Architecting Systems with UML 2.0

Morgan Björkander and Cris Kobryn, *Telelogic*

Signaling the end of the method wars, the Object Management Group (OMG) first standardized the Unified Modeling Language[1] in 1997. The software industry rapidly accepted it as the standard modeling language for specifying software and system architectures. Although UML is primarily intended for general-purpose modeling, it's receiving extensive use in diverse specialized areas, such as business process modeling and real-time-systems modeling.

Despite these successes, development tools have been slow to realize UML's full potential. In addition, the software industry has evolved considerably during the last six years, and the first version of UML (UML 1.x) is now dated. Two examples illustrate the difficulty of programming in UML 1.x. The first is the mainstreaming of component-based development for enterprise applications, as with the propagation of J2EE (Java 2 Platform, Enterprise Edition), COM+ (the extension to Microsoft's Component Object Model), and, recently, Microsoft's .NET. The second is the common use of more mature modeling languages, such as SDL (Specification and Description Language)[2] and ROOM (Real-Time Object-Oriented Modeling),[3] to specify components and system architectures for real-time applications. If you try to model either of these demanding domains with UML 1.x, you will soon find that it does not cope well with their paradigms or complexity. Consequently, UML tools serving these domains have had to provide proprietary language extensions to address UML's precision and scalability limitations.

Despite the successes of UML 1.x, modeling tools have been slow to realize its full potential for specifying software and system architectures. UML 2.0 will make the language more current and improve its expressive power and precision.

During the last few years, application creation's focus has shifted from code to models, and model-driven development has taken off. Various model transformation and code generation techniques let you automatically generate applications from models. Some tools take this

0740-7459/03/$17.00 © 2003 IEEE

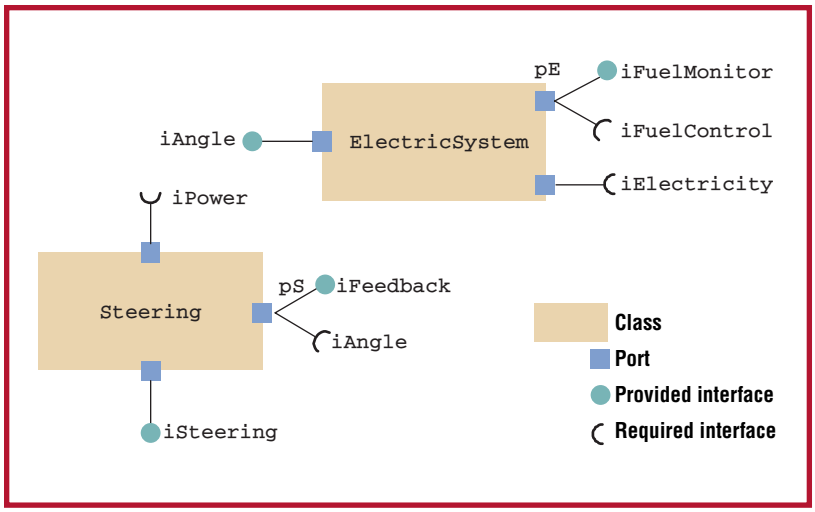


Figure 1. Classes with ports and interfaces.

one step further, and also let you execute the models. In this context, a modeling language can serve as a visual programming language, but with variable levels of abstraction. This approach allows much earlier verification than is normally possible, because you don't need to produce code in order to check the system's functionality. Unfortunately, UML lacks the precision to properly handle such models.

Aside from UML 1.x's inability to adequately support such technology advances, it presents other significant problems for tool vendors and users. To address these issues, the OMG process requires a major revision of the language. Fortunately, the standardization process for creating UML 2.0 started more than three years ago and is nearing completion. In this article, we look at some major improvements proposed for UML 2.0.[4] The new concepts are not forced on users. Backward compatibility is a strong requirement for UML 2.0, and you should be able to continue using the basic language as you did before, if you desire.

Modeling structure

For most system architectures, a fundamental necessity is specifying their structures. A sound structure makes distributing the system's development and maintaining its integrity easier over time. Many modern programming languages have adopted interfaces as a means to describe the services available from specific classes. Here we distinguish between two kinds of interfaces for classes:

- *Provided interfaces* describe the services that a class implements.
- *Required interfaces* describe the services that others must provide for the class to operate properly in a particular environment.

This distinction supports the development of each class as a standalone entity that does not need to know anything about the entities that implement the required interfaces. UML also allows you to specify provided and required interfaces for components, but because of space limitations, here we focus on the use of interfaces with classes.

