Overview of UML

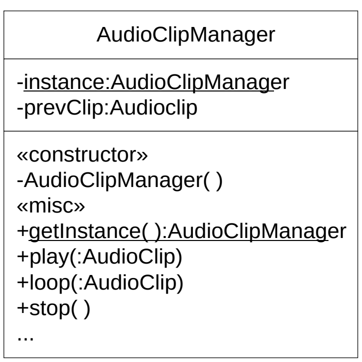
UML is a notation that you can use for object oriented analysis and design. UML stands for Unified Modeling Language. This chapter contains a brief overview of UML that will introduce you to the subset of UML and the extensions to UML used in this book. For a complete description of UML see http://www.rational.com/uml/documentation.html.

Books that are specifically about UML call the pieces of information stored in instances of a class attributes; they call a class' encapsulations of behavior operations. Those terms, like UML, are not specific to any implementation language. This book is not language neutral. It assumes that you are using Java as your implementation language. This book also uses Java specific terms in most places, rather than terms that are language neutral but less familiar to Java programmers. For example, it uses the words attribute and variable interchangeably, preferring the Java specific term variable. It uses the words operation and method interchangeably, preferring the Java specific term method.

UML defines a number of different kinds of diagrams. The kinds of diagrams that this book uses are Class Diagrams, Collaboration diagrams and Statechart Diagrams. The rest of this chapter is organized into sections that describe each of those kinds of diagrams and the elements that appear in them.

Class Diagram

A class diagram is a diagram that shows classes, interfaces and their relationships. The most basic element of a class diagram is a class. Below is an example of a class that shows many of the features that a class can have in a class diagram



Basic Class

Classes are drawn as rectangles. The rectangles can be divided into two or three compartments. The class rectangle shown above has three compartments. The top compartment contains the name of the class. The middle compartment lists the class' variables. The bottom compartment lists the class's methods.

The symbols that precede each variable and method are *visibility indicators*. The possible visibility indicators and their meanings are

+	public
#	protected
-	private

The variables in the middle compartment are shown as *visibilityIndicator name* : *type*

Therefore, the two variables shown in the class are private variables. The name of the first variable is `instance` and its type is `AudioClipManager`. The name of the second variable is `prevClip` and it type is `AudioClip`.

Though not shown above, an initial value can be indicated for a variable by following the variable's type with an equals sign and the value like this:

```
ShutDown:boolean = false
```

You will notice that the first variable shown in the class is underlined. If a variable is underlined that means that it is a static variable. That applies to methods too. Underlined methods are static methods.

The methods in the bottom compartment are shown as

visibilityIndicator name **(** *formalParameters* **)** : *returnType*

The `getInstance` method shown in the class above returns an `AudioClipManager` object.

UML indicates a void method by leaving out the "`:` *`returnType`*" from a method to indicate that it doesn't return anything. Therefore, the `stop` method shown in the class above does not return any result.

A method's formal parameters consist of a name and a type like this:

```
setLength(length:int)
```

If a method has multiple parameters, commas like this separate them like this:

```
setPosition(x:int, y:int)
```

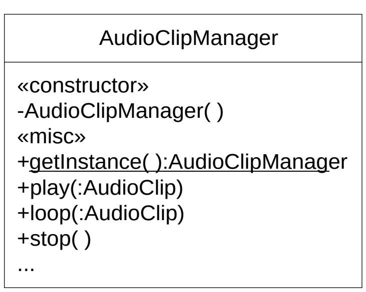
Two of the methods in the above class are preceded by a word in guillemets, like this:

```
«constructor»
```

In a UML drawing a word in guillemets is called a stereotype. A stereotype is used like an adjective to modify what comes after it. The `constructor` stereotype indicates that the methods that follow it are constructors. The `misc` stereotype indicates that the methods that come after it are regular methods. Additional uses for stereotypes are described later in this chapter.

One last element that appears in the class above is an ellipsis (…). If an ellipsis appears in the bottom compartment of a class that means that the class has additional methods that the diagram does not show. If an ellipsis appears in the middle compartment of a class, that means that the class as additional variables that the diagram does not show.

Often, it is not necessary or helpful to show as many details of a class as were shown in the above class. A class may be drawn with only two compartments:

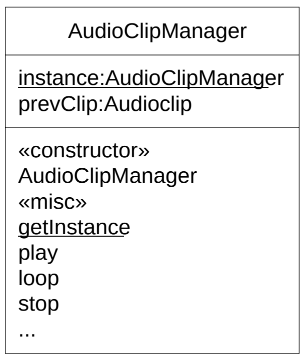


Two Compartment Class

When a class is drawn like the above class with only two compartments, its top compartment contains its name and its bottom compartment shows its methods. If a class is drawn with only two compartments, that just means that its variables are not shown. It does not mean that it has no variables.

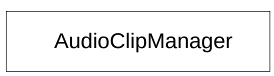
The visibility indicators may be omitted from methods and variables. When a method or variable is shown without a visibility indicator, that only means there is no indication of the method’s or variable’s visibility. It does not imply that the methods or variables are public, protected or private.

