UML basics: An introduction to the Unified Modeling Language

Getting started with visual modeling of your software

Donald Bell   June 15, 2003

from The Rational Edge: This introduction to the Unified Modeling Language, or UML, provides an overview of the most important diagrams used in the visual modeling of computing programs. The article is ideal for those who have little knowledge of UML concepts, including managers as well as novice practitioners.

View more content in this series

Way back in the late twentieth century —

1997 to be exact — the Object Management Group (OMG) released the Unified Modeling Language (UML). One of the purposes of UML was to provide the development community with a stable and common design language that could be used to develop and build computer applications. UML brought forth a unified standard modeling notation that IT professionals had been wanting for years. Using UML, IT professionals could now read and disseminate system structure and design plans — just as construction workers have been doing for years with blueprints of buildings.

It is now the twenty-first century — 2003 to be precise — and UML has gained traction in our profession. On 75 percent of the resumes I see, there is a bullet point claiming knowledge of UML. However, after speaking with a majority of these job candidates, it becomes clear that they do not truly know UML. Typically, they are either using it as a buzz word, or they have had a sliver of exposure to UML. This lack of understanding inspired me to write this quick introduction to UML, focused on the basic diagrams used in visual modeling. When you are finished reading you will not have enough knowledge to put UML on your resume, but you will have a starting point for digging more deeply into the language.

Deploy with confidence

Consistently deliver high-quality software faster using DevOps Continuous Delivery.
Edit your code anywhere with Git repos and issue tracking, deliver continuously with an automated pipeline, get Insights to improve quality, and more.

A little background

As I mentioned, UML was meant to be a unifying language enabling IT professionals to model computer applications. The primary authors were Jim Rumbaugh, Ivar Jacobson, and Grady Booch, who originally had their own competing methods (OMT, OOSE, and Booch). Eventually, they joined forces and brought about an open standard. (Sound familiar? A similar phenomenon spawned J2EE, SOAP, and Linux.) One reason UML has become a *standard* modeling language is that it is programming-language independent. (UML modeling tools from IBM Rational are used extensively in J2EE shops as well in .NET shops.) Also, the UML notation set is a language and not a methodology. This is important, because a language, as opposed to a methodology, can easily fit into any company's way of conducting business without requiring change.

Try Bluemix free for 30 days

Take advantage of powerful Bluemix services and infrastructure offerings to build, deploy, and run your apps in the cloud. Give it a try!

Since UML is not a methodology, it does not require any formal work products (i.e., "artifacts" in IBM Rational Unified Process® lingo). Yet it does provide several types of diagrams that, when used within a given methodology, increase the ease of understanding an application under development. There is more to UML than these diagrams, but for my purposes here, the diagrams offer a good introduction to the language and the principles behind its use. By placing standard UML diagrams in your methodology's work products, you make it easier for UML-proficient people to join your project and quickly become productive. The most useful, standard UML diagrams are: use case diagram, class diagram, sequence diagram, statechart diagram, activity diagram, component diagram, and deployment diagram.

It is beyond the scope of this introductory article to go into great detail about each type of diagram. Instead, I will provide you with enough information for a general understanding of each one and then supply more details in later articles.

Use-case diagram

A use case illustrates a unit of functionality provided by the system. The main purpose of the use-case diagram is to help development teams visualize the functional requirements of a system, including the relationship of "actors" (human beings who will interact with the system) to essential processes, as well as the relationships among different use cases. Use-case diagrams generally show groups of use cases — either all use cases for the complete system, or a breakout of a particular group of use cases with related functionality (e.g., all security administration-related use cases). To show a use case on a use-case diagram, you draw an oval in the middle of the diagram and put the name of the use case in the center of, or below, the oval. To draw an actor (indicating a system user) on a use-case diagram, you draw a stick person to the left or right of your diagram (and just in case you're wondering, some people draw prettier stick people than others). Use simple lines to depict relationships between actors and use cases, as shown in Figure 1.

