# UML Notes

Matthew C. Schumaker

September 1, 2003

## Contents

<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 UML</td>
<td>3</td>
</tr>
<tr>
<td>1.1 Why use UML?</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Review</td>
<td>3</td>
</tr>
<tr>
<td>1.3 What is UML?</td>
<td>4</td>
</tr>
<tr>
<td>1.4 Things</td>
<td>4</td>
</tr>
<tr>
<td>1.4.1 Structural Things</td>
<td>4</td>
</tr>
<tr>
<td>1.4.2 Behavioral Things</td>
<td>5</td>
</tr>
<tr>
<td>1.5 Group Things</td>
<td>5</td>
</tr>
<tr>
<td>1.5.1 Annotational Things</td>
<td>5</td>
</tr>
<tr>
<td>1.6 Relationships</td>
<td>6</td>
</tr>
<tr>
<td>1.6.1 Dependency</td>
<td>6</td>
</tr>
<tr>
<td>1.6.2 Association</td>
<td>7</td>
</tr>
<tr>
<td>1.6.3 Generalization</td>
<td>7</td>
</tr>
<tr>
<td>1.6.4 Realization</td>
<td>7</td>
</tr>
<tr>
<td>1.7 Diagrams</td>
<td>7</td>
</tr>
<tr>
<td>1.7.1 Class Diagrams</td>
<td>8</td>
</tr>
<tr>
<td>1.7.2 Object Diagrams</td>
<td>8</td>
</tr>
<tr>
<td>1.7.3 Use Case Diagrams</td>
<td>8</td>
</tr>
<tr>
<td>1.7.4 Sequence Diagrams</td>
<td>8</td>
</tr>
<tr>
<td>1.7.5 Collaboration Diagrams</td>
<td>8</td>
</tr>
<tr>
<td>1.7.6 State Chart Diagrams</td>
<td>8</td>
</tr>
<tr>
<td>1.7.7 Activity Diagrams</td>
<td>8</td>
</tr>
<tr>
<td>1.7.8 Component Diagrams</td>
<td>8</td>
</tr>
<tr>
<td>1.7.9 Deployment Diagrams</td>
<td>9</td>
</tr>
<tr>
<td>2 Class Notation</td>
<td>9</td>
</tr>
<tr>
<td>2.1 Name Compartment</td>
<td>9</td>
</tr>
<tr>
<td>2.2 Attribute Compartment</td>
<td>9</td>
</tr>
<tr>
<td>2.2.1 Visibility</td>
<td>10</td>
</tr>
</tbody>
</table>
2.2.2 Multiplicity ........................................ 10
2.2.3 Type and Initial-value ............................. 10
2.2.4 Property String .................................. 10

2.3 Operation Compartment ............................. 10
2.3.1 Parameter-list ................................... 10
2.3.2 Property-string .................................. 11

3 Association Notation ................................. 11

4 Sample Class Diagrams ............................... 11
  4.1 Graph Example ................................... 11
  4.2 Code Example ................................... 11

5 Object Notation ....................................... 14

6 Interactions (Messages) .............................. 14

7 Interaction Diagrams ................................. 15
  7.1 Collaboration Diagram Notation .................... 15
  7.2 Sequence Diagram Notation ....................... 16

8 Sample Interaction Diagrams ......................... 18
  8.1 Simple Collaboration .............................. 18
  8.2 Simple Sequence ................................. 18
  8.3 Real World Collaboration ......................... 18
  8.4 Real World Sequence .............................. 19
1 UML

UML (Unified Modeling Language) is a standard graphical notation for visualizing, specifying, constructing and documenting software. It is useful for both interactions between software developers and clients. It can be as expressive or vague as the author desires.

UML is the result of combining research from three researchers: Grady Booch, Ivar Jacobson and James Rumbaugh. It first came into use in the mid 1990s with version 1.0 submitted in January 1997 to the Object Management Group, OMG.

Currently many companies are involved in the continued development of UML. The leading company is Rational which employed the three originators and is now owned by IBM. GE, Oracle, HP, TI and other companies are all involved with the development of UML.