A method’s parameters may be omitted if their return values are also omitted. For example, the visibility indicators and method parameters are omitted from the following class:



Simplified Class

The simplest form of a class is has just one compartment that contains the class name, like this:



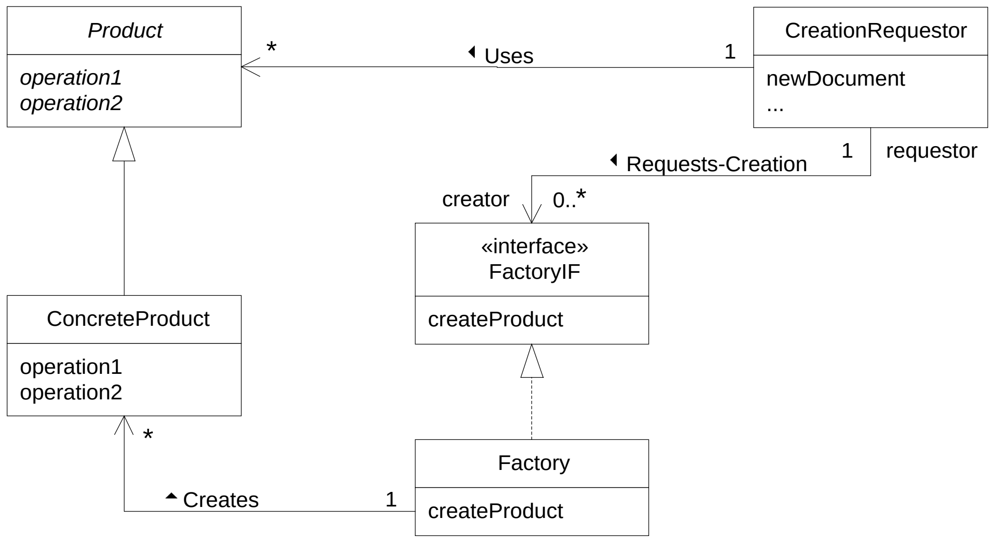
One Compartment Class

A one compartment representation of a class merely identifies the class. It provides no indication about what variables or methods the class has.

Interfaces are drawn in a manner similar to classes. The difference is that the name in the top compartment is preceded by an `interface` stereotype. Below is an example of an interface.

Interface

Classes and interfaces are important elements of class diagrams. The other elements of a class diagram show the relationships between classes and interfaces. Below is a typical class diagram.



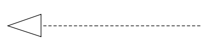
Class Diagram

The lines in the above diagram indicate the relationship between the classes and interface. A solid line with a closed head like the one below indicates the relationship of a subclass that inherits from a superclass.

Inherits From Superclass

The above class diagram shows the abstract class `Product` as the superclass of the `ConcreteProduct` class. You can tell that it is abstract because its name is italicized. You can tell that its methods are abstract because they are also italicized.

A similar sort of line is used to indicate that a class implements an interface. It is a dotted or dashed line with a closed head like the one below.



Implements an Interface

The above class diagram shows that the `Factory` class implements the `FactoryIF` interface.

The other lines show the other types of relationships between the classes and interface. UML calls these other types of relationships *associations*. There are a number of things that can appear with an association that provide information about the nature of an association. The following items are optional but this book consistently uses them wherever it makes sense.

- Association Name
  Somewhere around the middle of an association there may be an association name. The name of an association is always capitalized. There may be a triangle at one end of the association name. The triangle suggests in which direction you should read the association. It makes more explicit the information provided by the arrowheads at the ends of an association.

  Looking at the above class diagram, you will see that the association between the `Factory` and `ConcreteProduct` classes has the name `Creates`.

- Navigation Arrows
  Arrowheads that may appear at the ends of an association are called *navigation arrows*. Navigation arrows indicate the direction in which you may navigate an association.

  Looking at the association named `Creates` in the above class diagram, you will see that it has a navigation arrow pointing from the `Factory` class to the `ConcreteProduct` class. Because of the nature of creation, it seems clear that means the `Factory` class is responsible for creating instances of the `ConcreteProduct` class.

  The nature of some associations is less obvious. To make the nature of such associations clear, it may be necessary to supply additional information about the association. One common way to clarify the nature of an association is to name the role that each class plays in the association.

- Role Name
  To clarify the nature of an association, the name of the role that each class plays in the association can appear at each end of an association, next to the corresponding class. Role names are always lower case. That makes them easier to distinguish from association names, which are always capitalized.

  In the preceding class diagram, the `CreationRequestor` class and the `FactoryIF` interface participate in an association named `Requests-Creation`. The `CreationRequestor` class participates in that association in a role called `requestor`. The `FactoryIF` interface participates in that association in a role called `creator`.

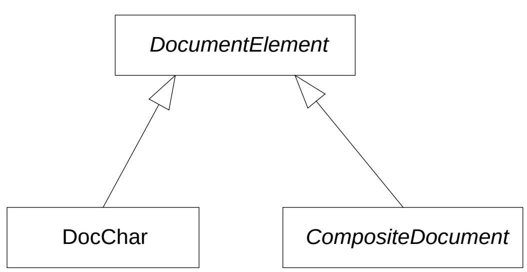
- Multiplicity Indicator
  Another detail of an association that is usually supplied is how many instances of each class participate in an occurrence of an association. A multiplicity indicator may appear at each end of an association to provide that information. A multiplicity indicator can be a simple number like 0 or 1. It can be a range of numbers indicated like this:

  ```
  0..2
  ```

  An asterisk as the high value of a range means an unlimited number of occurrences. The multiplicity indicator 1..* means at least one instance; 0..* means any number of instances. A simple * is equivalent to 0..*.

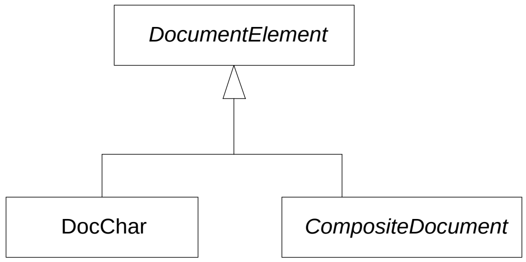
  Looking at the multiplicity indicators in the preceding drawing, you will see that each one of the associations in the drawing is a one-to-many relationship.

  Below is a class diagram that shows a class with multiple subclasses.



