we say that *A* is the subclass of *B* and that *B* is the superclass of *A*. Furthermore, we say that we have "pure inheritance" when *A* inherits all of the attributes and methods of *B*. The UML modeling notation for inheritance is a line with a closed arrowhead pointing from the subclass to the superclass.

134. Apply the Sentence Rule for Inheritance

One of the following sentences should make sense: "A subclass IS A superclass" or "A subclass IS KIND OF A superclass." For example, it makes sense to say that a student is a person, but it does not make sense to say that a student is an address or is like an address, and so, the class *Student* likely should not inherit from *Address*—association is likely a better option, as you can see in Figure 25. If it does not make sense to say that "the subclass is a superclass" or at least "the subclass is kind of a superclass," then you are likely misapplying inheritance.

135. Place Subclasses Below Superclasses

It is common convention to place a subclass, such as *Student* in Figure 25, below its superclass—*Person* in this case. Evitts (2000) says it well: Inheritance goes up.

136. Beware of Data-Based Inheritance

If the only reason two classes inherit from each other is that they share common data attributes, then this indicates one of two things: either you have missed some common behavior (this is likely if the sentence rule applies) or you should have applied association instead.

137. A Subclass Should Inherit Everything

A subclass should inherit all of the attributes and operations of its superclass, and therefore all of its relationships as well—a concept called pure inheritance. The advantage of pure inheritance is that you only have to understand what a subclass inherits, and not what it does not inherit. Although this sounds trivial, in a deep class hierarchy it makes it a lot easier if you only need to understand what each class adds, and not what it takes away. Larman (2002) calls this the "100% rule." Note that this contradicts the *Redraw Inherited Associations Only When Something Changes* guideline, and you'll need to decide accordingly.

138. Differentiate Generalizations Between Associations

It is possible, albeit uncommon, for an association to inherit from another association. In the case you should use a different

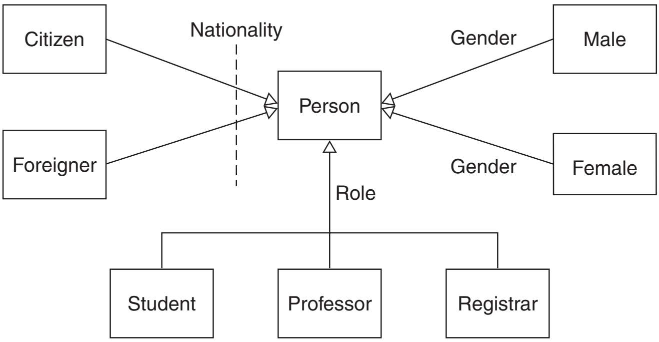

Figure 26. Indicating power types.

color or line weight for the generalization relationship to distinguish it (Object Management Group 2004).

139. Indicate Power Types on Shared Generalization

Power types, a meta modeling concept, can be indicated using a role name associated with a generalization relationship. Figure 26 shows three different ways to model power types. The preferred way was used to indicate that the *Student*, *Professor*, and *Registrar* classes are distinguished by the role that the person plays at the university, because it is concise. The same person may also be either a *Citizen* or a *Foreigner*, but not both. We know this because of the dashed line. The way that the *Gender* power type is indicated, on each of the generalization relationships, clutters the diagram.

5.6 Aggregation and Composition Guidelines

Sometimes an object is made up of other objects. For example, an airplane is made up of a fuselage, wings, engines, landing gear, flaps, and so on. A delivery shipment contains one or more packages. A team consists of two or more employees. These are all examples of the concept of aggregation, which

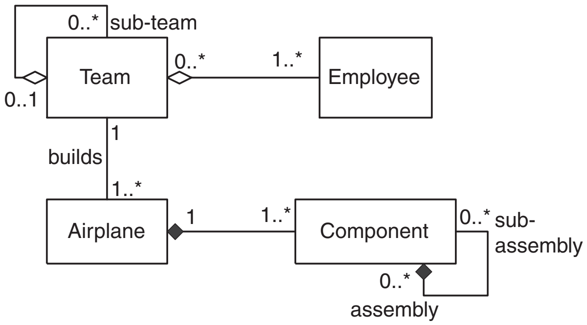

Figure 27. Examples of aggregation and composition.

represents "is part of" relationships. An engine is part of a plane, a package is part of a shipment, and an employee is part of a team. Aggregation is a specialization of association, specifying a whole–part relationship between two objects. Composition is a stronger form of aggregation, where the whole and the parts have coincident lifetimes, and it is very common for the whole to manage the life cycle of its parts. From a stylistic point of view, because aggregation and composition are both specializations of association, the guidelines for associations apply.