A UML class might be used in many different ways, because different stakeholders require different sets of services. Although implementing multiple interfaces on a class might partially express these different uses, developers also need to be able to group interfaces belonging to particular stakeholders. Each such group provides a different view of the class. UML 2.0 realizes this through the *port* construct (see Figure 1). As we will see, these ports play another important role when you assemble classes.

A port connects a class's internals to its environment. It functions as an intentional opening in the class's encapsulation through which messages are sent either into or out of the class, depending on the port's provided or required interfaces. A port that has both provided and required interfaces is *bidirectional*.

Decomposing a system

A particular influence on UML 2.0's use of interfaces has been component-based development, which aims to hierarchically decompose a system into smaller and smaller parts and then connect ("wire") these parts together. Of course, once you've defined a part, you can reuse it in many other contexts. The provided interfaces let you view a class as a black-box component, whose implementation is known only if you look inside it. You can also describe the order in which its services can be invoked by using alternative views (for example, using interactions or state machines) that don't directly expose the internal implementation. Naturally, these choices affect how you describe your structures.

Figure 2 defines a typical compositional structure of an `Automobile` class. The diagram provides a hierarchical view of the parts that make up an automobile.

In contrast, Figure 3a shows how the automobile's parts are connected to each other.

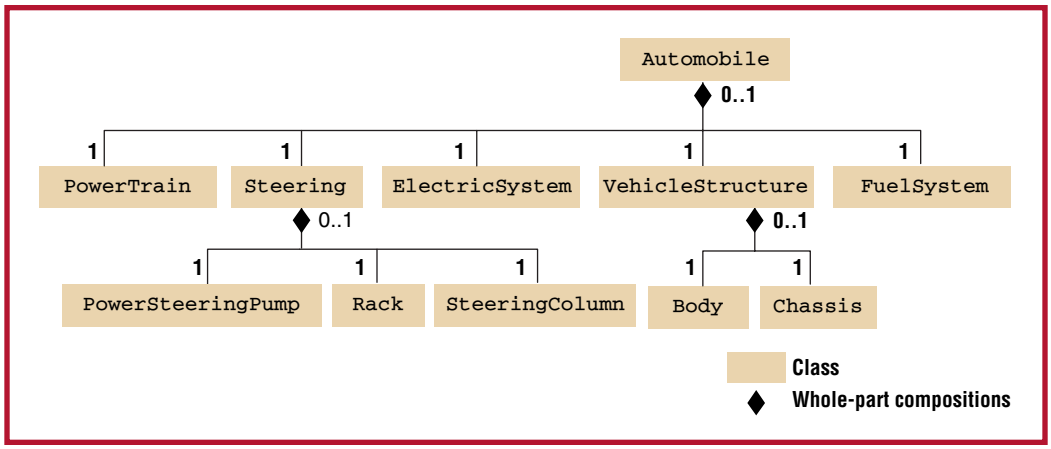


Figure 2. An `Automobile` class's compositional structure.