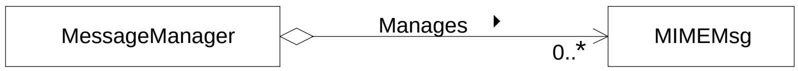
Multiple Inheritance Arrows

Though the above drawing is perfectly valid, UML allows a more aesthetically pleasing way to draw a class with multiple subclasses. The arrowheads may be combined as they are in the below diagram. The diagram below is identical in meaning to the diagram above.



Single Inheritance Arrow

Sometimes there is a need to convey more structure that is implied by a simple one-to-many relationship. The type of one-to-many relationship where one object contains a collection of other objects, is called an *aggregation*. A hollow diamond at the end of the association indicates aggregation. The hollow diamond appears at the end of the association attached to the class that contains instances of the other class. The class diagram below shows an aggregation.



Aggregation

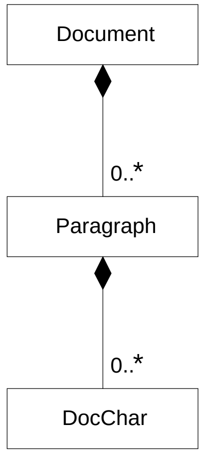
The above class diagram shows a class named `MessageManager`. Each of its instances contains zero or more instances of a class named `MIMEMsg`.

UML has another notation to indicate a stronger relationship than aggregation. That relationship is called composite aggregation. For an aggregation to be composite

- Aggregated instances must belong to only one composite at a time.

- Some operations must propagate from the composite to its aggregated instances. For example, when a composite object is cloned, its clone method will typically clone the aggregated instances so that the cloned composite will own clones of the original aggregated instances.

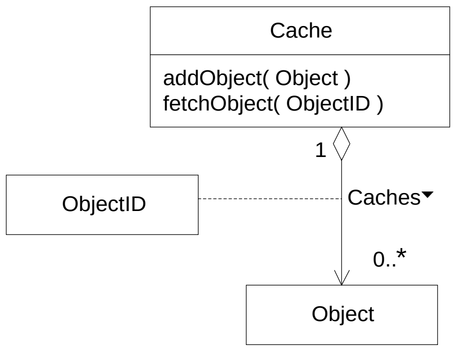
Below is a class diagram that contains a composite aggregation



Composite Aggregation

The above class diagram shows a `Document` class. `Document` objects can contain `Paragraph` objects. `Paragraph` objects can contain `DocChar` objects. Because of the composite aggregation, you know that `Paragraph` objects do not share `DocChar` objects and `Document` objects do not share `Paragraph` objects.

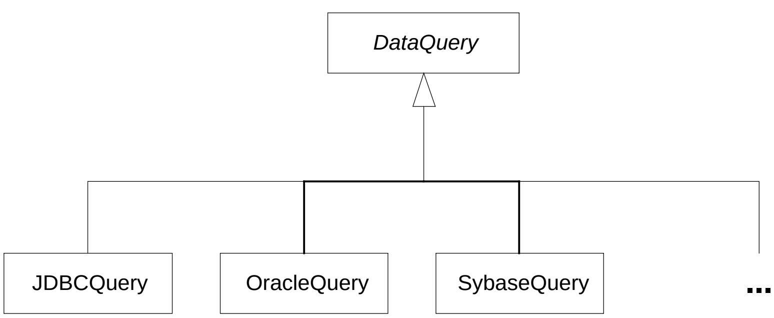
Some associations are indirect. Instead of classes being directly associated with each other, they are associated indirectly through a third class. Consider the following class diagram.



Association Class

The above association shows that instances of the `Cache` class refer to instances of the `Object` class through an instance of the `OjbectID` class.

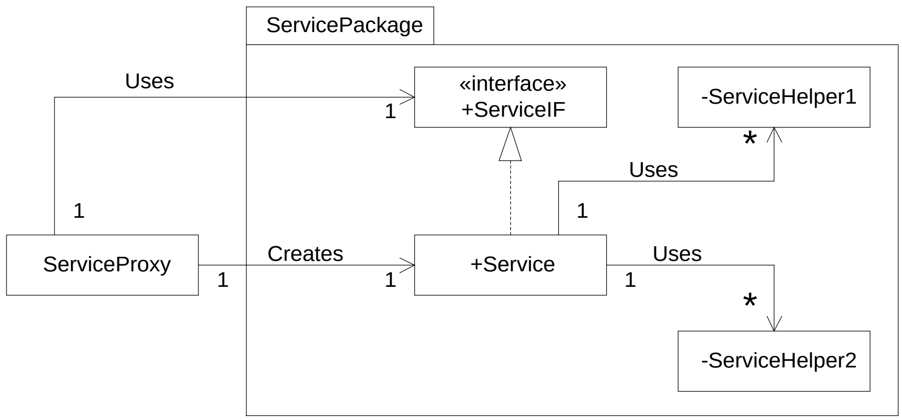
There is another use for ellipsis in a class diagram. Some class diagrams need to show that a class has a large or open ended set of subclasses, while only showing a few subclasses as examples of the sort of subclasses that the class has. The following class diagram shows how ellipsis can be used to show just that:



Open Ended Subclasses

The above class diagram shows a class named `DataQuery` that has subclasses named `JDBCQuery`, `OracleQuery`, `SybaseQuery` and an indefinite number of other classes that are indicated by the ellipsis.

