I got all this from the research from the internet and it helped me create the files I needed.

Sequence diagrams are typically used to model:

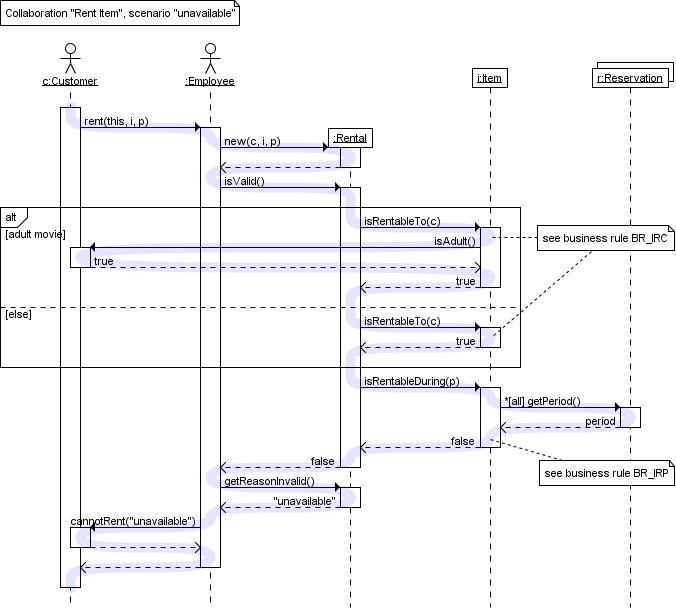
1. **Usage scenarios**. A usage scenario is a description of a potential way your system is used. The logic of a usage scenario may be part of a use case, perhaps an alternate course. It may also be one entire pass through a use case, such as the logic described by the basic course of action or a portion of the basic course of action, plus one or more alternate scenarios. The logic of a usage scenario may also be a pass through the logic contained in several use cases. For example, a student enrolls in the university, and then immediately enrolls in three seminars.
2. **The logic of methods**. Sequence diagrams can be used to explore the logic of a complex operation, function, or procedure. One way to think of sequence diagrams, particularly highly detailed diagrams, is as [visual object code](http://www.agilemodeling.com/artifacts/sequenceDiagram.htm#VisualCoding).
3. **The logic of services**. A service is effectively a high-level method, often one that can be invoked by a wide variety of clients. This includes web-services as well as business transactions implemented by a variety of technologies such as CICS/COBOL or CORBA-compliant object request brokers (ORBs).

- See more at: <http://www.agilemodeling.com/artifacts/sequenceDiagram.htm#sthash.bBZsoHQr.dpuf>

UML sequence diagrams are used to show how objects interact in a given situation. An important characteristic of a sequence diagram is that time passes from top to bottom : the interaction starts near the top of the diagram and ends at the bottom (i.e. **L**ower equals**L**ater).

A popular use for them is to document the dynamics in an object-oriented system. For each key collaboration, diagrams are created that show how objects interact in various representative scenarios for that collaboration.

The [**UML sequence diagram gallery**](http://www.tracemodeler.com/gallery/index.html) contains many examples, but here's a typical sequence diagram based on a system use case scenario:



The diagram above shows how objects interact in the "rent item" collaboration when the item is not available during the requested period.

To clarify how execution switches from one object to another, a blue highlight was added to represent the flow of control. Note that this highlight is not part of the diagram itself.

As with all UML diagrams, comments are shown in a rectangle with a folded-over corner :

UML sequence diagram with a comment.

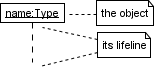
To relate the comment to whatever diagram elements it is about, connect them with dashed lines.

**Targets**

Objects as well as classes can be targets on a sequence diagram, which means that messages can be sent to them. A target is displayed as a rectangle with some text in it. Below the target, its lifeline extends for as long as the target exists. The lifeline is displayed as a vertical dashed line.

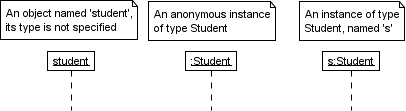
Object

The basic notation for an object is

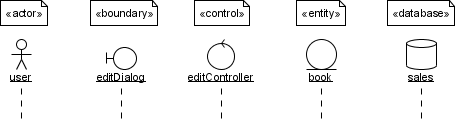


Where 'name' is the name of the object in the context of the diagram and 'Type' indicates the type of which the object is an instance. Note that the object doesn't have to be a *direct* instance of Type, a type of which it is an *indirect* instance is possible too. So 'Type' can be an abstract type as well.