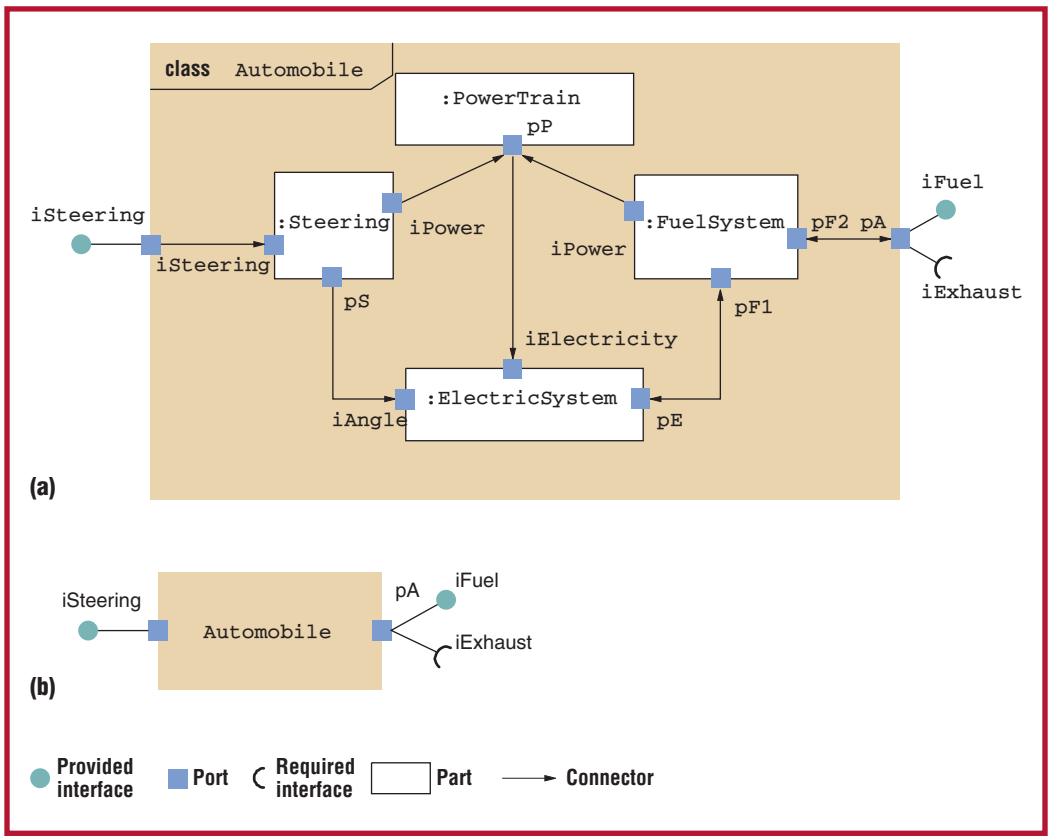


Figure 3. An `Automobile` class's internal structure (some details omitted): (a) white-box view; (b) black-box view.

This white-box view of the class shows the `Automobile` class's different parts—that is, `Automobile`'s implementation. This view of the composition tree focuses on the relationships between parts at the same level and on how *connectors* join them. These connectors describe the communications paths that are valid in this particular context. You can zoom out of this view, which would give you the black-box view of the `Automobile` class (see Figure 3b), or zoom into any of the parts to access its white-box view. The two complementary views let you decompose systems of arbitrary complexity. The class owning the parts is usually called the *container class*; each part represents the use of a class in the context of the container. Another way to think of a part is as a set of instances of a particular type.

This example also shows the second use of ports: as a connection point of a class, through which you can connect parts. The context in which a class is used determines how it interacts with its environment; in each such context it can function as a part that connects to

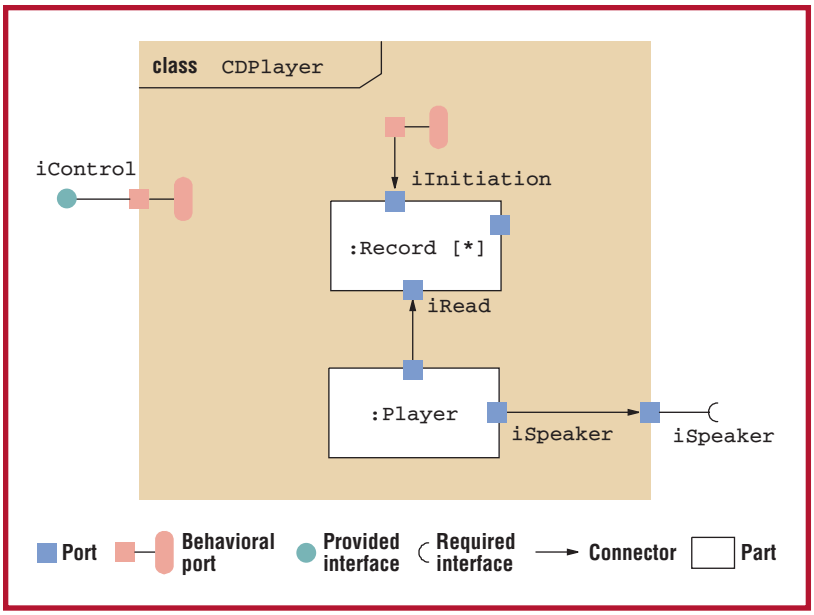


Figure 4. A class's internal structure with behavioral ports.

other parts. In the previous example, the `PowerTrain` class functioned as a part in the context of an `Automobile` class; it would have been connected differently in the context of a `Boat` class.

Adding behavior to the mix

A system does not consist of structure alone. In some cases, as we saw previously, a class's behavior is delegated to its parts, but in the end a class must also include behavior. The class itself might have a behavior, which might be represented through behavioral constructs, such as a state machine, an activity, or an interaction. Also, the methods of its operations might be defined using behaviors. When a class's internal structure consists of parts, the parts commonly communicate with the container class. Just as each part might have a behavior, the container might have behavior describing how to create the parts, initialize them, or route messages, for example.

In a class's black-box view, you cannot know whether a port is attached to the container class's behavior or routed to a part through a connector, because the view doesn't reveal this level of implementation detail. In the white-box view, however, some ports are behavioral ports, as Figure 4 shows. The behavioral ports are attached directly to the container class's behavior and are particularly common when a state machine represents that behavior.

A container class always has some implicit behavior that is never shown because it is part of the semantics of hierarchical decomposition. Whenever an instance of the container class is created, the container instance also creates instances representing its parts. Similarly, whenever an instance of a container class is deleted, that instance deletes its part instances.