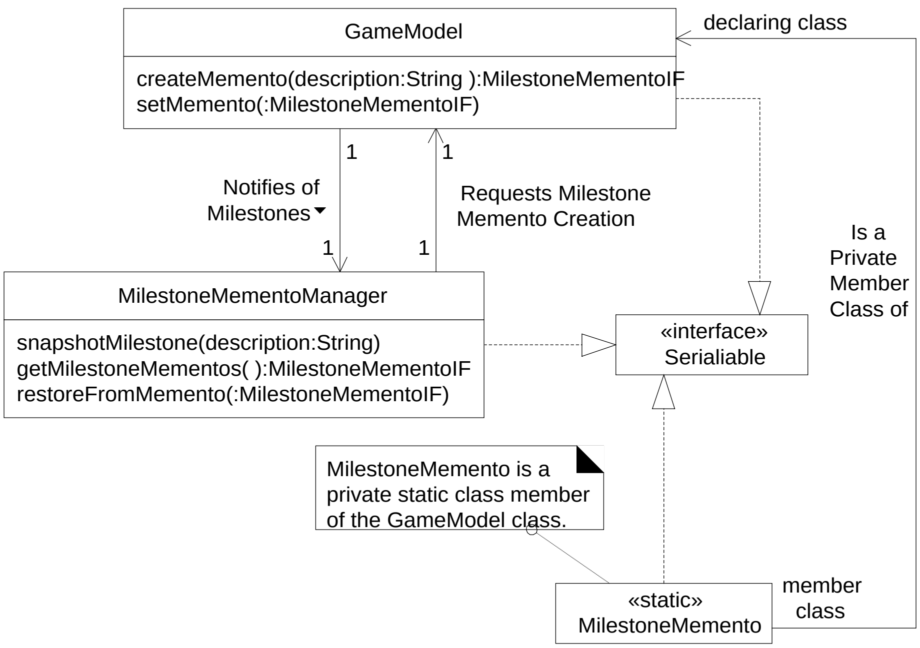
The classes in a class diagram can be organized into packages. Packages are drawn as a large rectangle with a small rectangle above the large rectangle. The small rectangle contains the name of the package. The small and large rectangles are arranged to have an overall shape similar to a manila folder. The class diagram below contains a package.



Package

The above diagram shows a package named `ServicePackage`. A visibility indicator can precede the name of classes and interfaces that appear within a package. Public classes are accessible to classes outside of the package; private classes are not.

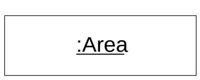
Sometimes there are aspects of a design that cannot be made sufficiently clear without a comment in a diagram. Comments in UML are drawn as a rectangle with its upper right corner turned down. Comments are attached to the diagram element the relate to by a dashed line. The class diagram below contains a comment.



Private Static Classes

The above class diagram shows the static class `MilestoneMemento` which is a private member of the `GameModel` class. There is no standard way in UML to represent a static private member class. The diagram uses a stereotype as an extension to UML to indicate that the `MilestoneMemento` class is static. It uses an association to indicate that the `MilestoneMemento` is a private member of the `GameModel` class. To make the relationship even more clear, there is a comment about it in the above class diagram.

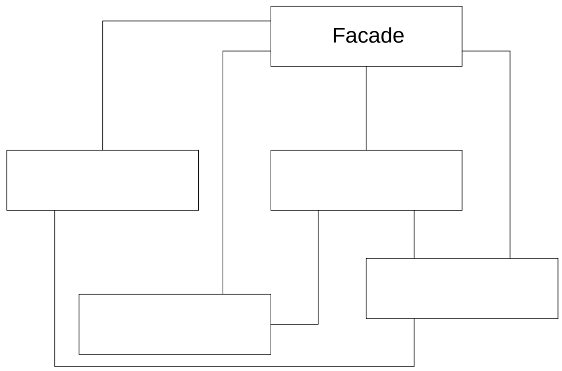
Class diagrams can include objects. Most of the objects in the diagrams in this book are drawn like this:



Object

The object shown above is an instance of a class named `Area`. The underline tells you that it is an object. A name may appear to the left of the colon(:). The only significance of the name is that it you can use it to identify the individual object.

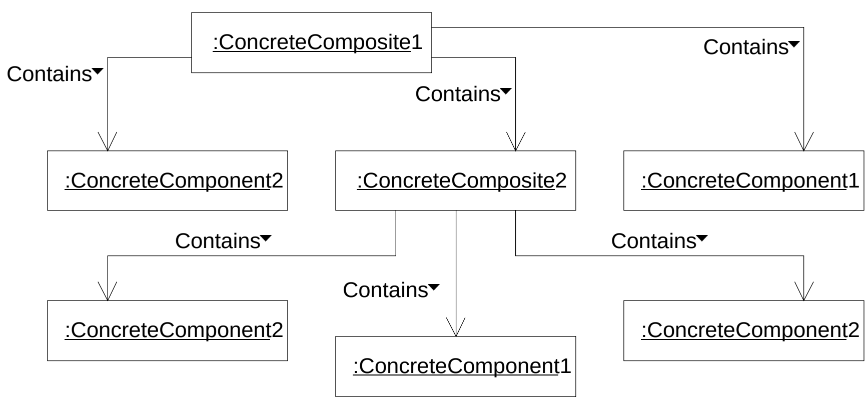
Some diagrams indicate an object as just an empty rectangle with nothing inside of the rectangle. Obviously, blank objects are cannot used to identify any particular kind of object. However, they can be used in a diagram that shows a structure in which of objects of unspecified type are connected. The class diagram below shows such a structure.



Blank Objects

The lines that connect two objects are not associations. The lines that connect objects are called links. Links are connections between objects, whereas associations are relationships between classes. A link is an occurrence of an association, just as an object is an instance of a class. Links can have association names, navigation arrows and most of the other embellishments that associations can have. However, since a link is a connection between two objects, links may not have multiplicity indicators or aggregation diamonds.

Some diagrams consist of just objects and links. Such diagrams are considered to be a kind of class diagram. However, there is a special name for that kind of diagram. A diagram that consists of just objects and links is called an object diagram. Below is an example of an object diagram.