Figure 1: Sample use-case diagram
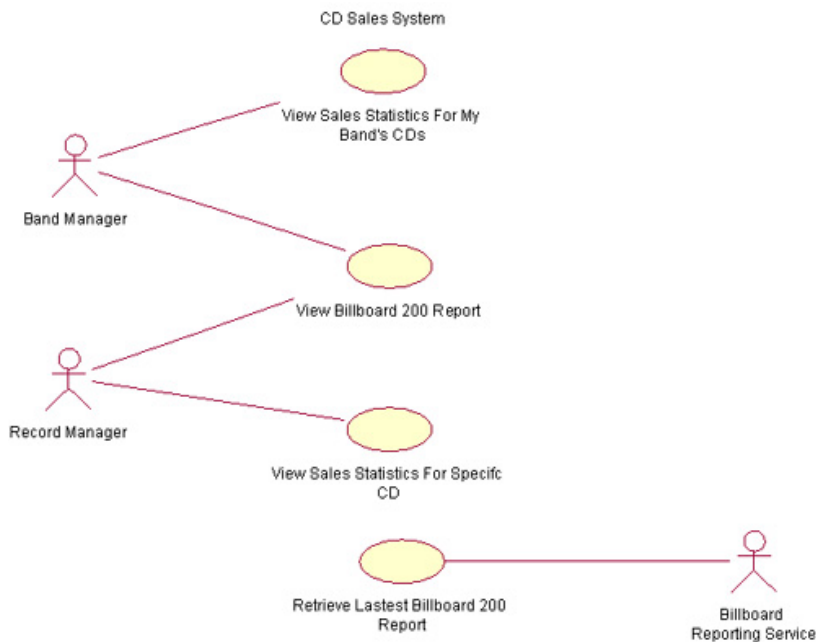
A use-case diagram is typically used to communicate the high-level functions of the system and the system's scope. By looking at our use-case diagram in Figure 1, you can easily tell the functions that our example system provides. This system lets the band manager view a sales statistics report and the Billboard 200 report for the band's CDs. It also lets the record manager view a sales statistics report and the Billboard 200 report for a particular CD. The diagram also tells us that our system delivers Billboard reports from an external system called Billboard Reporting Service.

In addition, the absence of use cases in this diagram shows what the system *doesn't* do. For example, it does not provide a way for a band manager to listen to songs from the different albums on the Billboard 200 — i.e., we see no reference to a use case called Listen to Songs from Billboard 200. This absence is not a trivial matter. With clear and simple use-case descriptions provided on such a diagram, a project sponsor can easily see if needed functionality is present or not present in the system.

Class diagram

The class diagram shows how the different entities (people, things, and data) relate to each other; in other words, it shows the static structures of the system. A class diagram can be used to display logical classes, which are typically the kinds of things the business people in an organization talk about — rock bands, CDs, radio play; or loans, home mortgages, car loans, and interest rates. Class diagrams can also be used to show implementation classes, which are the things that programmers typically deal with. An implementation class diagram will probably show some of the same classes as the logical classes diagram.The implementation class diagram won't be drawn with the same attributes, however, because it will most likely have references to things like Vectors and HashMaps.

A class is depicted on the class diagram as a rectangle with three horizontal sections, as shown in Figure 2. The upper section shows the class's name; the middle section contains the class's attributes; and the lower section contains the class's operations (or "methods").

Figure 2: Sample class object in a class diagram
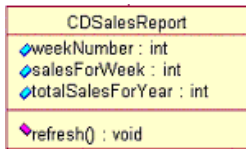
In my experience, almost every developer knows what this diagram is, yet I find that most programmers draw the relationship lines incorrectly. For a class diagram like the one in Figure 3, you should draw the inheritance relationship[1] using a line with an arrowhead at the top pointing to the super class, and the arrowhead should be a *completed triangle*. [Note: For more information on inheritance and other object-oriented principles, see the Java tutorial What Is Inheritance?] An association relationship should be a solid line if both classes are aware of each other and a line with an *open arrowhead* if the association is known by only one of the classes.