140. Apply the Sentence Rule for Aggregation

It should make sense to say "the part IS PART OF the whole." For example, in Figure 27, it makes sense to say that a course is part of a program or that a component is part of an airplane. However, it does not make sense to say that a team is part of an airplane or an airplane is part of a team—but it does make sense to say that a team builds airplanes, an indication that association is applicable.

141. Be Interested in Both the Whole and the Part

For aggregation and composition, you should be interested in both the whole and the part separately—both the whole and the part should exhibit behavior that is of value to your system. For example, you could model the fact that my watch has hands on it, but if this fact isn't pertinent to your system (perhaps you sell watches but not watch parts), then there is no value in modeling watch hands.

142. Place the Whole to the Left of the Part

It is a common convention to draw the whole, such as *Team* and *Airplane*, to the left of the part, such as *Employee* and *Component*, respectively.

143. Apply Composition to Aggregates of Physical Items

Composition is usually applicable whenever aggregation is *and* when both classes represent physical items. For example, in Figure 27 you can see that composition is used between *Airplane* and *Component*, whereas aggregation is used between *Team* and *Employee*—airplanes and components are both physical items, whereas teams are not.

144. Apply Composition When the Parts Share Their Persistence Life Cycle with the Whole

If the persistence life cycle of the parts is the same as that of the whole, if they're read in at the same time, if they're saved at the same time, if they're deleted at the same time, then composition is likely applicable.

145. Don't Worry About the Diamonds

When you are deciding whether to use aggregation or composition over association, Craig Larman (2002) says it best: "If in doubt, leave it out." The reality is that many modelers will agonize over when to use aggregation when the reality is that there is very little difference between association, aggregation, and composition at the coding level.

6. UML Package Diagrams

A UML package diagram depicts two or more packages and the dependencies between them. A package is a UML construct that enables you to organize model elements, such as use cases or classes, into groups. Packages are depicted as file folders and can be applied on any UML diagram, although any diagram that depicts only packages (and their interdependencies) is considered a package diagram. UML package diagrams are in fact new to UML 2, although they were informally part of UML 1—what we called package diagrams in the past were in fact UML class diagrams or UML use-case diagrams consisting only of packages. Create a package diagram to

- depict a high-level overview of your requirements,
- depict a high-level overview of your design,
- logically modularize a complex diagram,
- organize source code,
- model a framework (Evitts 2000).

6.1 Class Package Diagram Guidelines

146. Create Class Package Diagrams to Logically Organize Your Design

Figure 28 depicts a UML package diagram that organizes a collection of related classes. In addition to the package guidelines presented later in this chapter, apply the following heuristics to organize classes into packages:

- Classes of a framework belong in the same package.
- Classes in the same inheritance hierarchy typically belong in the same package.
- Classes related to one another via aggregation or composition often belong in the same package.
- Classes that collaborate with each other a lot often belong in the same package.

147. Create UML Component Diagrams to Physically Organize Your Design

If you have decided on a component-based approach to design, such as that promoted by Enterprise Java Beans (EJB) or Visual Basic, you should prefer a UML component diagram over a UML package diagram to depict your physical design. A version of Figure 28 as a UML component diagram is presented in Chapter 11 and, as you can see, that diagram is better suited to a physical design. Always remember to follow Agile Modeling's (Chapter 17) *Apply the Right Artifact(s)* practice.

148. Place Inheriting Packages Below Base Packages

Inheritance between packages is depicted in Figure 28 and, as you can see, the inheriting package is shown below the base package. This approach is consistent with other inheritance guidelines.

149. Vertically Layer Class Package Diagrams

Dependencies between packages indicate that the contents of the dependent package depend on, or have structural knowledge of, the contents of the other package. In Figure 28, the packages are placed on the diagram to reflect the logical layering of your architecture. The user interface interacts with