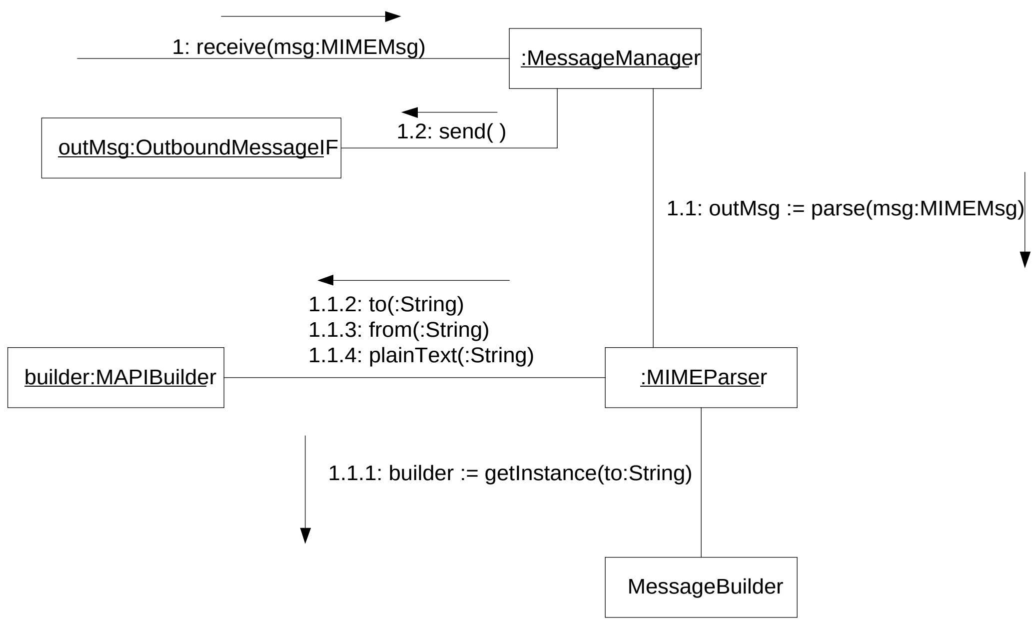


Object Diagram

Collaboration Diagram

Class and object diagrams show relationships between classes and objects. They also provide information about the interactions that occur between classes. They do not show the sequence in which the interactions occur or any concurrency that they may have.

Collaboration diagrams show objects, the links that connect them and the interactions that occur over each link. They also show the sequence and concurrency requirements of each interaction. Below is a simple example of a collaboration diagram



Collaboration Diagram

Any number of interactions can be associated with a link. Each interaction involves a method call. Next to each interaction or group of interactions is an arrow that points to the object whose method is called by the interaction. The entire set of objects and interactions shown in a collaboration diagram is collectively called a *collaboration*.

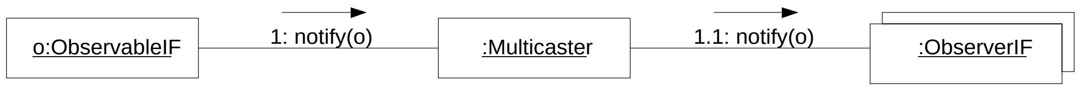
Each of the interactions shown in the above diagram begins with a sequence number and a colon. Sequence numbers indicate the order in which method calls occur. An interaction with the number 1 must come before an interaction with the number 2 and so on.

Multilevel sequence numbers consist of two or more numbers separated by a period. Notice that most of the sequence numbers in the above diagram are multilevel sequence numbers. Multilevel sequence numbers correspond to multiple levels of method calls. The portion of multilevel sequence number to the left of its rightmost period is called its prefix. For example, the prefix of 1.3.4 is 1.3.

Interactions numbered with a multilevel sequence number occur during another interaction's method call. The other method call is determined by the interaction's prefix. So the method calls of the interactions numbered 1.1 and 1.2 are made during the method call of interaction 1. Similarly, interactions numbered 1.1.1, 1.1.2, 1.1.3… occur during the method call of interaction 1.1.

Among interactions numbered with the same prefix, their methods are called in the order determined by the last number in their sequence numbers. Therefore, the methods of interactions numbered 1.1.1, 1.1.2, 1.1.3… are called in that order.

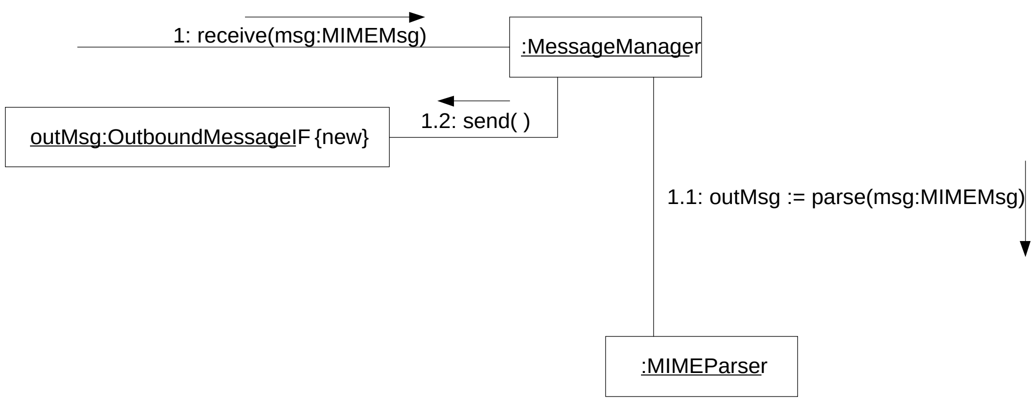
As mentioned earlier, links represent a connection between two objects. Because of that, links may not have any multiplicity indicators. That works well for links that represent an occurrence of an association between a definite number of objects. Associations that have a star multiplicity indicator on either end involve an indefinite number of objects. There is no way to draw an indefinite number of links to an indefinite number of objects. UML provides a symbol that allows us to draw links that connect an indefinite number of objects. That symbol is called a multiobject. It represents an indefinite number of objects. It looks like a rectangle behind a rectangle. The collaboration diagram below contains a multiobject.