Figure 3: A complete class diagram, including the class object shown in Figure 2
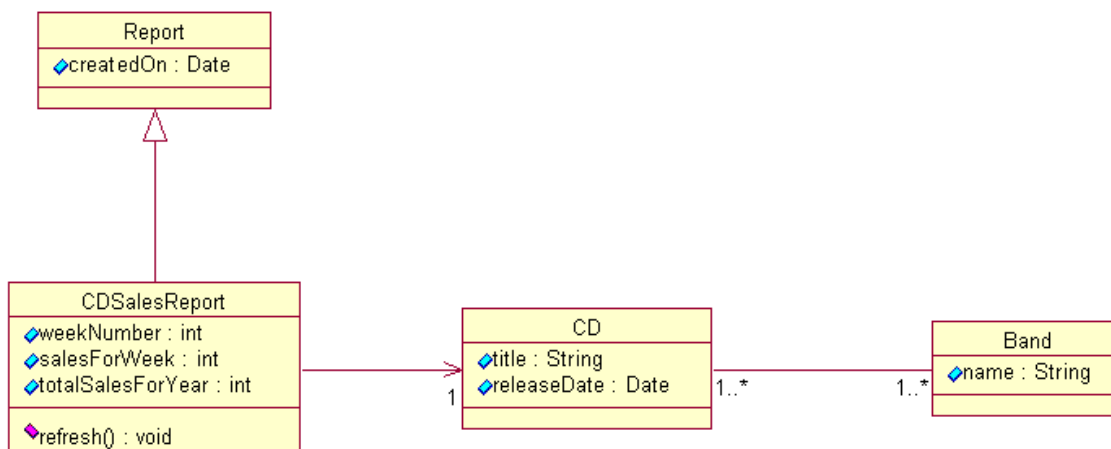
In Figure 3, we see both the inheritance relationship and two association relationships. The CDSalesReport class inherits from the Report class. A CDSalesReport is associated with one CD, but the CD class doesn't know anything about the CDSalesReport class. The CD and the Band classes both know about each other, and both classes can be associated to one or more of each other.

A class diagram can incorporate many more concepts, which we will cover later in this article series.
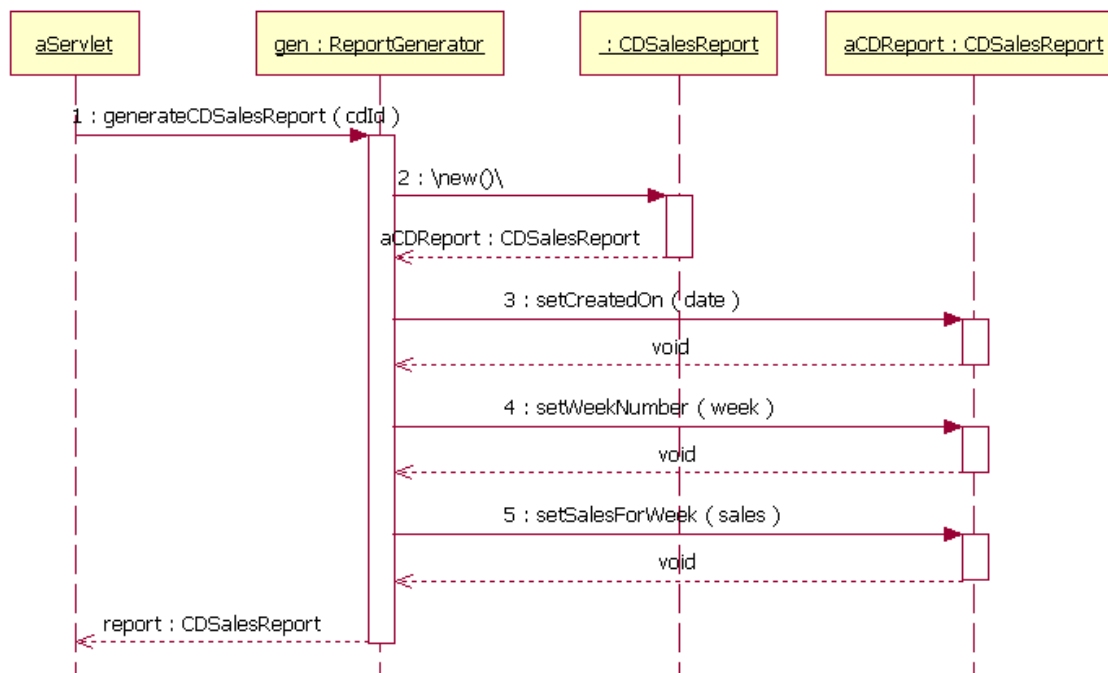
Sequence diagram

Sequence diagrams show a detailed flow for a specific use case or even just part of a specific use case. They are almost self explanatory; they show the calls between the different objects in their sequence and can show, at a detailed level, different calls to different objects.

