from Describing Software Architecture with UML, Workshop at the 23rd International Conference on Software Engineering, Toronto, Canada, 2001

UML Component Diagrams and Software Architecture—Experiences from the WREN Project

Chris Lüer **David S. Rosenblum**

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
+1-949-824-{2703,6534}
{chl,dsr}@ics.uci.edu

ABSTRACT

In the course of building Wren, a component-based development environment, we encountered several difficulties when we tried to use UML component diagrams for modeling software architectures. One, the semantic interpretation of interfaces and dependencies is not clear in all cases. Two, component diagrams lack the expressive power to model unmet requirements.

Keywords

Component-based software engineering, component diagrams, design methods, software architecture, UML.

1 INTRODUCTION

This paper describes our experiences with the use of UML component diagrams in the Wren project [2]. Wren is a research prototype of a component-based development environment. Important capabilities of the environment include the ability to locate potential components of interest from component distribution sites, to evaluate the identified components for suitability to an application, to incorporate selected components into application design models, and to physically integrate selected components into the application.

Wren makes it possible to connect components (which are defined as encapsulated sets of Java classes) graphically and interactively to an application. We looked at UML component diagrams as one possible notation for the architectural diagrams supported by Wren.

We decided to use a modification of UML component diagrams. The advantage of component diagrams is that they are simple and standardized. Their disadvantages are discussed below; we mitigated these disadvantages by the addition of the concept of ports, and by conventions for the interpretation of the diagrams.

This paper is based on earlier work about modeling software architectures in UML [4].

2 TYPE-ORIENTED AND INSTANCE-ORIENTED DIAGRAMS

Different UML diagrams take a different view on the structure of a system. While some diagrams are semantically equivalent to each other (e.g. sequence diagrams and collaboration diagrams), others are orthogonal (e.g. sequence diagrams and class diagrams).

An important difference among diagrammatic notations lies in the kind of entities they can represent. A sequence diagram, for instance, depicts objects and method calls, i. e. objects and dependencies between objects. A class diagram depicts classes on the one hand, and associations, inheritances, and other dependency relations on the other hand, i. e. classes and dependencies between them.

The distinction between type-oriented diagrams (such as class diagrams) and instance-oriented diagrams (such as sequence diagrams) must also be made on the architectural level.

Deployment diagrams are large-scale instance diagrams provided by UML. They depict processes and machines and their dependencies. While they are important for modeling software architectures in UML, they are not discussed in this paper. Component diagrams are large-scale type-oriented diagrams.

Type- and instance-oriented architectural diagrams complement each other; it is not possible to replace one by the other. Instance-oriented architectural diagrams show the runtime structure of an application. Which processes and threads are involved, where they are physically located, what protocols they use to communicate with each other, how control and data are distributed.

Type-oriented architectural diagrams, on the other hand, deal with *components* and their dependencies. As defined in [7]5, a component is a set of classes, and does not have state; therefore components cannot be shown in instance-oriented diagrams. An example of a component is a class library.

The purpose of a component diagram is to show how

components depend on each other, which components are used in the application, which might be substituted, and which might be missing. When an application developer builds an application out of components, he/she needs to have an overview of the architecture of the application, so that design decisions can be based on it.

3 UML COMPONENT DIAGRAMS

Interpretation of Interfaces and Dependencies

UML component diagrams have three elements: components, interfaces (usually shown in the lollipop notation), and dependencies, which are directed relations between two of the other elements. A typical example is shown in Figure 1: component C1 depends on component C1 via interface I.

There are at least three different possible interpretations of this diagram.

(1) Classes in C1 will use an instance of a class that implements I, and I is defined in C2.

(2) A class in C1 implements I, and I is defined in C2.

(3) Classes in C1 will use an instance of a class in C2 that implements I.

Two issues are in question: the location of the interface I, and the nature of the dependency relation.

Unless one assumes a model where interfaces are global, and available to any component at any time, one has to know the location of an interface definition before one can use it.

Corba is an example of a system with omnipresent interfaces; while this may be an appropriate solution for an object request broker system, in most systems this will not be possible. For example, in Java, interfaces are defined in class files and thus have to be found, loaded, and linked by the runtime environment in the same way that it is done with classes. Even systems that do not have a concept of interface that exists at runtime, such as most C++ implementations, need to be aware of the location of interfaces at least at compile time. A class that implements interface X can typically not be compiled without access to X.

If the diagram in Fig. 1 is assumed to mean that interface I is part of (defined in) component B, interpretations 1 and 3 will be equivalent. This is the preferred interpretation, in our opinion, because otherwise one cannot determine from the diagram where I is located. However, this information is an essential part of the architecture of the system.