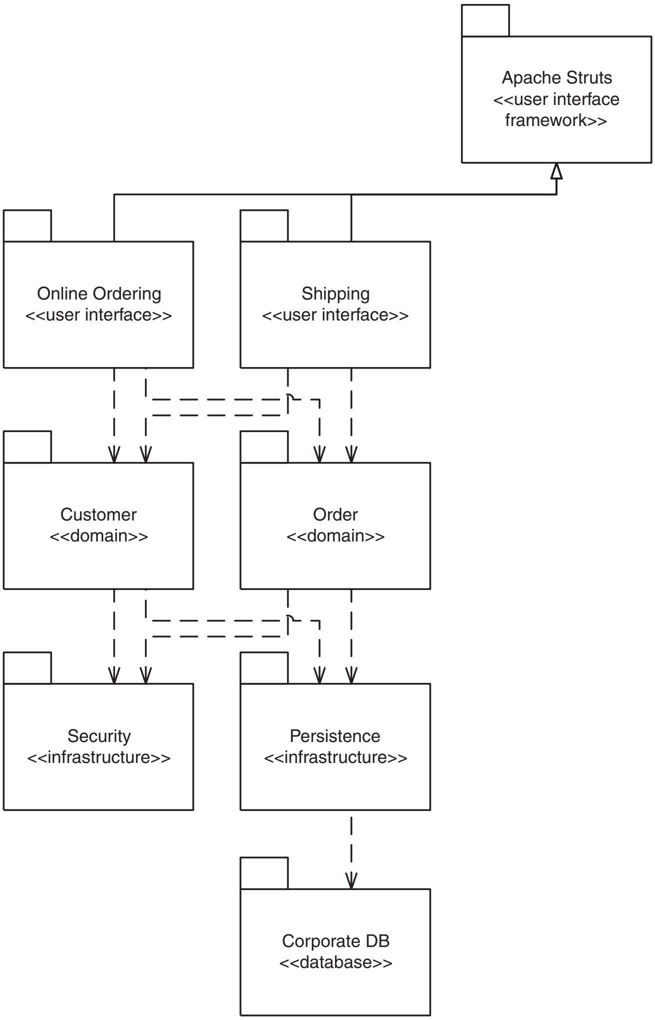

Figure 28. A package diagram organizing classes.

domain classes, which in turn use infrastructure classes, some of which access the database—which is traditionally depicted in a top-down manner.

6.2 Use-Case Package Diagram Guidelines

Use cases are a primary requirement artifact in many object-oriented development methodologies. This is particularly true of instantiations of the Unified Process (Rational Corporation 2002; Ambler, Nalbone, and Vizdos 2005). For larger projects, UML package diagrams are often created to organize these usage requirements.

150. Create Use-Case Package Diagrams to Organize Your Requirements

In addition to the package guidelines presented below, apply the following heuristics to organize UML use-case diagrams into package diagrams:

- Keep associated use cases together: included, extending, and inheriting use cases belong in the same package as the base/parent use case.
- Group use cases on the basis of the needs of the main actors. For example, in Figure 29, the *Enrollment* package contains use cases pertinent to enrolling students in seminars, a vital collection of services provided by the university.

151. Include Actors on Use-Case Package Diagrams

Including actors on UML package diagrams helps to put the packages in context, making diagrams easier to understand.

152. Arrange Use-Case Package Diagrams Horizontally

The primary audience of use-case package diagrams is project stakeholders; therefore, the organization of these diagrams