Multiobject

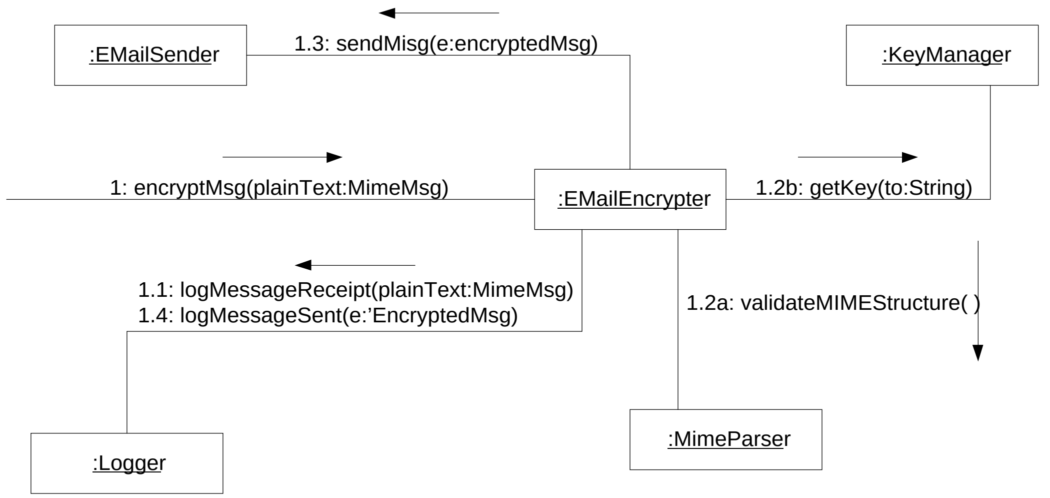
The above collaboration diagram shows an `ObservableIF` object calling a `Multicaster` object's `notify` method. The `Multicaster` object's implementation of `notify` method calls `notify` method of an indefinite number of `ObserverIF` objects linked to the `Multicaster` object.

Objects created as a result of a collaboration may be marked with the property `{new}`. Temporary objects that exist only during a collaboration may be marked with the property `{transient}`. The collaboration diagram below shows a collaboration that creates an object.



New Object

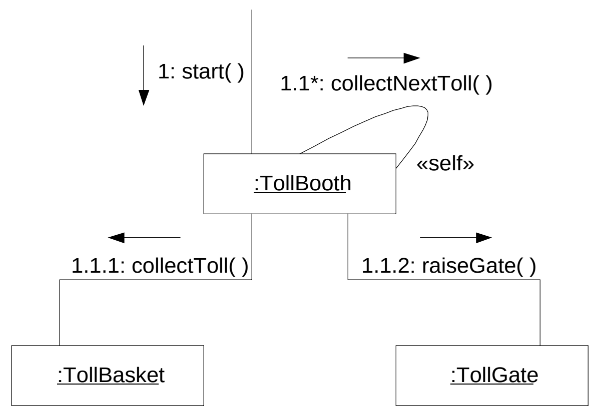
Some interactions occur concurrently, rather than sequentially. A letter at the end of a sequence number indicates concurrent interactions. For example, the methods of interactions numbered 2.2a and 2.2b would be called concurrently and each call would run in a separate thread. Consider the following collaboration diagram.



E-Mail Encrypter

In the above diagram, the top level interaction is the one numbered 1. During that interaction, first interaction 1.1 is invoked. Then interactions 1.2a and 1.2b are invoked at the same time. After that, interactions 1.3 and 1.4 are invoked, in that order.

An asterisk after a sequence number indicates a repeated interaction. Consider the following collaboration diagram:

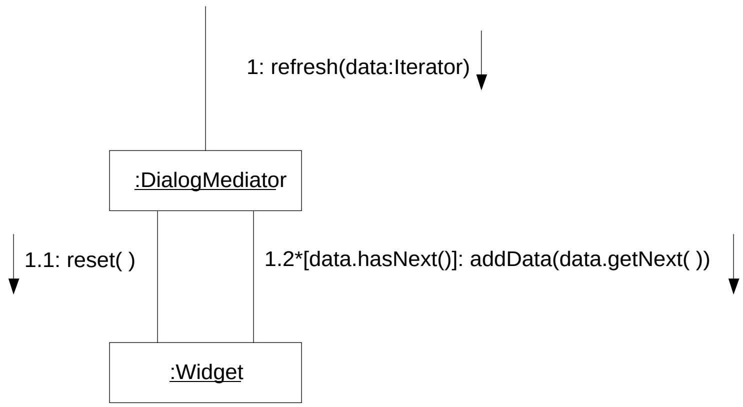


Toll Booth

The above collaboration begins by calling the `TollBooth` object's `start` method. That method repeatedly calls the object's `collectNextToll` method. The `collectNextToll` method repeatedly calls the `TollBasket` object's `collectToll` method and the `TollGate` object's `raiseGate` method.

One other thing to notice about the above collaboration diagram is the «`self`» stereotype that appears next to the link for interaction 1.1 It serves to clarify the fact that the link is a self reference.

Unlike the example in the above collaboration diagram, most repetitive interactions occur conditionally. UML allows a condition to be associated with a repetitive interaction by putting it after the asterisk inside of square brackets. The following collaboration diagram shows an example of a conditional repetitive interaction.



Refresh

The above collaboration diagram shows an `Iterator` object being passed to a `DialogMediator` object's `refresh` method. Its `refresh` method, in turn, calls a `Widget` object's `reset` method and then repeatedly calls its `addData` method while the `Iterator` object's `hasNext` method returns true.