1.1 Why use UML?

Why do we use UML? is a common questioned asked by many novice and experienced software developers and is complicated to answer.

First and foremost, software development is a very new field. It has only existed in it current form for roughly 40 years. Compared to other sciences and engineering disciplines which can trace their origins back to the Renaissance, software development is barely an infant. Most other fields have established practices and notations, especially engineering. UML is a first step in creating a set notations for describing software.

Why don’t we just use code?, this question can be answered in two parts. First, code actually isn’t very expressive. You can generally fit 80 lines of code on a screen at once while most software projects can be measured in thousands to millions of lines of code. With this in mind, 80 lines isn’t going to tell you much. Second, coding style isn’t universal. While some companies have strict guidelines on how source code is to be formatted, these guidelines aren’t universal. Which means you may be looking through different pieces of code written by different developers all looking slightly different. This adds the burden of having to “parse” their style into your own style.

1.2 Review

To review, UML is a standard set of notation for the visualization of software. Developed in the mid 1990s, it has been adopted as a standard with many companies working on it and supporting it. Its important because its the first widespread standard for describing software. What it accomplishes is hard to do in code.
1.3 What is UML?

As mentioned before UML is a set of notation used to describe software. The descriptions can be: detailed or vague, system or business oriented, static or dynamic. Its this flexibility that makes UML so useful.

UML is composed of three parts: Things, Relationships and Diagrams. Things are the abstractions that are first-class citizens. Relationships tie Things together. Diagrams group interesting collections of Things.

1.4 Things

There are four basic types of Things in UML.

1. Structural Things
2. Behavioral Things
3. Grouping Things
4. Annotational Things

1.4.1 Structural Things

Structural things are the nouns of UML models. There are a variety of different notations for different things. First there is the class:

<table>
<thead>
<tr>
<th>Class Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attributes</td>
</tr>
<tr>
<td>Functions</td>
</tr>
</tbody>
</table>

Figure 1: Class Notation

As seen in figure 1, there are three compartments. The top compartment contains the name of the class. It can also contain some additional information above the class name surrounded by “<<>>”. This can state whether a class is an interface, has a stereotype or other bits of information.
The next thing is an object, figure 2. Objects are instantiated class. This means that a class must exist in the design before an object of that type can be created.

The notation is similar to that of the class except the compartments for attributes and methods is absent. The class name must always be present. The object name is optional. If it is absent then the object is considered anonymous. However the ":" must still be present and the name underlined to differentiate between an object and a class.

These two things are the basic structural things. Other exist and I will cover them if necessary.

1.4.2 Behavioral Things

These are the dynamic parts of UML models. They represent behavior over time and space. There are only two primary types of behavioral things: interactions and state machines. I will only cover interactions for now.

Interactions, see figure ??, represents messages exchanged among objects.

![Method Name](image)

Figure 3: Interaction Notation

In it basic form, the notation consists of a line with an arrow at one end and the name of the method or message being called. The arrow always points to the object the message is being passed to or the object whose method is being invoked.

1.5 Group Things

There is one basic group thing notation and that is a package. Packages are used to organize structural things into groups. You can think of packages as libraries. The basic notation of a package can be seen in figure 4.

1.5.1 Annotational Things

These are the explanatory parts of UML. These can be comments, snippets of codes or anything else that could shed light on a model. The primary annotational thing is the note, figure 5. You would typically use notes to add constraints or comments that are
1.6 Relationships

Relationships are what tie things together. Like things, there are four types of relationships that are available in UML. The four relationships are:

1. Dependency
2. Association
3. Generalization
4. Realization

1.6.1 Dependency

Dependencies show that one object depends on another. Often this is used between storage classes and the objects they store. This is the base type of relationship, all other relationships imply this relationship. Generally the dependency is directional but this isn’t always necessary, a label may also be present. Figure 6 shows a basic dependency. The direction of the dependency points toward the independent thing.
1.6.2 Association