The dependency relation in Figure 1 can either mean that A depends on interface I itself, or that it depends on an implementation thereof.

Dependency on the interface itself will be most relevant at compile time, when the compiler needs access to the interface definition in order to compile implementing classes. However, in systems that use dynamic linking, such as Java, the interface definition will have to be accessible at runtime also.

Dependency on an implementation of the interface (without dependency on the interface itself) will not be of interest at compile-time. But the designer has to be aware of it. Assuming that component C1 needs an instance of I to do its task, there will not necessarily be any way a compiler can assure that there is any class in the system that can construct such an instance. Still, the system will not run correctly without such an instance.

In our opinion, both kinds of dependencies should be modeled, unless there exists a convention that makes this unnecessary, as for example the convention that all interfaces are globally accessible.

In simple systems, it will probably often be the case that a component depends both on the interface itself, and on an implementation thereof. However, this constitutes a bad design practice. The purpose of interfaces is to create exchangeability of their implementations; i.e. a component depends only on interface I, instead of depending on class C that implements I, so that class C can be exchanged for another class that also implements I. If a component C1 provides both an interface I (i.e. its definition) and implementations thereof, a component C2 that uses the interface loses the advantage of implementation exchangeabilty, because it cannot exchange the implementation of I without exchanging its definition. This means that putting an interface and classes that implement it into the same component couples this component very strongly to compo-

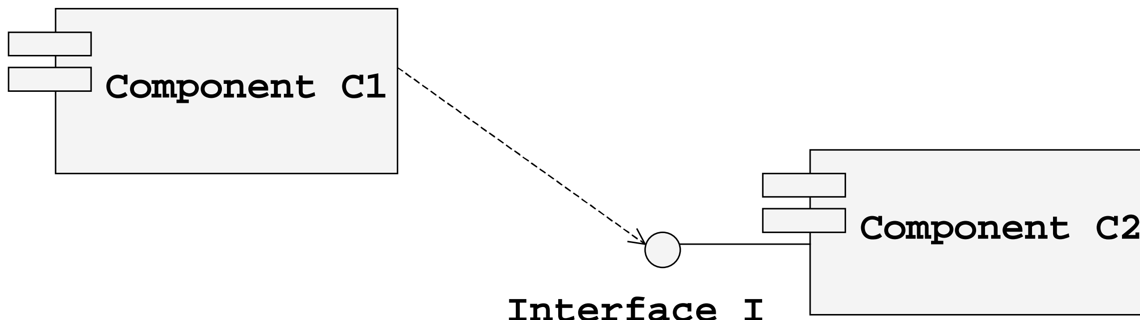

Figure 1: Typical UML Component Diagram.

nents that might use the interface. This is clearly not desirable.

***Requires* and *Provides* Ports**

Dependencies in UML component diagrams typically point from a component to an interface, as above. Most UML tools, for example Rational Rose, allow for relations like dependencies to exist only if they are connected to an entity at both ends. This is appropriate since relations are not considered first-class entities in UML.

However, this makes it impossible to show unmet requirements. While this is unproblematic for class diagrams because classes are usually designed at the same time as their associations, this is an inadequacy with respect to architectural diagrams.

The designer of a component-oriented application will build it iteratively, by taking a small set of components, and then adding more components, possibly rearranging the architecture at the same time. A designer needs to be aware what is missing in the application that is being built; he/she needs to know which components are needed in order to make the existing components work.

For this reason, we propose *requires* ports as a diagrammatic extension to UML component diagrams.

Explicit specification of the features that a component or module requires is well-known from module interaction languages [5] [1], architecture description languages like Darwin [3], and UML for Real-Time [6], for example. Examples of less explicit *requires* specifications are *import* statements in programming languages like C.

Port is a convenient term for handles to the features that a component provides or requires. *Provides* ports are equivalent to interfaces in UML component diagrams, but since there is currently no UML modeling element that corresponds to *requires* ports, we propose the term *port* as a unifying expression.

Figure 2 shows what a UML-like component diagram using both kinds of ports might look like. Requires ports are black circles, provides ports are white circles. Both are labeled with the name of the interface that is required or provided. *Requires* Port Printer in component DisplayBean (lower left corner) is an example of an unmet requirement. It is clearly visible to the designer that he/she has to find a component that implements interface Printer before component DisplayBean can be executed.