Both name and type are optional, but at least one of them should be present. Some example :



As with any UML-element, you can add a stereotype to a target. Some often used stereotypes for objects are «actor», «boundary», «control», «entity» and «database». They can be displayed with icons as well :



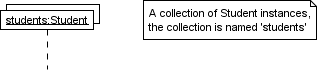
An object should be named only if at least one of the following applies

* You want to refer to it during the interaction as a message parameter or return value
* You don't mention its type
* There are other anonymous objects of the same type and giving them names is the only way to differentiate them

Try to avoid long but non-descriptive names when you're also specifying the type of the object (e.g. don't use 'aStudent' for an instance of type Student). A shorter name carries the same amount of information and doesn't clutter the diagram (e.g. use 's' instead).

MultiObject

When you want to show how a client interacts with the elements of a collection, you can use a multiobject. Its basic notation is



Again, a name and/or type can be specified. Note however that the 'Type' part designates the type *of the elements* and not the type of the collection itself.

Class

The basic notation for a class is

UML sequence diagram of a class.

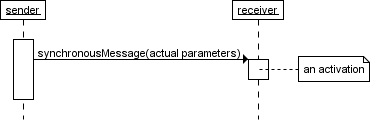
Only class messages (e.g. shared or static methods in some programming languages) can be sent to a class. Note that the text of a class is not underlined, which is how you can distinguish it from an object.

**Messages**

When a target sends a message to another target, it is shown as an arrow between their lifelines. The arrow originates at the sender and ends at the receiver. Near the arrow, the name and parameters of the message are shown.

Synchronous message

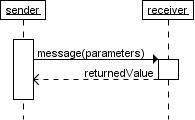
A synchronous message is used when the sender waits until the receiver has finished processing the message, only then does the caller continue (i.e. a blocking call). Most method calls in object-oriented programming languages are synchronous. A closed and filled arrowhead signifies that the message is sent synchronously.



The white rectangles on a lifeline are called activations and indicate that an object is responding to a message. It starts when the message is received and ends when the object is done handling the message.

When a messages are used to represent method calls, each activation corresponds to the period during which an activation record for its call is present on the call stack.

If you want to show that the receiver has finished processing the message and returns control to the sender, draw a dashed arrow from receiver to sender. Optionally, a value that the receiver returns to the sender can be placed near the return arrow.



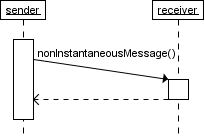
If you want your diagrams to be easy to read, only show the return arrow if a value is returned. Otherwise, hide it.

Instantaneous message

Messages are often considered to be instantaneous, i.e. the time it takes to arrive at the receiver is negligible. For example, an in-process method call. Such messages are drawn as a horizontal arrow.



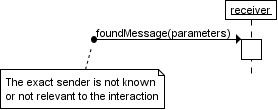
Sometimes however, it takes a considerable amount of time to reach the receiver (relatively speaking of course) . For example, a message across a network. Such a non-instantaneous message is drawn as a slanted arrow.



You should only use a slanted arrow if you really want to emphasize that a message travels over a relatively slow communication channel (and perhaps want to make a statement about the possible delay). Otherwise, stick with a horizontal arrow.

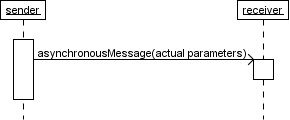
Found message

A found message is a message of which the caller is not shown. Depending on the context, this could mean that either the sender is not known, or that it is not important who the sender was. The arrow of a found message originates from a filled circle.



Asynchronous messages

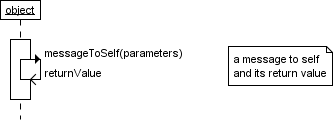
With an asynchronous message, the sender does not wait for the receiver to finish processing the message, it continues immediately. Messages sent to a receiver in another process or calls that start a new thread are examples of asynchronous messages. An open arrowhead is used to indicate that a message is sent asynchrously.