A sequence diagram has two dimensions: The vertical dimension shows the sequence of messages/calls in the time order that they occur; the horizontal dimension shows the object instances to which the messages are sent.

A sequence diagram is very simple to draw. Across the top of your diagram, identify the class instances (objects) by putting each class instance inside a box (see Figure 4). In the box, put the class instance name and class name separated by a space/colon/space " : " (e.g., myReportGenerator : ReportGenerator). If a class instance sends a message to another class instance, draw a line with an open arrowhead pointing to the receiving class instance; place the name of the message/method above the line. Optionally, for important messages, you can draw a dotted line with an arrowhead pointing back to the originating class instance; label the return value above the dotted line. Personally, I always like to include the return value lines because I find the extra details make it easier to read.

Reading a sequence diagram is very simple. Start at the top left corner with the "driver" class instance that starts the sequence. Then follow each message down the diagram. Remember: Even though the example sequence diagram in Figure 4 shows a return message for each sent message, this is optional.

Figure 4: A sample sequence diagram

By reading our sample sequence diagram in Figure 4, you can see how to create a CD Sales Report. The aServlet object is our example driver. aServlet sends a message to the

ReportGenerator class instance named gen. The message is labeled generateCDSalesReport, which means that the ReportGenerator object implements this message handler. On closer inspection, the generateCDSalesReport message label has cdId in parentheses, which means that aServlet is passing a variable named cdId with the message. When gen instance receives a generateCDSalesReport message, it then makes subsequent calls to the CDSalesReport class, and an actual instance of a CDSalesReport called aCDReport gets returned. The gen instance then makes calls to the returned aCDReport instance, passing it parameters on each message call. At the end of the sequence, the gen instance returns aCDReport to its caller aServlet.

Please note: The sequence diagram in Figure 4 is arguably too detailed for a typical sequence diagram. However, I believe it is simple enough to understand, and it shows how nested calls are drawn. Also, with junior developers, sometimes it is necessary to break down sequences to this explicit level to help them understand what they are supposed to do.

Statechart diagram

The statechart diagram models the different states that a class can be in and how that class transitions from state to state. It can be argued that every class has a state, but that every class shouldn't have a statechart diagram. Only classes with "interesting" states — that is, classes with three or more potential states during system activity — should be modeled.

As shown in Figure 5, the notation set of the statechart diagram has five basic elements: the initial starting point, which is drawn using a solid circle; a transition between states, which is drawn using a line with an open arrowhead; a state, which is drawn using a rectangle with rounded corners; a decision point, which is drawn as an open circle; and one or more termination points, which are drawn using a circle with a solid circle inside it. To draw a statechart diagram, begin with a starting point and a transition line pointing to the initial state of the class. Draw the states themselves anywhere on the diagram, and then simply connect them using the state transition lines.

