UML basics: The component diagram

Donald Bell December 15, 2004

from The Rational Edge: This article introduces the component diagram, a structure diagram within the new Unified Modeling Language 2.0 specification.

View more content in this series

*This is the next installment in a series of articles about the essential diagrams used within the Unified Modeling Language, or UML. In my previous article on the UML's class diagram, (*The Rational Edge*, September 2004), I described how the class diagram's notation set is the basis for all UML 2's structure diagrams. Continuing down the track of UML 2 structure diagrams, this article introduces the component diagram.*

The diagram's purpose

The component diagram's main purpose is to show the structural relationships between the components of a system. In UML 1.1, a component represented implementation items, such as files and executables. Unfortunately, this conflicted with the more common use of the term component," which refers to things such as COM components. Over time and across successive releases of UML, the original UML meaning of components was mostly lost. UML 2 officially changes the essential meaning of the component concept; in UML 2, components are considered autonomous, encapsulated units within a system or subsystem that provide one or more interfaces. Although the UML 2 specification does not strictly state it, components are larger design units that represent things that will typically be implemented using replaceable" modules. But, unlike UML 1.x, components are now strictly logical, design-time constructs. The idea is that you can easily reuse and/or substitute a different component implementation in your designs because a component encapsulates behavior and implements specified interfaces. [Note: The physical items that UML1.x called components are now called "artifacts" in UML 2. An artifact is a physical unit, such as a file, executable, script, database, etc. Only artifacts live on physical nodes; classes and components do not have "location." However, an artifact may manifest components and other classifiers (i.e., classes). A single component could be manifested by multiple artifacts, which could be on the same or different nodes, so a single component could indirectly be implemented on multiple nodes.]

In component-based development (CBD), component diagrams offer architects a natural format to begin modeling a solution. Component diagrams allow an architect to verify that a system's required functionality is being implemented by components, thus ensuring that the eventual system will be acceptable.

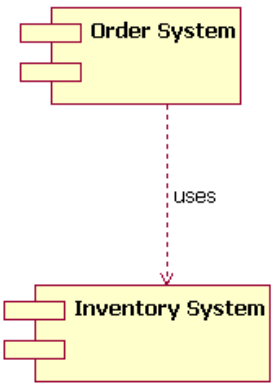
In addition, component diagrams are useful communication tools for various groups. The diagrams can be presented to key project stakeholders and implementation staff. While component diagrams are generally geared towards a system's implementation staff, component diagrams can generally put stakeholders at ease because the diagram presents an early understanding of the overall system that is being built.

Developers find the component diagram useful because it provides them with a high-level, architectural view of the system that they will be building, which helps developers begin formalizing a roadmap for the implementation, and make decisions about task assignments and/or needed skill enhancements. System administrators find component diagrams useful because they get an early view of the logical software components that will be running on their systems. Although system administrators will not be able to identify the physical machines or the physical executables from the diagram, a component diagram will nevertheless be welcomed because it provides early information about the components and their relationships (which allows sys-admins to loosely plan ahead).

The notation

The component diagram notation set now makes it one of the easiest UML diagrams to draw. Figure 1 shows a simple component diagram using the former UML 1.4 notation; the example shows a relationship between two components: an Order System component that uses the Inventory System component. As you can see, a component in UML 1.4 was drawn as a rectangle with two smaller rectangles protruding from its left side.

Figure 1: This simple component diagram shows the Order System's general dependency using UML 1.4 notation



The above UML 1.4 notation is still supported in UML 2. However, the UML 1.4 notation set did not scale well in larger systems. For that reason, UML 2 dramatically enhances the notation set of the component diagram, as we will see throughout the rest of this article. The UML 2 notation set scales better, and the notation set is also more informative while maintaining its ease of understanding.

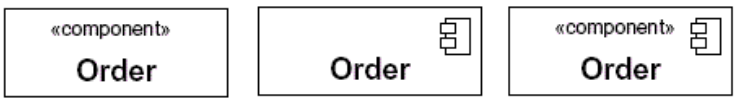
Let's step through the component diagram basics according to UML 2.

The basics

Drawing a component in UML 2 is now very similar to drawing a class on a class diagram. In fact, in UML 2 a component is merely a specialized version of the class concept. Which means that the notation rules that apply to the class classifier also apply to the component classifier. (If you read and understood my previous article [http://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/index.html] regarding structure diagrams in general, and class diagrams in particular, you are well under way to understanding component diagrams.)

In UML 2, a component is drawn as a rectangle with optional compartments stacked vertically. A high-level, abstracted view of a component in UML 2 can be modeled as just a rectangle with the component's name and the component stereotype text and/or icon. The component stereotype's text is «component»" and the component stereotype icon is a rectangle with two smaller rectangles protruding on its left side (the UML 1.4 notation element for a component). Figure 2 shows three different ways a component can be drawn using the UML 2 specification.