4 CONCLUSIONS

- Both type- and instance-oriented diagrams are necessary for modeling software architectures.
- It must be possible to model the location of interfaces, i.e. in which component they are defined.
- Use of an interface and use of an implementation of an interface are two different concepts that must be modeled separately.
- Component diagrams need *requires* specifications or

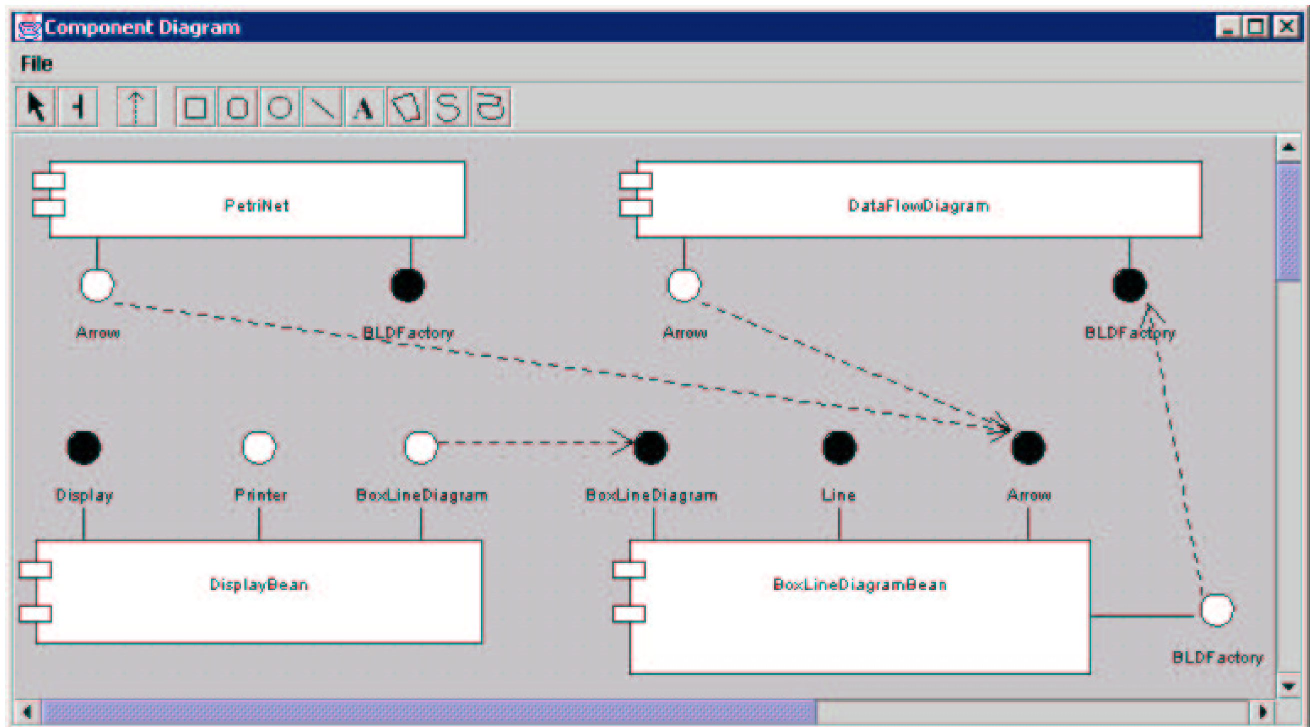

Figure 2: Modified Component Diagram. *Requires* ports are white, *provides* ports are black.

a similar mechanism to show what components are missing in an application.

- The concept of ports unifies *requires* specifications with provided interfaces.

REFERENCES

1. DeRemer, F. and Kron, H.H. Programming-in-the-Large Versus Programming-in-the-Small. *IEEE Transactions on Software Engineering*, *2* (2). 80-86, 1976.
2. Lüer, C. and Rosenblum, D.S. Wren—A Component-Based Development Environment, Dept. of Information and Computer Science, University of California, Irvine, Technical Report UCI-ICS-00-28, 2000.
3. Magee, J. and Kramer, J. Dynamic Structure in Software Architectures. *Software Engineering Notes*, *21* (6). 3-14, 1996.
4. Medvidovic, N., Rosenblum, D.S., Robbins, J.E. and Redmiles, D.F. Modeling Software Architectures in the Unified Modeling Language. *ACM Transactions on Software Engineering and Methodology*, *to appear*, 2001.
5. Prieto-Diaz, R. and Neighbors, J.M. Module Interconnection Languages. *Journal of Systems and Software*, *6* (4). 307-334, 1986.
6. Selic, B. and Rumbaugh, J. Using UML for Modeling Complex Real-Time Systems, Rational Software Corporation, 1998.
7. Szyperski, C. *Component Software*. ACM, New York, 1997.