It is important to note that the definition of UML does not define the meaning of conditions associated while repetitive interactions very precisely. In particular, the definition of UML says that what appear between the square brackets can "be expressed in pseudocode or an actual programming language." This book consistently uses Java for that purpose.

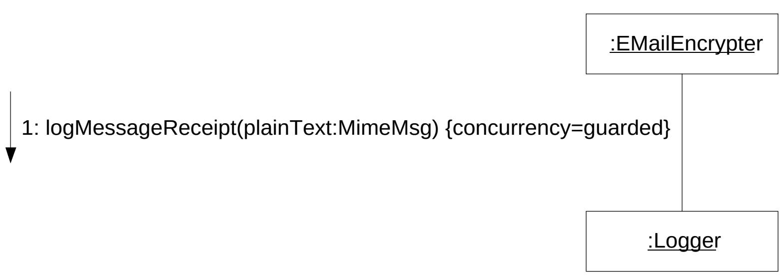
When dealing with multiple threads, something that often needs to be specified about methods is what happens when two threads try to call the same method at the same time. UML allows that to be specified by placing one of the following constructs after a method:

`{concurrency = sequential}`
: This means that only one thread at a time should call a method. No guarantee is made about the correctness of the method's behavior if the method is called by multiple threads at a time.

`{concurrency = concurrent}`
: This means that if multiple threads call a method at the same time they will all execute it concurrently and correctly.

`{concurrency = guarded}`
: This means that if multiple threads call a method at the same time, only one thread at a time will be allowed to execute the method. While one thread is executing the method, other threads will be forced to wait until it is their turn. The that is similar to the behavior of synchronized Java methods.

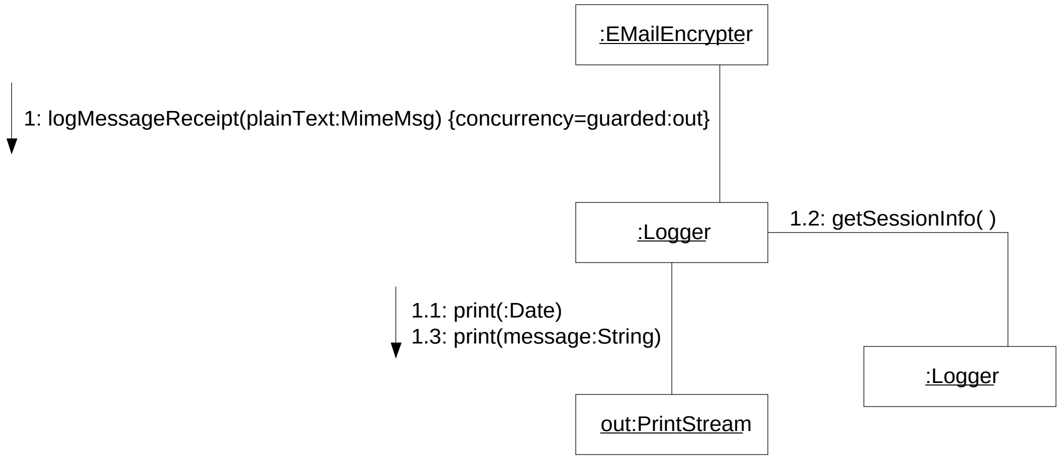
The following collaboration diagram shows an example of a synchronized method.



Synchronized Method Call

There are refinements to thread synchronization used in this book for which there is no standard representation in UML. This book uses some extensions to the `{concurrency=guarded}` construct to represent that those refinements.

Sometimes the object that threads need to synchronize on is not the same object whose method is called by

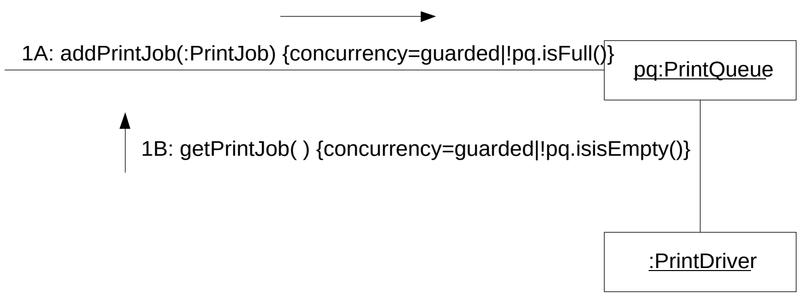


an interaction. Consider the following collaboration diagram.

Synchronization Using a Third Object

In the above diagram, `{concurrency=guarded:out}` refers to the object labeled `out`. Before the method call can actually take place the thread that controls the call must own the lock associated with the `out` object. That is identical to Java's semantics for a synchronized statement.

Sometimes there are preconditions beyond acquiring ownership of a lock that must be met before a thread may proceed with a method call. Such preconditions are indicated by vertical bar followed by the precondition. The collaboration diagram below show such preconditions following `guarded` and a vertical bar.



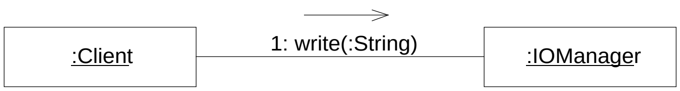
Print Queue

The above collaboration diagram shows two asynchronous interactions. One interaction calls a `PrintQueue` object's `addPrintJob` method to add a print job to the `PrintQueue` object. In the other interaction, a `PrintDriver` object calls the `PrintQueue` object's `getPrintJob` method to get a print job from the `PrintQueue` object. Both interactions have synchronization preconditions. If the print queue is full, then the interaction that calls the `addPrintJob` method will wait until the print queue is not full before proceeding to make the call to the `addPrintJob` method. If the print queue is empty, then the interaction that calls the `getPrintJob` method will wait until the print queue is not empty before proceeding to make the call to the `getPrintJob` method.