Figure 5: Statechart diagram showing the various states that classes pass through in a functioning system
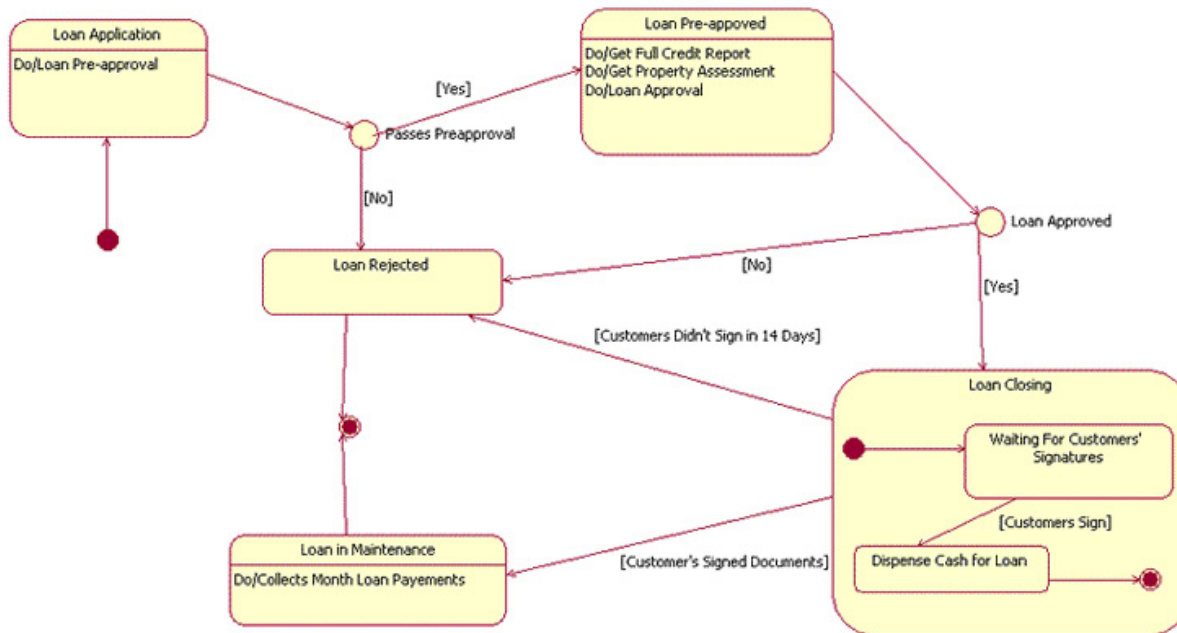
The example statechart diagram in Figure 5 shows some of the potential information they can communicate. For instance, you can tell that loan processing begins in the Loan Application state. When the pre-approval process is done, depending on the outcome, you move to either the Loan Pre-approved state or the Loan Rejected state. This decision, which is made during the transition process, is shown with a decision point — the empty circle in the transition line. By looking at the example, a person can tell that a loan cannot go from the Loan Pre-Approved state to the Loan in Maintenance state without going through the Loan Closing state. Also, by looking at our example diagram, a person can tell that all loans will end in either the Loan Rejected state or the Loan in Maintenance state.
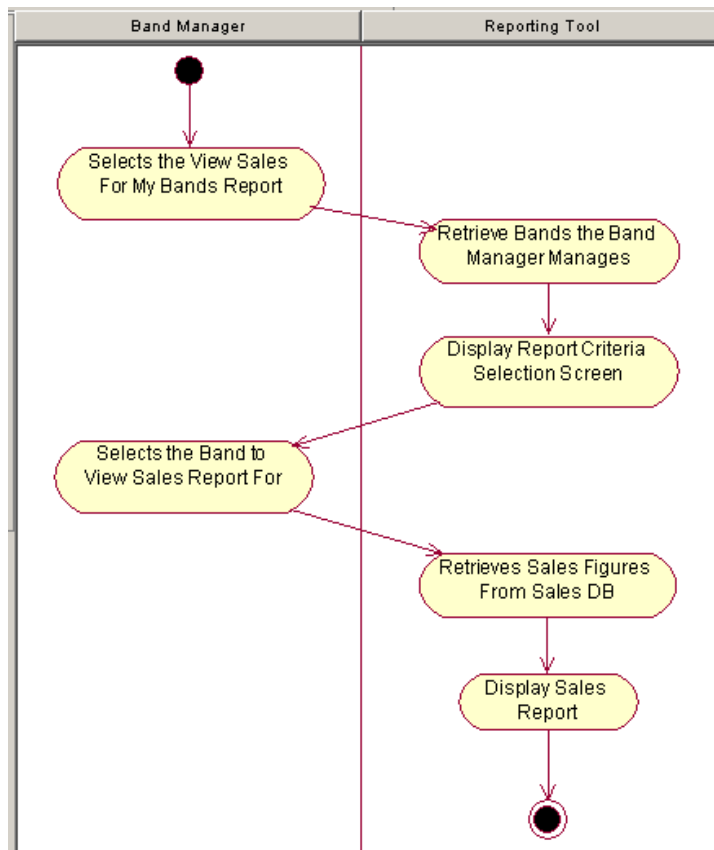
Activity diagram