This is probably the most common type of relationship seen in UML. It is used predominantly to show that one class is a member of another. The association is either directional or undirected. In the case of a directional association the arrow points toward the thing that is a member. If no direction is present the association is understood to be bidirectional and both things are aware of each other. Various adornments are often present, multiplicity and role names being the most common. Figure 7 shows an example.

\[ \text{0..1} \]
\[ \text{mEmployee} \]

Figure 7: Association Notation

1.6.3 Generalization

The generalization show the specialization/generalization relationship. specialized element (the children) are substitutable for the generalized thing (the parent). The hollow head always points towards the parent, while the end points toward the children. Figure 8 shows an example.

\[ \text{mEmployee} \]

Figure 8: Generalization Notation

1.6.4 Realization

The last relationship is realization. This is rather specialized relationship that only appears in a couple places. The most common is between a class that defines an interface and the class that implements it. Figure 9 shows an example. The arrow of the realization always points towards the declaration (interface).

\[ \text{mEmployee} \]

Figure 9: Realization Notation

1.7 Diagrams

Diagrams are views into the model. In all, there are nine diagrams. I will be covering only the most common diagrams in detail.
1.7.1 Class Diagrams
This is probably the most common diagram. It consists of classes and relationships. It is considered a static diagram.

1.7.2 Object Diagrams
This is a static diagram as well. Its purpose is to provide a snapshot of the system in use. It will look similar to the class diagram relationships shown among the different objects in the diagram.

1.7.3 Use Case Diagrams
This is a static diagram based off of use cases. Each use case for the system is present with relationships showing how the use cases relate to each other.

1.7.4 Sequence Diagrams
This is a type of interaction diagram. Interaction Diagrams are meant to show change over time. It consists of objects and messages. Sequence diagrams are interaction diagrams the stress the time-ordering of messages. These types of diagrams translate very easily into code.

1.7.5 Collaboration Diagrams
This is another type of interaction diagram. This diagram is a kind of cross between sequence and object diagrams. They are like sequence diagrams in that they show messages being passed but they are like object diagrams in that they stress the structural organization of objects over time-ordering of messages. Collaboration and sequence diagrams are isomorphic, which means they can be transformed from one to the other.

1.7.6 State Chart Diagrams
Another type of interaction diagram, state chart diagrams consists of states, transitions and events. They stress events and how they change the system.

1.7.7 Activity Diagrams
Activity diagrams are a special type of state chart diagrams. They are similar to flow charts in the the states are specific activities undertaken by the system and events cause transitions in these activities.

1.7.8 Component Diagrams
Component diagrams are one of two types of diagrams for modeling the physical aspects of a system. They show the organization and dependencies among a set of components.
1.7.9 Deployment Diagrams

Deployment diagrams are the second type of diagrams for modeling the physical aspects of a system. Generally this shows how machines and resources are layed deployed. It is a useful diagram when trying to get a good over view of how a system is put together.

2 Class Notation

As mentioned above, the UML Class notation consists of box dived into three compartments, see figure 10.

![Class Notation Diagram]

Figure 10: Class Notation

2.1 Name Compartment

The top most compartment contains the name of the class. Additionally, it may contain stereotypes or classifiers between a pair of \texttt{<>} brackets above the class name. Examples of classifiers are \texttt{interface} and \texttt{datatype}. If the class is abstract, the class name will appear in \textit{italics}. Though less common, a multiplicity may appear in the upper right corner which tells how many instances of the class can be created. A multiplicity of 1 means the class is a singleton.

2.2 Attribute Compartment

The next compartment is the attribute (member variables) compartment. The basic form of the each is line is as follows:

\begin{verbatim}
[visibility] name [multiplicity] [: type] [= initial-value] {[property-string]}
\end{verbatim}

Items between [] are optional. If the name is \texttt{underlined} then it is class scoped, i.e. static, if not, it is instance scoped.
2.2.1 Visibility

Visibility means whether the attribute is public, private, protected or package visible. Each visibility has a symbol associated with it.