The mechanisms discussed above determine when the methods of a collaboration are called. They do not say anything about when method calls return. The arrows that point at the objects whose methods are called provide information about when the methods may return.

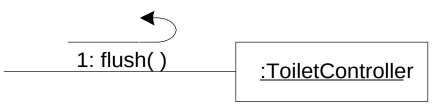
All the arrows in the above diagram have a closed head, which indicates that the calls are synchronous. The method calls do not return, until the method has completed doing whatever it does.

An open arrow head indicates an asynchronous method call. An asynchronous method call returns to its caller immediately, while the method does its work asynchronously in a separate thread. The collaboration diagram bellow shows an asynchronous method call.



Asynchronous Method Call

UML only defines arrowheads for synchronous and asynchronous calls. As extensions to UML, UML allows other types of arrows to indicate different types of method calls. To indicate a balking call, this book uses a bent back arrow, as shown in the diagram below.



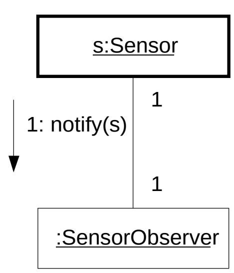
Balk

When a balking call is made to an object's method, if there is no other thread executing that object's method, then when the method is finished doing what it does, it returns. However, when a balking call is made and there is another thread currently executing that object's method, the method returns immediately without doing anything.

You may have noticed that the object that makes the top level call that initiates a collaboration is not shown in all of the above collaboration diagrams. That means that the object that initiates the collaboration is not considered to be a part of the collaboration.

The objects that you have seen how to model in UML, up to this point, are passive in nature. They don't do anything until one of their methods is called.

Some objects are active. The have a thread associated with them that allows the to initiate operations asynchronously and independently of whatever else is going on in a program. Active objects are indicated as a object with a thick border. The below diagram contains an example of an active object.

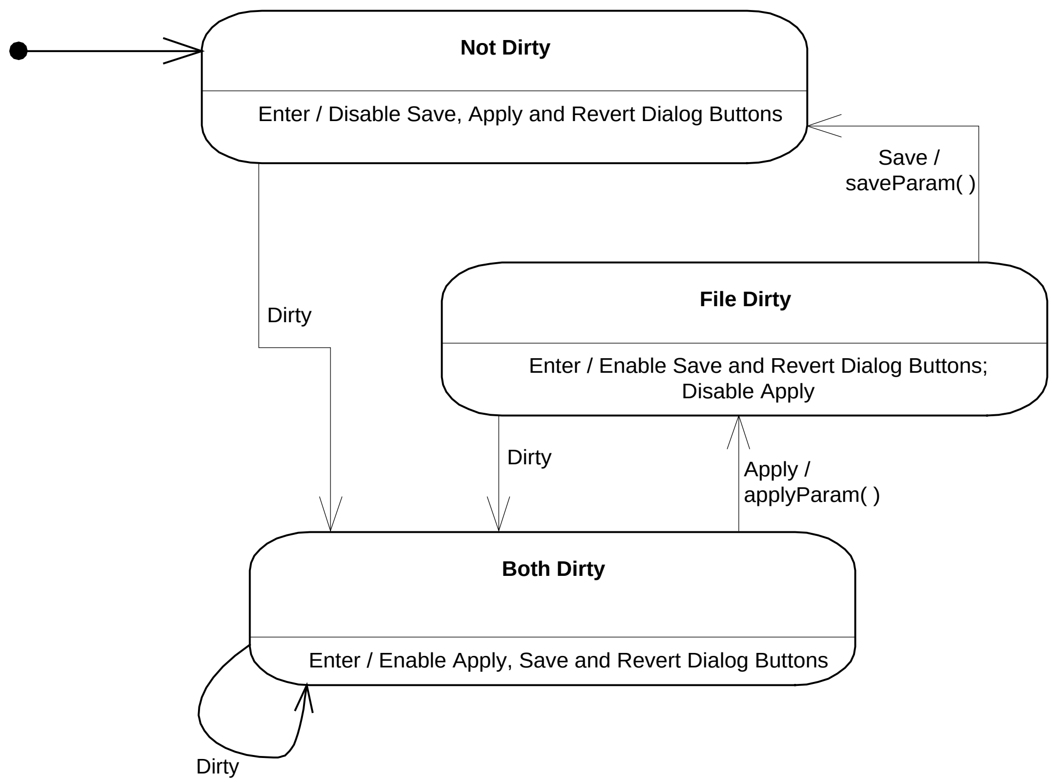


Active Sensor

The above diagram shows an active `Sensor` object that calls a `SensorObserver` object's method without another object first calling one of its methods.

Statechart Diagram

Statechart diagrams are used to model a class' behavior as a state machine. Below is an example of a simple state diagram.



Statechart Diagram

A statechart diagram shows each state as a rounded rectangle. All of the states in the above diagram are divided into two compartments. The upper compartment contains the name of the state. The lower compartment contains a list of events that the object responds to while in that state, without changing state. Each event in the list is followed by a slash and the action it performs in response to the event. UML predefines two such events:

- The `enter` event occurs when an object enters a state.

- The `exit` event occurs when an object leaves a state.

If there are no events that a state responds to without changing state, then its rectangle is not divided into two compartments. Such a state is drawn as a simple rounded rectangle that just contains the state's name.

Every state machine has an initial state that is the state an object is in before the first transition occurs. The initial state is drawn as a small solid filled-in circle.

Transitions between states are shown in statechart diagrams as lines between states. Normally, transition lines are required to have a label that indicates the event that triggers the transition. The event may be followed by a slash and the action that occurs when the transition takes place.

If a statechart includes a final state, the final state is drawn as a small solid circle inside of a larger circle.