Activity diagrams show the procedural flow of control between two or more class objects while processing an activity. Activity diagrams can be used to model higher-level business process at the business unit level, or to model low-level internal class actions. In my experience, activity diagrams are best used to model higher-level processes, such as how the company is currently doing business, or how it would like to do business. This is because activity diagrams are "less technical" in appearance, compared to sequence diagrams, and business-minded people tend to understand them more quickly.

An activity diagram's notation set is similar to that used in a statechart diagram. Like a statechart diagram, the activity diagram starts with a solid circle connected to the initial activity. The activity is modeled by drawing a rectangle with rounded edges, enclosing the activity's name. Activities can be connected to other activities through transition lines, or to decision points that connect to different activities guarded by conditions of the decision point. Activities that terminate the modeled process are connected to a termination point (just as in a statechart diagram). Optionally,

the activities can be grouped into swimlanes, which are used to indicate the object that actually performs the activity, as shown in Figure 6.

Figure 6: Activity diagram, with two swimlanes to indicate control of activity by two objects: the band manager, and the reporting tool

In our example activity diagram, we have two swimlanes because we have two objects that control separate activities: a band manager and a reporting tool. The process starts with the band manager electing to view the sales report for one of his bands. The reporting tool then retrieves and displays all the bands that person manages and asks him to choose one. After the band manager selects a band, the reporting tool retrieves the sales information and displays the sales report. The activity diagram shows that displaying the report is the last step in the process.
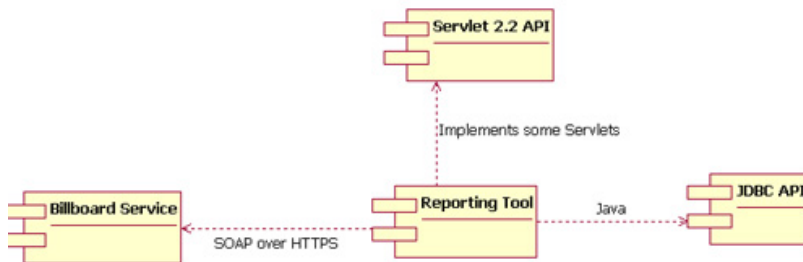
Component diagram

A component diagram provides a physical view of the system. Its purpose is to show the dependencies that the software has on the other software components (e.g., software libraries) in the system. The diagram can be shown at a very high level, with just the large-grain components, or it can be shown at the component package level. [Note: The phrase component package level is a programming language-neutral way of referring to class container levels such as .NET's namespaces (e.g., System.Web.UI) or Java's packages (e.g., java.util).]

Modeling a component diagram is best described through an example. Figure 7 shows four components: Reporting Tool, Billboard Service, Servlet 2.2 API, and JDBC API. The arrowed

lines from the Reporting Tool component to the Billboard Service, Servlet 2.2 API, and JDBC API components mean that the Reporting Tool is dependent on those three components.

Figure 7: A component diagram shows interdependencies of various software components the system comprises

Deployment diagram

The deployment diagram shows how a system will be physically deployed in the hardware environment. Its purpose is to show where the different components of the system will physically run and how they will communicate with each other. Since the diagram models the physical runtime, a system's production staff will make considerable use of this diagram.

The notation in a deployment diagram includes the notation elements used in a component diagram, with a couple of additions, including the concept of a node. A node represents either a physical machine or a virtual machine node (e.g., a mainframe node). To model a node, simply draw a three-dimensional cube with the name of the node at the top of the cube. Use the naming convention used in sequence diagrams: [instance name] : [instance type] (e.g., "w3reporting.myco.com : Application Server").