- Public: +
- Private: -
- Protected: #
- Package:

2.2.2 Multiplicity

Multiplicity refers to how many items there are. Ranges can be specified such as 2..4, 0..*, or 0..1 and explicit numbers can be given. In all cases "*" means infinite.

2.2.3 Type and Initial-value

Type and initial value are self explanatory

2.2.4 Property String

There are three valid property strings: "changeable", "addOnly" and "frozen". Changeable means the value can be replaced any times. addOnly is usually only used when the multiplicity is greater than 1 and means items can only be added, not removed. Frozen means the attribute can not be changed after the item has initialized.

2.3 Operation Compartment

The last section is the operations (member functions). It has a syntax similar to that of attributes.

\[
[\text{visibility}] \ name \ [(\text{parameter-list})] \ [: \ \text{return-type}] \ [(\text{property-string})]
\]

In addition, the scope of the function is related via underlining like attributes and if the name is in *italics* then the function is abstract as well.

2.3.1 Parameter-list

The parameter list has its own comma delimited format as well:

\[
[\text{direction}] \ name : \ type \ [= \ \text{default \ value}]
\]

Direction can be one of three types: in, out, inout. It refers to whether or not a parameter can be modified. In means it can not and out means it can. Inout is a combination of the two.
2.3.2 Property-string

The property-string is different from attributes. For operations there is the choice of:

- isQuery - Function has not side effects, system is unchanged.
- sequential - Callers must coordinate outside the object so only one flow is in the object at one time.
- guarded - The function takes care of controlling access in the case of multiple flows.
- concurrent - The integrity of the object is guaranteed in the case of multiple flows.

3 Association Notation

In figure 11, a simple association is shown. The role names of an association have a similar syntax to that of attributes in class. They can have a visibility and an initial value.

\[ 0..1 \]
\[ \text{mEmployee} \]

Figure 11: Association Notation

There is a further refinement to the association class and that is a composition. A composition is a type of aggregation where parts are need to create a whole. In a normal association, simple aggregation is understood and parts can be shared. In composition parts can not be shared just as a leaf can only belong to one tree. It is represented with a black diamond at the “whole” end of the association. In the case of a tree/leaf composition, the black diamond would be on the tree end of the association.

4 Sample Class Diagrams

4.1 Graph Example

See figure 12.

4.2 Code Example

See figure 13.
Figure 12: UML Example 1
Figure 13: UML Example 2
5 Object Notation

For objects, there isn’t anything too serious to remember. In fact, there are only two types of objects: Named and Anonymous. Figure 14 shows what a named object looks like while figure 15 illustrates and anonymous object.

![Object Name:Class Name](image1)

Figure 14: Named Object

![:Class Name](image2)

Figure 15: Anonymous Object

Technically in UML you can have anonymously typed objects, i.e. objects with just a name. However I shy away from this notation since it can be confusing.

Just like any type of object in the real world, objects in UML can contain state and attribute values. State information is placed below the name and type of an object between [ ] braces. For attributes, a separate compartment is drawn below the name compartment with name/value pairs on the same line. For example “SSN=123-45-6789”. This additional information is useful in Object Diagrams which I will cover later. For dynamic diagrams, their use isn’t as common.

The last notational element of objects is the /textbf{bold} box outline. If an object’s box is bold (thicker), this indicates that the object has some sort of thread or process that it controls.

6 Interactions (Messages)

As mentioned above, interactions are the ”things” of dynamic diagrams. They exist to show change. Defined in UML are five types of messages (interactions).

- **Call**: Invokes an operation on an object; an object may send a message to itself.
- **Return**: Returns a value to the caller
- **Send**: Sends a signal to an object (asynchronous)
- **Create**: Creates an object
- **Destroy**: Destroys an object; an object may commit suicide by destroying itself

Figure 16 shows the general notation for the different message types. Absent from the figure is the text for the message. Depending on the type of diagram used, different text can be used.
7 Interaction Diagrams

Interaction diagrams illustrate how objects interact via messages (interactions). There are two types of interaction diagrams: Sequence and Collaboration.