Acting on actions

One of the UML 2.0 work's great undertakings has been to integrate *activities* with their related *actions*. (An action is a fundamental unit of behavioral specification that represents some transformation or processing. Actions tend to be general, and apply to all kinds of behavior—for instance, state machines, interactions, or activities. An activity is the basic behavioral construct used for modeling activity diagrams, which specify behavior using a hybrid control and dataflow model.) A few years ago, an initiative began to make UML able to specify executable models. This initiative essentially replaced the previous action model with a new one in which you could express actions with precise procedural semantics. The overlap with activities was glaring, however. UML 2.0 has significantly improved this part of the language; UML actions are now defined in as much as detail as an ordinary programming language's actions (or statements).

The ability to describe system functionality at a higher abstraction level than in an ordinary programming language such as Java or C++ makes it possible to execute UML models. This enables system verification at a much earlier stage in the development life cycle and allows tools to automate tests by using UML models. However, UML cannot serve as a programming language out of the box. First, no notation has been specified for UML's actions, so different tool vendors can specify their own syntax. Second, because UML is intended for many different areas, it has a flexible definition of semantics that must be tightened in some areas or further specified in others. Finally, programmers normally define a set of predefined data types to be used directly in models; UML's *profiles* mechanism is intended to deal with each of these cases.

Being able to execute a model provides the additional benefit that the model becomes independent not only from the platform but also from the target language. Given appropriate transformation rules, you can generate code

for different languages and optimize it for different situations. Because the information in the model lets you generate appropriate optimizations and distributed deployments, the need decreases to focus on these issues when defining the system's functionality.

UML 1.x was never quite able to live up to its original hype, which sometimes portrayed it as a magic solution for all known software or system problems. Although UML 2.0 is likewise not a panacea, its improvements can significantly increase development's efficiency. This increase occurs despite the known flaws introduced by revision, where languages typically suffer from design-by-committee compromises. This is the price we pay for standards and for having different tools that can interoperate.

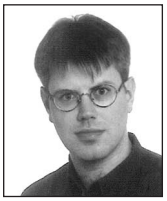
Because UML aims to be suitable for many different audiences, it is a large language. So, it has the concomitant design problem that removing things from a general-purpose language with a large user base is difficult. For example, business process modelers would shun a major revision without activities; likewise, embedded-systems engineers would avoid an update lacking state machines. These two paradigms for expressing behavior are each useful in their own right. However, not everyone needs to learn or apply all of UML, and users are expected to specialize in the areas they need to do their work. To further help in this regard, the language has been subdivided into a number of compliance points, which should make it easier to implement and learn selectively and incrementally.

UML 2.0 work is entering a new phase that focuses more on tuning and bug fixing than adding new features. After completion of the major revision will come updates to modeling tools that implement the powerful new UML 2.0 features. If the tools live up to the new specification's promise, expect significant improvements in how you architect your systems.

References

1. *Unified Modeling Language Specification, Version 1.5*, OMG document formal/03-03-01, Object Management Group, 2003; www.omg.org/technology/documents/formal/uml.htm.
2. *ITU Recommendation Z.100: Specification and Description Language (SDL)*, Int'l Telecommunication Union, 2000.
3. B. Selic, G. Gullekson, and P.T. Ward, *Real-Time Object-Oriented Modeling*, John Wiley & Sons, 1994.
4. *UML 2.0 Superstructure, 3rd Revision*, OMG document ad/03-04-01, Object Management Group, 2003, www.omg.org/cgi-bin/doc?ad/03-04-01.

About the Authors



Morgan Björkander is a senior methods engineer at Telelogic. He is participating in the creation and standardization of UML 2.0, and his major technical interest is model-based application creation. He received his MSc in computer science from the Lund Institute of Technology. He is a member of the IEEE. Contact him at Telelogic AB, PO Box 4128, Kungsgatan 6, SE-203 12 Malmö, Sweden; mbj@telelogic.com.

Cris Kobryn is the chief technologist for Telelogic, where he is responsible for standards leadership, strategic planning, and technology evangelism. His areas of expertise are distributed software architectures, component-based development, and software modeling. As an Object Management Group representative, he has been a major contributor to the Unified Modeling Language specification. He chaired international standardization teams to specify UML 1.1 and UML 2.0, and serves as the cochair of the OMG's Analysis and Design Task Force. He received his BA in geochemistry from Colgate University and his BSCS from San Diego State University. He is a member of the IEEE, ACM, AAAI, and INCOSE. Contact him at Telelogic, PO Box 2320, Fallbrook, CA 92088; cris.kobryn@telelogic.com.