Figure 8: Deployment diagram
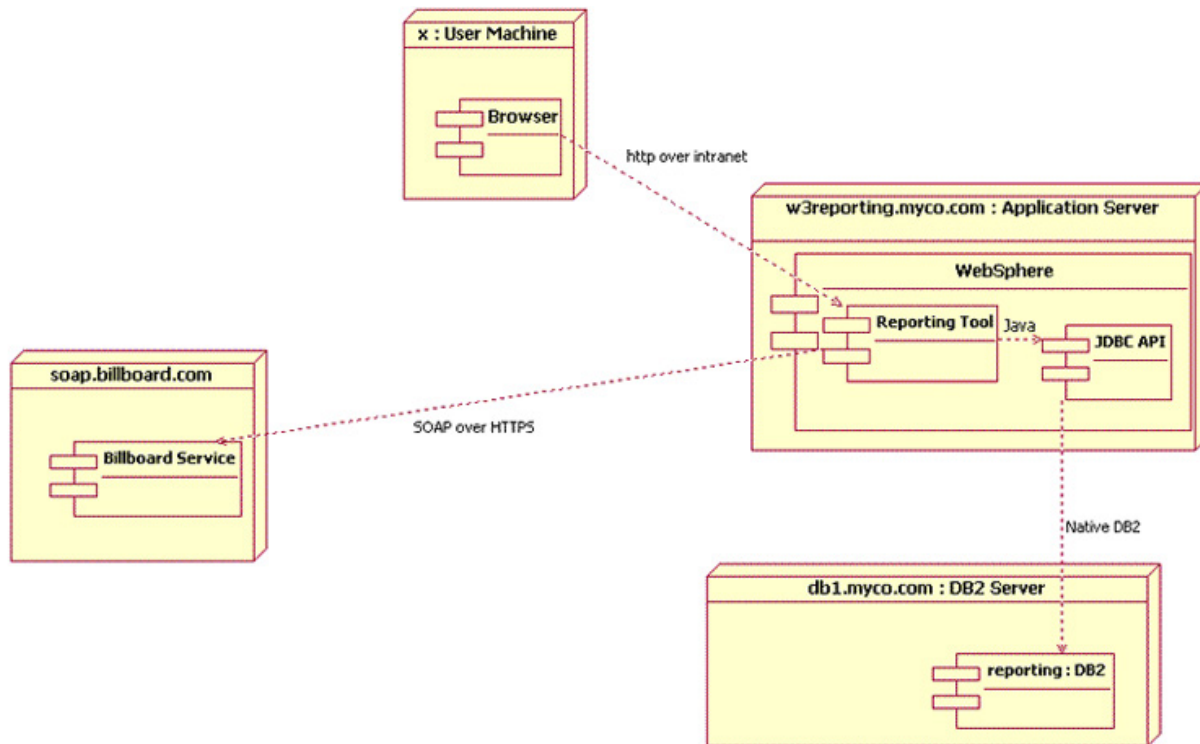
The deployment diagram in Figure 8 shows that the users access the Reporting Tool by using a browser running on their local machine and connecting via HTTP over their company's intranet to the Reporting Tool. This tool physically runs on the Application Server named w3reporting.myco.com. The diagram shows the Reporting Tool component drawn inside of IBM WebSphere, which in turn is drawn inside of the node w3.reporting.myco.com. The Reporting Tool connects to its reporting database using the Java language to IBM DB2's JDBC interface, which then communicates to the actual DB2 database running on the server named db1.myco.com using native DB2 communication. In addition to talking to the reporting database, the Report Tool component communicates via SOAP over HTTPS to the Billboard Service.

Conclusion

Although this article provides only a brief introduction to Unified Modeling Language, I encourage you to start applying the information you have learned here to your own projects and to dig more deeply into UML. There are several software tools that help you to integrate UML diagrams into your software development process, but even without automated tools, you can use markers on a whiteboard or paper and pencils to draw your UML diagrams and still achieve benefits.

Related topics

- UML Modeler in IBM Rational Software Architect Version 7.5
- Rational Tau

© Copyright IBM Corporation 2003
(www.ibm.com/legal/copytrade.shtml)
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)