Collaboration diagrams are useful for a few reasons: they are space economical because they can grow in two dimensions (compared to sequence diagrams), they are better at illustrating complex branching, iteration and concurrent behavior. Their downside is that the notation can be complex and it is difficult to see the sequence of messages that is taking place.

Sequence diagrams are useful because the notation is simpler and the diagram clearly shows the time ordering of messages. Since the diagram can only grow in one direction, deep sequences can’t fit on the screen all at once leading to hard time interpreting the diagram.

7.1 Collaboration Diagram Notation

Collaboration diagrams consist of three elements:

1. Objects
2. Links
3. Messages

Figure 17 shows the anatomy of a collaboration diagram. There are two similar notations that different people use. The above notation is the more standard one.

You may be wondering what a "link" is. In simple terms, a link is an instance of an association. They only exist on collaboration diagrams. Above the link you will find the messages that traverse the link and the direction they go. Each message has three parts. A sequence number which is optional though recommended. The actual
message, which is usually some sort of operation, and any parameters which is another optional feature. Other pieces of information can be shown as well. If the message is part of some iteration an "*" can be shown between the sequence number and ":". An expression can be appended to the iteration in the form of "*[i:=1..N]". Conditions can also be shown between the sequence number and ":" by putting the condition between a set of [] braces.

Sequence numbers in collaboration diagrams can be difficult to keep track, especially if a nested numbering is used. I feel that you should avoid nested numbering unless absolutely necessary.

7.2 Sequence Diagram Notation

Sequence Diagrams are involve a simpler, more natural (to most programmers) notation for illustrating change. These diagrams consist of only two elements:

1. Objects
2. Messages

However, the notation for an object has some additions to it. First it has a dashed line extending down from the bottom of the object. This line is the "lifeline" of the object and signifies how long the object exists. When an object is explicitly destroyed during the course of the diagram, a large "X" is placed at the bottom of the lifeline to indicate the loss of the object. The next addition is the activation box. This box sits on the lifeline and indicates that the object is performing or waiting on some computation. Figure 18 shows these parts of an object.
Generally speaking, unless an object is explicitly created or destroyed during a sequence diagram then it exists before or after the completion of the sequence.

The next part of a sequence diagram are the messages. In sequence diagrams, you will not see any links, only messages. Here the different types of messages will be used more often as well. In all cases messages originate from activation boxes and point towards the receiving object. They can either point towards the actual object box in the case of object creation, they can point towards the top of an activation box or point towards a destruction marker. Like collaboration diagrams, there is a notation that can be used on the actual message to specify different types of message actions. Figure 19 shows these different notations.

You may ask how do you call a static method? Simply, a class is added to the diagram. Everything else remains the same.

What do you do if a number of actions need to be performed in an iteration? Surround the messages with a box and place the iteration syntax at the bottom of the box.
8 Sample Interaction Diagrams

8.1 Simple Collaboration
This example is a simple set of messages needed to add a node to a graph from a file, see figure 20

Figure 20: Simple Collaboration Diagram

8.2 Simple Sequence
The same set of messages except in the form of a sequence diagram, figure 21

8.3 Real World Collaboration
From my thesis, yet again, is a section of code for the performing of a Kernigan and Lin graph bisection. It is highly simplified, see the handout for the code this was derived from
8.4 Real World Sequence

Same example except as a sequence diagram, figure 23
mGraph : GraphAdjacencyList

mCells : Vector

mNets : Vector

mBisection : BisectionKLFM

bisection : Bisection

1 : getNodeList()

2 *[i:=1..N] : add(new Cell(gNode.getId()))

3 *[i:=1..M] : addCell(cell)

4 : <<create>>

5 *[i:=1..N] : addCell(cell)

6 *[i:=1..M] : addCell(cell)

7 *[i:=1..P] : performPass()

Figure 22: Real World Collaboration Diagram
getNodeList()

*[{i:=1..N}] add(new Cell(gNode.getId()))

*[{i:=1..M}] add(net)

<<create>>

*[{i:=1..N}] addCell(cell)

*[{i:=1..M}] addNet(net)

*[{i:=1..P}] performPass()