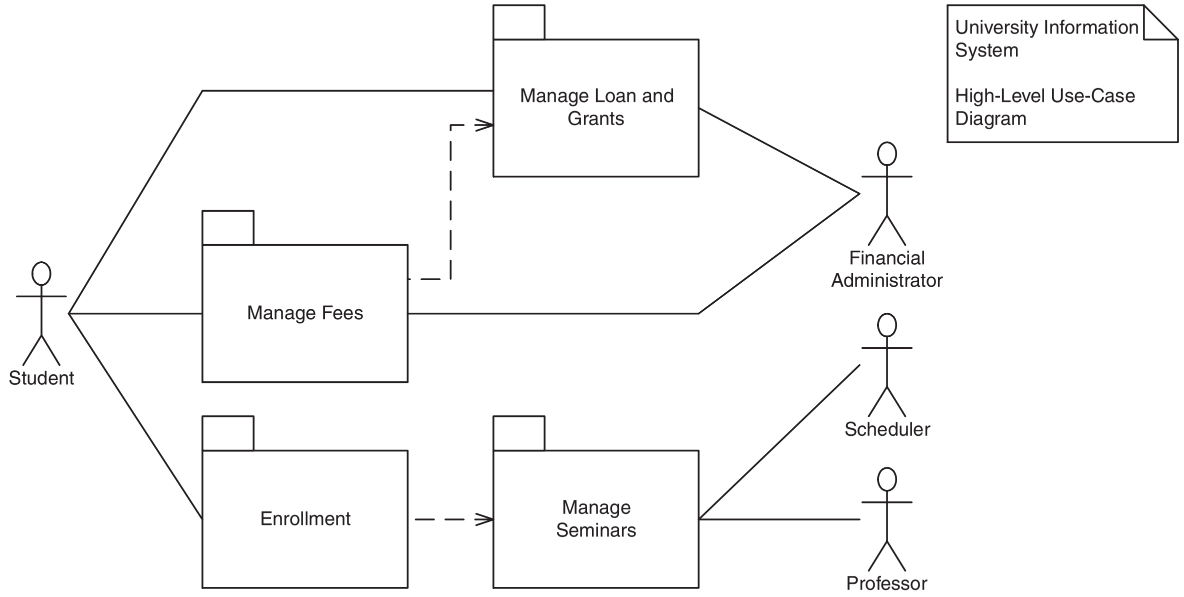

Figure 29. A UML package diagram that organizes use cases.

should reflect their needs. The packages in Figure 29 are arranged horizontally, with dependencies drawn from left to right to reflect the direction that people in Western cultures read.

6.3 Packages

The advice presented in this section is applicable to packages in any UML diagram, not just UML package diagrams.

153. Give Packages Simple, Descriptive Names

In both Figure 28 and Figure 29, the packages have simple, descriptive names, such as *Shipping*, *Customer*, *Enrollment*, and *Manage Student Loans and Grants*, which make it very clear what the package encapsulates.

154. Make Packages Cohesive

Anything that you put into a package should make sense when considered with the rest of the package contents. A good test to determine whether a package is cohesive is whether you can give your package a short, descriptive name. If you can't, then you likely have put several unrelated things into the package.

155. Indicate Architectural Layers with Stereotypes on Packages

It is very common to organize your design into architectural layers such as user interface, business/domain, persistence/data, and infrastructure/system. In Figure 28 you see that stereotypes such as `<<user interface>>`, `<<domain>>`, `<<infrastructure>>`, and `<<database>>` have been applied to packages.

156. Avoid Cyclic Dependencies Between Packages

Knoernschild (2002) advises that you avoid the situation in which package *A* is dependent on package *B* which is dependent on package *C* which in turn is dependent on package *A*—in this case, $A \rightarrow B \rightarrow C \rightarrow A$ forms a cycle. Because these packages are coupled to one another, they will be harder to test, maintain, and enhance over time. Cyclic dependencies are a good indicator that you need to refactor one or more packages, removing the elements from them that are causing the cyclic dependency.

157. Reflect Internal Relationships in Package Dependencies

When one package depends on another, it implies that there are one or more relationships between the contents of the two packages. For example, if it's a use-case package diagram, then there is likely an include, extend, or inheritance relationship between a use case in one package and one in the other package.

7. UML Sequence Diagrams

UML sequence diagrams are a dynamic modeling technique, as are UML communication diagrams. UML sequence diagrams are typically used to

- Validate and flesh out the logic and completeness of a usage scenario. A usage scenario is exactly what its name indicates—the description of a way that your system could be used. The logic of a usage scenario may be part of a use case, perhaps an alternate course; one entire pass through a use case, such as the logic described by the basic course of action or a portion of the basic course of action plus one or more alternate scenarios; or a pass through the logic contained in several use cases, such as when a student enrolls in the university and then immediately enrolls in three seminars.
- Explore your design because they provide a way for you to visually step through invocation of the operations defined by your classes.
- Give you a feel for which classes in your application are going to be complex, which in turn is an indication that you may need to draw state machine diagrams for those classes.
- Detect bottlenecks within an object-oriented design. By looking at what messages are being sent to an object, and

by looking at roughly how long it takes to run the invoked method, you quickly get an understanding of where you need to change your design to distribute the load within your system. Naturally, you will still want to gather telemetry data from a profiling tool to detect the exact locations of your bottlenecks.

7.1 General Guidelines

158. Strive for Left-to-Right Ordering of Messages

You start the message flow of a sequence diagram in the top left corner; a message that appears lower in the diagram is sent after one that appears above it. Because people in Western cultures read from left to right, you should strive to arrange the classifiers (actors, classes, objects, and use cases) across the top of your diagram in such a way as to depict message flow from left to right. In Figure 30, you can see that the classifiers have been arranged in exactly this way; had the *Seminar* object been to the left of the controller, this would not have been the case. Sometimes it isn't possible for all messages to flow from left to right; for example, it is common for pairs of objects to invoke operations on each other.