Figure 2: The different ways to draw a component's name compartment



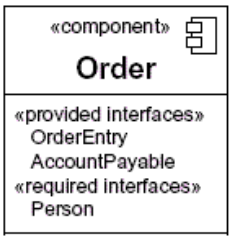
When drawing a component on a diagram, it is important that you always include the component stereotype text (the word "component" inside double angle brackets, as shown in Figure 2) and/or icon. The reason? In UML, a rectangle without any stereotype classifier is interpreted as a class element. The component stereotype and/or icon distinguishes this rectangle as a component element.

Modeling a component's interfaces Provided/Required

The Order components drawn in Figure 2 all represent valid notation elements; however, a typical component diagram includes more information. A component element can have additional compartments stacked below the name compartment. As mentioned earlier, a component is an

autonomous unit that provides one or more public interfaces. The interfaces provided represent the formal contract of services the component provides to its consumers/clients. Figure 3 shows the Order component having a second compartment that denotes what interfaces the Order component provides and requires. [Note: Even though components are autonomous units they still may depend on the services provided by other components. Because of this, documenting a component's required interfaces is useful.]

Figure 3: The additional compartment here shows the interfaces that the Order component provides and requires

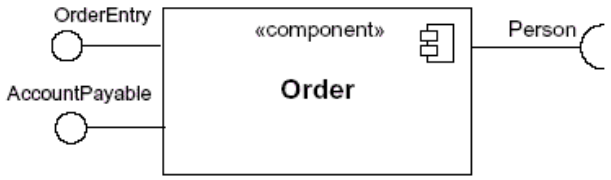


In the example Order component shown in Figure 3, the component provides the interfaces of OrderEntry and AccountPayable. Additionally, the component also requires another component that provides the Person interface. [Note: Figure 3 does not show the Order component in its complete context. In a real-world model the OrderEntry, AccountPayable, and Person interfaces would be present in the system's model.]

Another approach to modeling a component's interfaces

UML 2 has also introduced another way to show a component's provided and required interfaces. This second way builds off the single rectangle, with the component's name in it, and places what the UML 2 specification calls interface symbols" connected to the outside of the rectangle. This second approach is illustrated in Figure 4.

Figure 4: An alternative approach (compare with Figure 3) to showing a component's provided/required interfaces using interface symbols



In this second approach the interface symbols with a complete circle at their end represent an interface that the component provides — this "lollipop" symbol is shorthand for a realization relationship of an interface classifier. Interface symbols with only a half circle at their end (a.k.a. sockets) represent an interface that the component requires (in both cases, the interface's name

is placed near the interface symbol itself). Even though Figure 4 looks much different from Figure 3, both figures provide the same information — i.e., the Order component *provides* two interfaces: OrderEntry and AccountPayable, and the Order component *requires* the Person interface.

Modeling a component's relationships

When showing a component's relationship with other components, the lollipop and socket notation must also include a dependency arrow (as used in the class diagram). On a component diagram with lollipops and sockets, note that the dependency arrow comes out of the consuming (requiring) socket and its arrow head connects with the provider's lollipop, as shown in Figure 5.

Figure 5: A component diagram that shows how the Order System component depends on other components

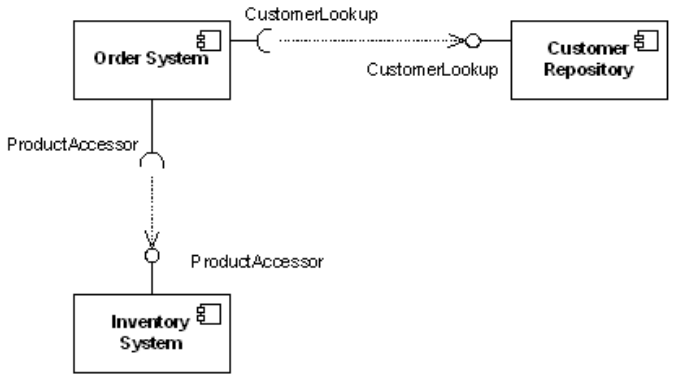


Figure 5 shows that the Order System component depends both on the Customer Repository and Inventory System components. Notice in Figure 5 the duplicated names of the interfaces CustomerLookup" and ProductAccessor." While this may seem unnecessarily repetitive in this example, the notation actually allows for different interfaces (and differing names) on each component depending on the implementation differences (e.g., one component provides an interface that is a subclass of a smaller required interface).

Subsystems

In UML 2 the subsystem classifier is a specialized version of a component classifier. Because of this, the subsystem notation element inherits all the same rules as the component notation element. The only difference is that a subsystem notation element has the keyword of subsystem" instead of component," as shown in Figure 6.

Figure 6: An example of a subsystem element

The UML 2 specification is quite vague on how a subsystem is different from a component. The specification does not treat a component or a subsystem any differently from a modeling perspective. Compared with UML 1.x, this UML 2 modeling ambiguity is new. But there's a reason. In UML 1.x, a subsystem was considered a package, and this package notation was confusing to many UML practitioners; hence UML 2 aligned subsystems as a specialized component, since this is how most UML 1.x users understood it. This change did introduce fuzziness into the picture, but this fuzziness is more of a reflection of reality versus a mistake in the UML 2 specification.