A small note on the use of asynchronous messages : once the message is received, both sender and receiver are working simultaneously. However, showing two simultaneous flows of control on one diagram is difficult. Usually authors only show one of them, or show one after the other.

Message to self

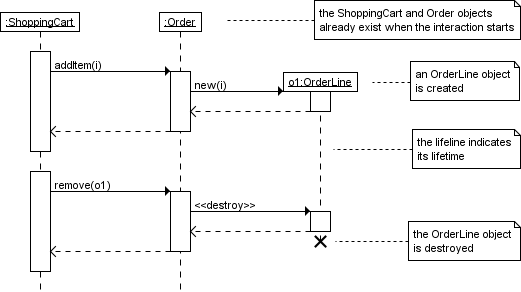
A message that an object sends itself can be shown as follows :



Keep in mind that the purpose of a sequence diagram is to show the interaction between objects, so think twice about every self message you put on a diagram.

Creation and destruction

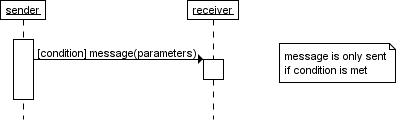
Targets that exist at the start of an interaction are placed at the top of the diagram. Any targets that are created during the interaction are placed further down the diagram, at their time of creation.



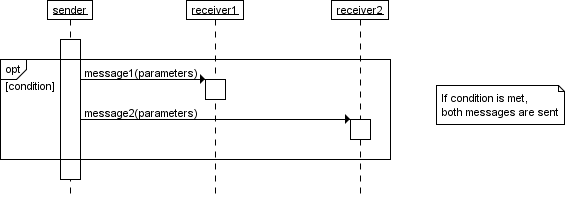
A target's lifeline extends as long as the target exists. If the target is destroyed during the interaction, the lifeline ends at that point in time with a big cross.

**Conditional interaction**

A message can include a guard, which signifies that the message is only sent if a certain condition is met. The guard is simply that condition between brackets.

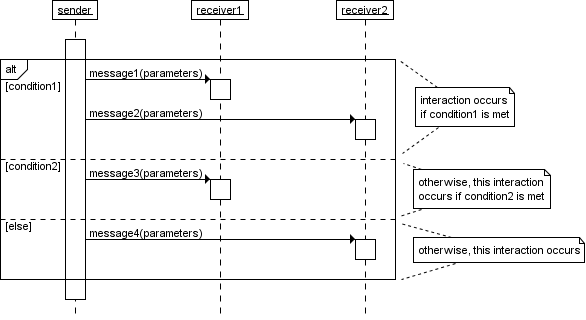


If you want to show that several messages are conditionally sent under the same guard, you'll have to use an 'opt' combined fragment. The combined fragment is shown as a large rectangle with an 'opt' operator plus a guard, and contains all the conditional messages under that guard.



A guarded message or 'opt' combined fragment is somewhat similar to the if-construct in a programming language.

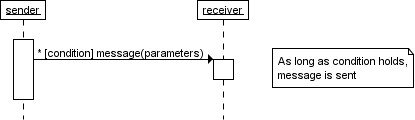
If you want to show several alternative interactions, use an 'alt' combined fragment. The combined fragment contains an operand for each alternative. Each alternative has a guard and contains the interaction that occurs when the condition for that guard is met.



At most one of the operands can occur. An 'alt' combined fragment is similar to nested if-then-else and switch/case constructs in programming languages.

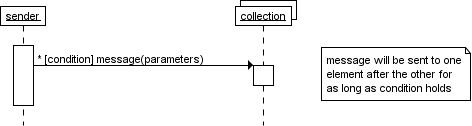
**Repeated interaction**

When a message is prefixed with an asterisk (the '\*'-symbol), it means that the message is sent repeatedly. A guard indicates the condition that determines whether or not the message should be sent (again). As long as the condition holds, the message is repeated.



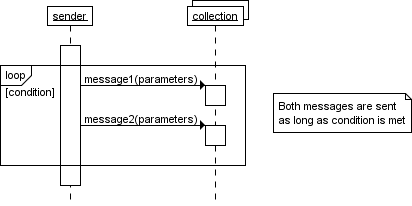
The above interaction of repeatedly sending the same message to the same object is not very useful, unless you need to document some kind of polling scenario.