159. Layer Object Lifelines

Layering is a common approach to object-oriented design. It is quite common for systems to be organized into user interface, process/controller, business, persistence, and system layers (Ambler 2004). When systems are designed in this fashion, the lifelines (either classifiers or instances of classifiers) within each layer usually collaborate closely with one another and are relatively decoupled from the other layers. It makes sense to layer your sequence diagrams in a similar manner. One such layering approach is to start with the upper layers, such as your user interface, on the left-hand side and work through to the

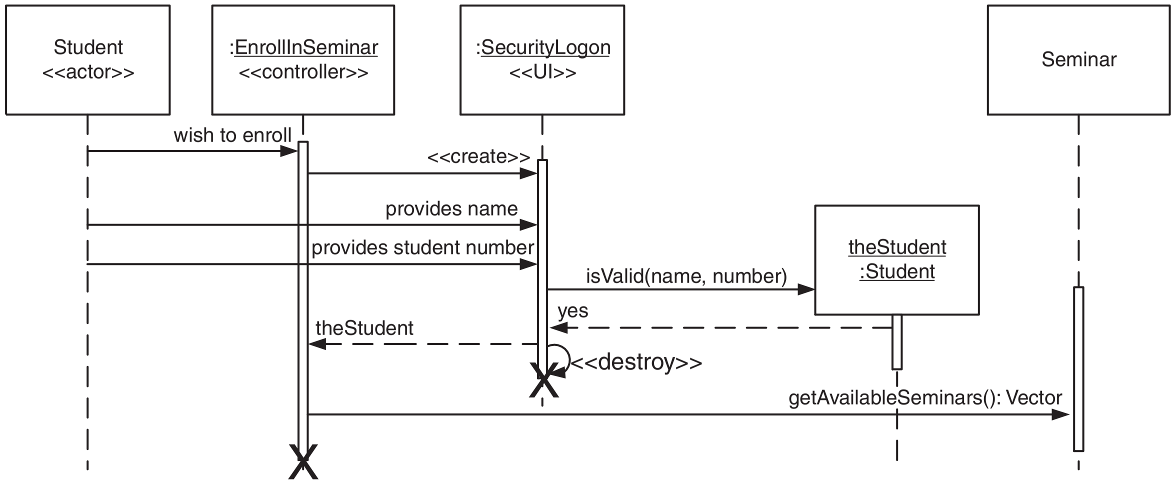

Figure 30. Enrolling a student in a seminar.

lower layers as you move to the right. Layering your sequence diagrams in this manner will often make them easier to read and will also make it easier to find layering logic problems. Figure 30 takes such an approach.

160. Give an Actor the Same Name as a Class, If Necessary

In Figure 30, you can see that there are an actor named *Student* and a class named *Student*. This is perfectly fine because the two classifiers represent two different concepts: the actor represents the student in the real world, whereas the class represents the student within the business application that you are building.

161. Include a Prose Description of the Logic

Figure 30 can be hard to follow, particularly for people not familiar with reading sequence diagrams, because it is very close to actual source code. It is quite common to include a business description of the logic you are modeling, particularly when the sequence diagram depicts a usage scenario, in the left-hand margin, as you can see in Figure 31. This increases the understandability of your diagram, and as Rosenberg and Scott (1999) point out, it also provides valuable traceability information between your use cases and sequence diagrams.

162. Place Proactive System Actors on the Leftmost Side of Your Diagram

Proactive system actors—actors that initiate interaction with yours—are often the focus of what you are modeling. For business applications the primary actor for most usage scenarios is a person or organization that initiates the scenario being modeled.