So right now you are probably scratching your head wondering when to use a component element versus a subsystem element. Quite frankly, I do not have a direct answer for you. I can tell you that the UML 2 specification says that the decision on when to use a component versus a subsystem is up to the methodology of the modeler. I personally like this answer because it helps ensure that UML stays methodology independent, which helps keep it universally usable in software development.

Beyond the basics

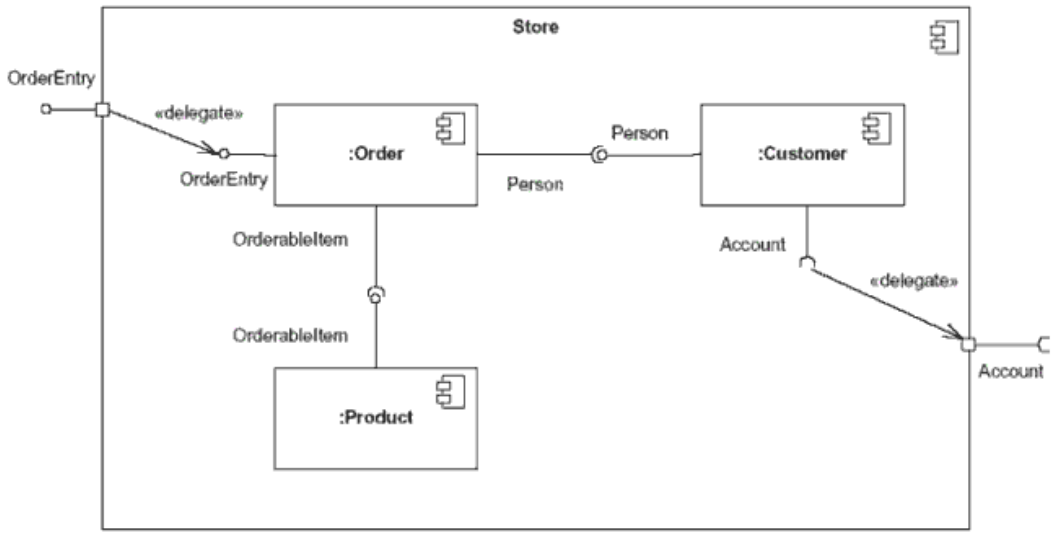
The component diagram is one of the easier-to-understand diagrams, so there is not much to cover beyond the basics. However, there is one area you may consider somewhat advanced.

Showing a component's internal structure

There will be times when it makes sense to display a component's internal structure. In my previous article on the class diagram, I showed how to model a class's internal structure; here I will focus on how to model a component's internal structure when it is composed of other components.

To show a component's inner structure, you merely draw the component larger than normal and place the inner parts inside the name compartment of the encompassing component. Figure 7 show's the Store's component inner structure.

Figure 7: This component's inner structure is composed of other components



Using the example shown in Figure 7, the Store component provides the interface of OrderEntry and requires the interface of Account. The Store component is made up of three components: Order, Customer, and Product components. Notice how the Store's OrderEntry and Account interface symbols have a square on the edge of the component. This square is called a port. In a simplistic sense, ports provide a way to model how a component's *provided/required* interfaces relate to its internal parts. [Note: In actuality, ports are applicable to any type of classifier (i.e., to a class or some other classifier your model might have). To keep this article simple, I refer to ports in their use on component classifiers.

By using a port, our diagram is able to de-couple the internals of the Store component from external entities. In Figure 7, the OrderEntry port delegates to the Order component's OrderEntry interface for processing. Also, the internal Customer component's required Account interface is delegated to the Store component's required Account interface port. By connecting to the Account port, the internals of the Store component (e.g. the Customer component) can have a local representative of some unknown external entity which implements the port's interface. The required Account interface will be implemented by a component outside of the Store component. [Note: Typically, when you draw a dependency relationship between a port and an interface, the dependent (requiring) interface will handle all the processing logic at execution time. However, this is not a hard and fast rule — it is completely acceptable for the encompassing component (e.g., the Store component in our example) to have its own processing logic instead of merely delegating the processing to the dependant interface.]

You will also notice in Figure 7 that the interconnections between the inner components are different from those shown in Figure 5. This is because these depictions of internal structures are really collaboration diagrams nested inside the classifier (a component, in our case), since collaboration diagrams show instances or roles of classifiers. The relationship modeled between the internal components is drawn with what UML calls an assembly connector." An assembly connector ties one component's *provided* interface with another component's *required* interface. Assembly connectors are drawn as lollipop and socket symbols next to each other. Drawing these assembly connectors in this manner makes the lollipop and socket symbols very easy to read.

Conclusion

The component diagram is a very important diagram that architects will often create early in a project. However, the component diagram's usefulness spans the life of the system. Component diagrams are invaluable because they model and document a system's architecture. Because component diagrams document a system's architecture, the developers and the eventual system administrators of the system find this work product-critical in helping them understand the system.

Related topics

- UML Modeler in IBM Rational Software Architect Version 7.5
- Rational Tau
- Rational Test Workbench Web UI Tester