A more common use of repetition is sending the same message to different elements in a collection. In such a scenario, the receiver of the repeated message is a multiobject and the guard indicates the condition that controls the repetition.



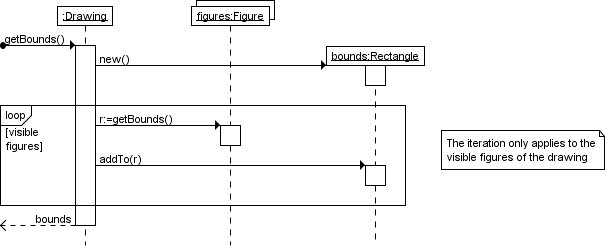
This corresponds to an iteration over the elements in the collection, where each element receives the message. For each element, the condition is evaluated before the message is sent. Usually though, the condition is used as a filter that selects elements from the collection (e.g. 'all', 'adults', 'new customers' as filters for a collection of Person objects). Only elements selected by the filter will receive the message.

If you want to show that multiple messages are sent in the same iteration, a 'loop' combined fragment can be used. The operator of the combined fragment is 'loop' and the guard represents the condition to control the repetition.



Again, if the receiver of a repeated message is a collection, the condition is generally used to specify a filter for the elements.

For example, to show that the bounds of a drawing are based on those of its visible figures we could draw the following sequence diagram :



Several things are worth noting in this example

* a local variable 'r' was introduced to clarify that it is the result of getBounds that is added.
* naming the resulting Rectangle 'bounds' avoids the introduction of an extra local variable.
* the loop condition is used as a filter on the elements of the figures collection.

# In closing...

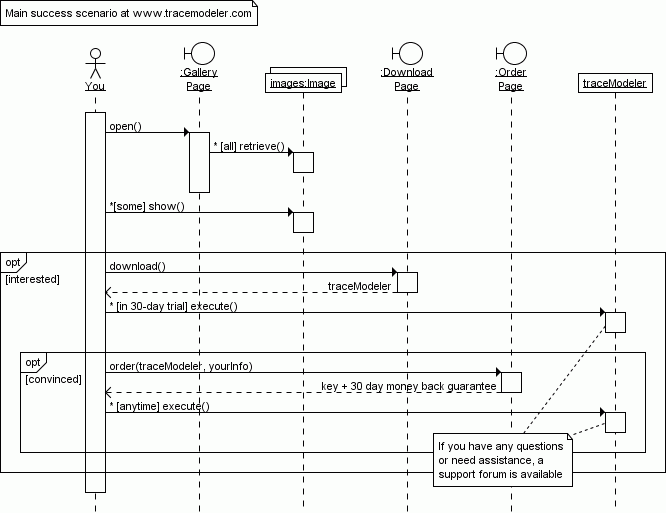
This quick introduction discussed only a few of the possible constructs you may encounter on a UML sequence diagram. A forthcoming article will discuss when an how to use sequence diagrams in projects. Look for it in [**the articles section**](http://www.tracemodeler.com/articles/index.html) or [**subscribe to the news feed**](http://www.tracemodeler.com/rss.xml) to receive notification when new articles and content is published.

As you set off to find out all there is to know about UML sequence diagrams, bear in mind that it is more important to know when and how to use them, than to know every possible construct in the UML specification.

For practical tips on how to improve your own diagrams, have a look at the [**Pimp my diagram series of articles**](http://www.tracemodeler.com/articles/pimp_my_diagram_introduction/index.html). Each episode discusses and improves a real-world sequence diagram example found on the web.

Be sure to take a peek in the [**gallery section**](http://www.tracemodeler.com/gallery/index.html), it contains many examples of sequence diagrams.

When you've reached this point in the text, you should be able to understand most sequence diagrams. You're also a very persistent reader! I leave you with a final thought : the subject of a sequence diagram doesn't have to be an interaction in a software system, any kind of interaction will do. For example, take a close look at the following diagram [wink]

**[](http://www.tracemodeler.com/download/index.html)**