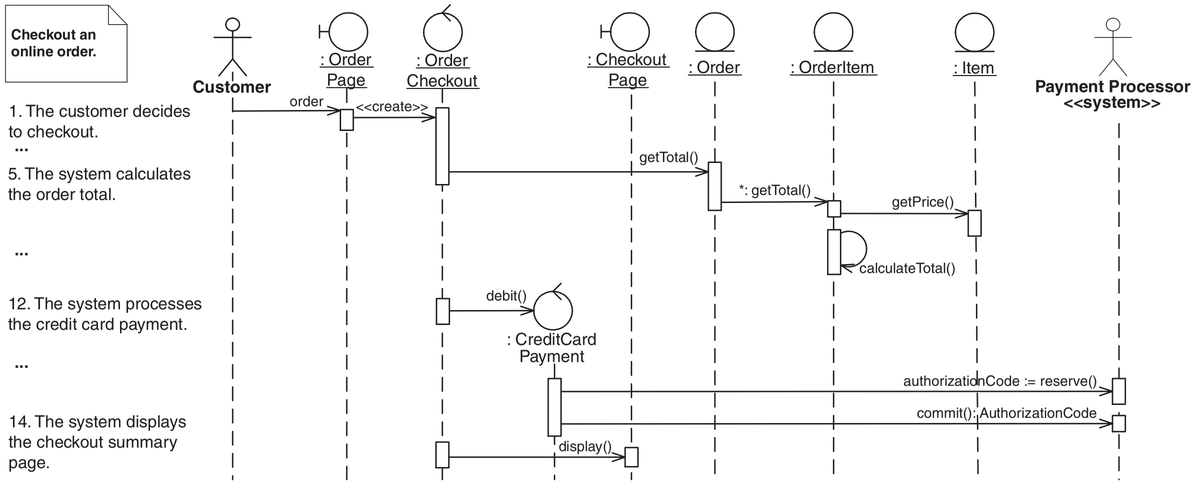
Checkout an online order.
1. The customer decides to checkout.
...
5. The system calculates the order total.
...
12. The system processes the credit card payment.
...
14. The system displays the checkout summary page.
Customer
: Order Page
: Order Checkout
: Checkout Page
: Order
: OrderItem
: Item
Payment Processor <<system>>
order
<<create>>
getTotal()
*: getTotal()
getPrice()
calculateTotal()
debit()
: CreditCard Payment
authorizationCode := reserve()
commit(): AuthorizationCode
display()

Figure 31. Checking out an online order.

163. Place Reactive System Actors on the Rightmost Side of Your Diagram

Reactive system actors—systems that initiates interaction with yours—should be placed on the rightmost side of your diagram because, for many business applications, these actors are treated as "back-end entities," that is, things that your system interacts with through access techniques such as C APIs, CORBA IDL, message queues, or Web services. In other words, put back-end systems at the back ends of your diagrams.

164. Avoid Modeling Object Destruction

Although memory management issues are important, in particular the removal of an object from memory at the right time, many modelers choose not to bother modeling object destruction on sequence diagrams (via an *X* at the bottom of an activation box or via a message with the `<<destroy>>` stereotype). Compare Figure 30 with Figure 31. Notice how object destruction introduces clutter into Figure 30 without any apparent benefit, yet Figure 31 gets along without indicating object destruction. Remember to follow Agile Modeling's (AM) (Chapter 17) practice *Depict Models Simply*.

Note that, in real-time systems, memory management is often such a critical issue that you may in fact decide to model object destruction.

165. Avoid Activation Boxes

Activation boxes are the little rectangles on the dashed lines hanging down from classifiers on UML sequence diagrams. Activation boxes are optional and are used to indicate focus of control, implying where and how much processing occurs. However, activation boxes are little better than visual noise because memory management issues are better left in the hands of programmers. Some modelers prefer the "continuous style" used in Figure 30, where the activation boxes remain until processing ends. Others prefer the "broken style" used in Figure 31. Both styles are fine. Choose one and move forward.

7.2 Guidelines for Lifelines

Lifelines are the classifiers or instances of classifiers that are depicted across the top of a sequence diagram. Actors, classes, components, objects, use cases, and so on are all considered lifelines. Note that naming conventions for classifiers are described elsewhere: in Chapter 4 for use cases, in Chapter 5 for classes and interfaces, and in Chapter 11 for components.

166. Name Objects Only When You Reference Them in Messages

Objects on sequence diagrams have labels in the standard UML format "name: ClassName," where "name" is optional (objects that have names are called named objects, whereas those without names are called anonymous objects). In Figure 30, the instance of *Student* was given the name *theStudent* because it is referred to as a return value to a message, whereas the instance of the *SecurityLogon* class did not need to be referenced anywhere else in the diagram and thus could be anonymous.

One tradeoff with this approach is that you will have some named objects on your diagrams and some anonymous ones, which some people will find confusing.

167. Name Objects When Several of the Same Type Exist

Whenever a sequence diagram includes several objects of the same type—for example, in Figure 32, you can see that there are two instances of the class *Account*—you should give all objects of that type a name to make your diagram unambiguous.

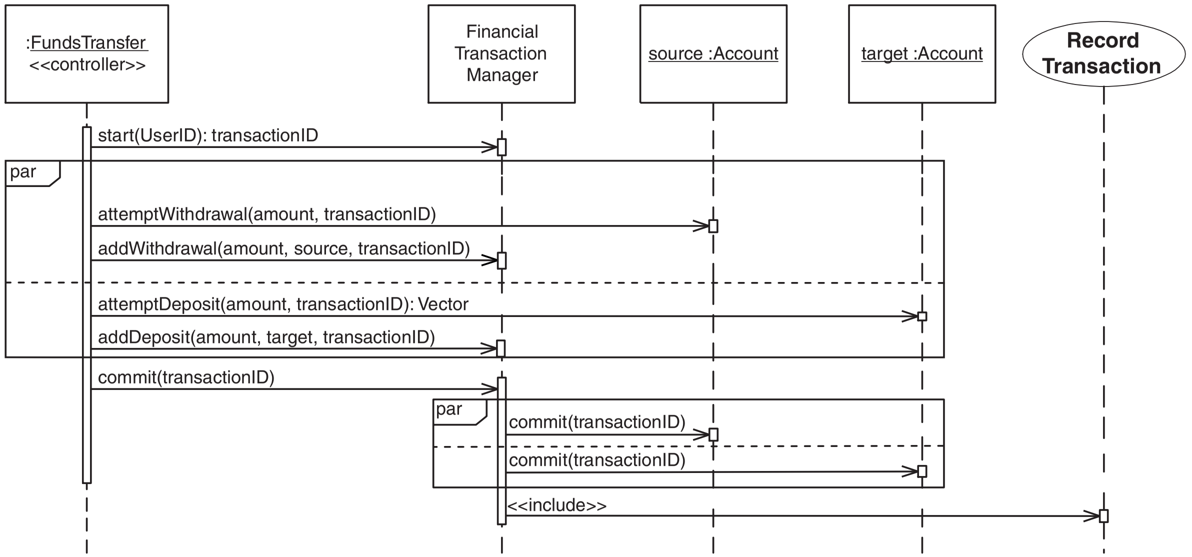
:FundsTransfer
<<controller>>
Financial Transaction Manager
source :Account
target :Account
Record Transaction
start(UserID): transactionID
par
attemptWithdrawal(amount, transactionID)
addWithdrawal(amount, source, transactionID)
attemptDeposit(amount, transactionID): Vector
addDeposit(amount, target, transactionID)
commit(transactionID)
par
commit(transactionID)
commit(transactionID)
<<include>>

Figure 32. Transferring funds between accounts.

Table 7. Common Stereotypes

Stereotype	**Usage**
ASP	During design to represent a Microsoft Active Server Page.
component	During design to indicate a component.
controller	To indicate a controller class that implements business logic pertaining to a usage scenario and/or logic that involves several business classes.
datastore	Persistent storage location for data, such as a relational database or a file system.
GUI	During design to represent a graphical user interface screen.
HTML	During design to represent an HTML page.
interface	During design to represent a Java interface.
JSP	During design to represent a Java Server Page.
report	During design to represent a printed or electronic report.
system	To indicate system actors.
user interface	For a generic user interface class, typically used on an analysis-level diagram where you haven't yet decided on an implementation platform.

168. Apply Textual Stereotypes to Lifelines Consistently

Table 7 summarizes common stereotypes that you may want to consider applying to lifelines. Don't invest a lot of time agonizing over which stereotypes you should use—for example `<<JSP>>` and `<<java server page>>` are both fine—just choose one and apply it consistently.

169. Focus on Critical Interactions

The AM practice *Create Simple Content* advises you to focus on the critical aspects of your system when you are creating a model and not to include extraneous details. Therefore, if your sequence diagram is exploring business logic, then you don't need to include the detailed interactions that your objects have with your database. Messages such as *save()* and *delete()* may be sufficient or, better yet, you could simply assume that persistence will happen appropriately and not even go to that level of detail. In Figure 31, there isn't any logic for reading orders and order items from the database or object cache. Nor do you see logic for the *CreditCardPayment* class to connect to the payment processor, but that surely must be happening in the background. By focusing only on the critical interactions—those that are pertinent to what you are modeling—you keep your diagrams as simple as possible but still get the job done, thus increasing your productivity as a modeler and very likely increasing the readability of your diagrams.

7.3 Message Guidelines

Note that naming conventions for operation signatures—guidelines that are pertinent to naming messages, parameters, and return values—are described in detail in Chapter 5.

170. Justify Message Names Beside the Arrowhead

Most modelers will justify message names, such as *calculate Total()* in Figure 31, so that they are aligned with the arrowheads. The general idea is that the receiver of the message will implement the corresponding operation, and so it makes sense that the message name is close to that classifier.

Notice that in Figure 32 this guideline was not followed. All of the message names are aligned so that they are beside the ends of the arrows, putting them close to the senders. The advantage of this approach is that it is very easy to see the logic of the scenario being modeled. The disadvantage is that it can be difficult to determine which operation is being invoked on the classifiers on the right-hand side of the diagram because you need to follow the lines across to the invocation boxes. As usual, pick one approach and apply it consistently.

171. Create Objects Directly

There are two common ways to indicate object creation on a sequence diagram: send a message with the `<<create>>` stereotype, as shown in Figure 31 with *OrderCheckout*, or directly show creation by dropping the classifier down in your diagram and invoking a message into its side, as you can see with *theStudent* in Figure 30 and *CreditCardPayment* in Figure 31. The primary advantage of the direct approach is that it visually communicates that the object doesn't exist until part way through the logic being modeled.

172. Apply Operation Signatures for Software Messages

Whenever a message is sent to a software-based classifier—such as a class, an interface, or a component—it is common convention to depict the message name using the syntax of your implementation language. For example, in Figure 32 the message *commit(transactionID)* is sent to the source account object.

173. Apply Prose to Messages Involving Human and Organization Actors

Whenever the source or target of a message is an actor representing a person or organization, the message is labeled with brief prose describing the information being communicated. For example, in Figure 30 the "messages" sent by the student actor are *provides name* and *provides student number*, descriptions of what the actual person is doing.

174. Prefer Names over Types for Parameters

Notice that, in Figure 32, for most message parameters the names of parameters and not their types[3] are shown, the only exception being the *UserID* parameter being passed in the *start()* message. The enables you to identify exactly what value is being passed in the message, sometimes type information is not enough. For example, the message *addDeposit(amount, target, transactionID)* conveys more information than *add Deposit(Currency, Account, int)*. Type information for operations is better captured in UML class diagrams.

175. Indicate Types as Parameter Placeholders

Sometimes the exact information that is being passed as a parameter isn't pertinent to what you are modeling, although the fact that something is being passed is pertinent. In this case, indicate the type of the parameter, as you can see in *start(UserID)* in Figure 32.

176. Apply the `<<include>>` *Stereotype for Use Case Invocations*

Figure 32 shows how a use case may be invoked in a sequence diagram, via a message with the `<<include>>` stereotype, a handy trick when you're modeling a usage scenario that includes a step in which a use case is directly invoked.

7.4 Guidelines for Return Values

177. Do Not Model Obvious Return Values

Return values are optionally indicated using dashed arrows with labels indicating the return values. For example, in

[3] This diagram follows Java naming conventions, where the names of types (classes and interfaces) start with uppercase letters, whereas the names of parameters start with lowercase letters.

Figure 30 the return value *theStudent* is indicated coming back from the *SecurityLogon* class as the result of invoking a message, whereas in Figure 31 no return value is indicated as the result of sending the message *getTotal()* to the order. In the first case, it isn't obvious that the act of creating a security logon object will result in the generation of a student object, whereas the return value of asking an order for its total is obvious.

178. Model a Return Value Only When You Need to Refer to It Elsewhere in a Diagram

If you need to refer to a return value elsewhere in your sequence diagram, often as a parameter passed in another message, indicate the return value in your diagram to explicitly show where it comes from.

179. Justify Return Values Beside the Arrowheads

Most modelers will justify return values, such as *yes* and *theStudent* in Figure 30, so that they are aligned with the arrowhead. The general idea is that the receiver of the return value will use it for something, and so it makes sense that the return value is close to the receiver.

180. Model Return Values as Part of a Method Invocation

Instead of cluttering your diagram with dashed lines, consider indicating the return value in the message name instead, using the notation *returnValue := message(parameters)* that you can see applied in Figure 31 with the *authorizationCode* := *reserve()* message. With this approach, you have only the single message line instead of a message line and a return-value line.

181. Indicate Types as Return-Value Placeholders

Sometimes the exact information that is being returned isn't pertinent to what you are modeling, although the fact that something is being returned is important. In this case, indicate the type of the return value, as you can see in *commit(): AuthorizationCode* in Figure 31.

182. Indicate the Actual Value for Simple Return Values

In Figure 30 the value *yes* is returned in response to the *isValid()* message, making it very clear that the student name and number combination was valid. Had the return value been named *Boolean*, thus indicating the type of answer, or *eligibilityIndicator*, thus indicating the name of the return value, it would not have been as clear.