Modeling Component Systems with the Unified Modeling Language

—A position paper for ICSE'98 Workshop

Philippe Kruchten
Rational Software Corp.
240-10711 Cambie Road
Richmond, BC V6X 3G5 Canada
pkruchten@rational.com
+1 (604) 231 3706

The Rational Process supports *component-based development*, both in terms of *representation* of component-based systems, using UML, and the actual *workflow*, i.e., activities and step by step guidance on how to model then to build them. The goal of the process is to enable software development organizations to rapidly build and deploy component-based systems. Often, component-based is immediately associated with a specific technology: CORBA, Microsoft ActiveX / COM / DCOM, JavaBeans (Enterprise and otherwise), etc. But we would like to propose techniques and tools that are *independent* from a specific technology, but still practical enough to be instantiated in any of these technologies or emerging technologies. This paper gives our definition of component and component-based development, and then goes on to shows how to represent component based system using the *Unified Modeling Language* (UML).

1. Definitions—What is CBD?

Component

For component, we need a definition broad enough to address conventional components (such as COM / DCOM, CORBA and JavaBean components) as well as alternative ones (web pages, data base tables, and executables using proprietary communication), yet not so broad as to encompass every possible artifact of a well-structured architecture.

> A *component* is a non-trivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces.

A closer examination of this definition is warranted. First, a component is *non-trivial*; it is functionally and conceptually larger than a single class or a single line of code. Typically, a component encompasses the structure and behavior of a collaboration of classes.

Second, a component is *nearly independent* of other components. It rarely stands alone. A given component collaborates with other components and in so doing assumes a specific architectural context. This architectural context is driven in large part by the implementation we chose.

Third, a component is a *replaceable part of a system*. A component is substitutable for any other component which realizes the same interfaces. This aspect helps during development, where parts of a system can be stubbed, sketched, then replaced by mature, robust implementations. It also supports the evolution of a system, once deployed by making it possible to upgrade and evolve parts of the system independently.

Fourth, a component *fulfills a clear function.* A component is logically and physically cohesive, and thus denotes a meaningful structural and/or behavioral chunk of a larger system. It is not jut some arbitrary grouping.

Fifth, a component exists *in the context of a well-defined architecture.* A component represents a fundamental building block upon which systems can be designed and composed. This definition is recursive: a system at one level of abstraction may simply be a component at a higher level of abstraction. Components never stand alone, however. Every component presupposes an architectural and technology context wherein it is intended to be used.

Finally, a component *conforms to a set of interfaces.* A component that conforms to a given interface means that it satisfies the contract specified by that interface and may be substituted in any context wherein that interface applies.

Interface

> An *interface* is a collection of operations that are used to specify a service of a component.

An interface serves to name a *collection of operations* and specify their signatures and protocols. An interface focuses upon the behavior, not the structure, of a given service. An interface offers no implementation for any of its operations.

An interface is used for *specifying a service.* An interface gives a name to a collection of operations that work together to carry out some logically interesting behavior of a system or a part of a system.

An interface defines a service offered *by a component (or a class).* An interface defines a service that is in turn implemented by a class or a component. As such, an interface spans the logical and physical boundaries of a system. One or more classes (which are likely a part of some component subsystem) may provide a logical implementation of a given interface; one or more components may provide a physical packaging that conforms to that same interface.

Component-Based Development

> *Component-based development (CBD)* is the creation and deployment of software-intensive systems assembled from components, as well as the development and harvesting of such components.

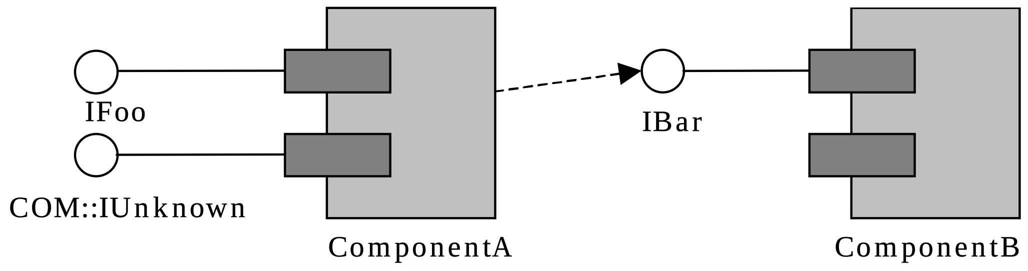
CBD is about building quality systems that satisfy business needs quickly.

CBD is about building systems out of parts more than handcrafting every individual element. CBD involves crafting the right set of primitive components from which to build families of systems and it includes *the harvesting of components*. Some components are intentionally made; others are discovered and adapted.

2. Representing Components and Interfaces using UML

Components and Interfaces

Components, as discussed above, are primarily a run-time concept. They offer interfaces (specifications of behavior) and are dependent only on other interfaces. In UML, a component is represented as:



The diagram shows a component, ComponentA, which realizes interface IFoo (realization is shown by the solid line from ComponentA to IFoo). The interface IFoo is contained in a package called COM, hence the name is fully qualified with the package name as well. The naming convention adopted by Microsoft's COM (prefixing the name of the interface with 'I') is adopted here but is not required by UML. Realization of an interface by a component means that the component offers the operations defined by the interface. A component may realize any number of interfaces.

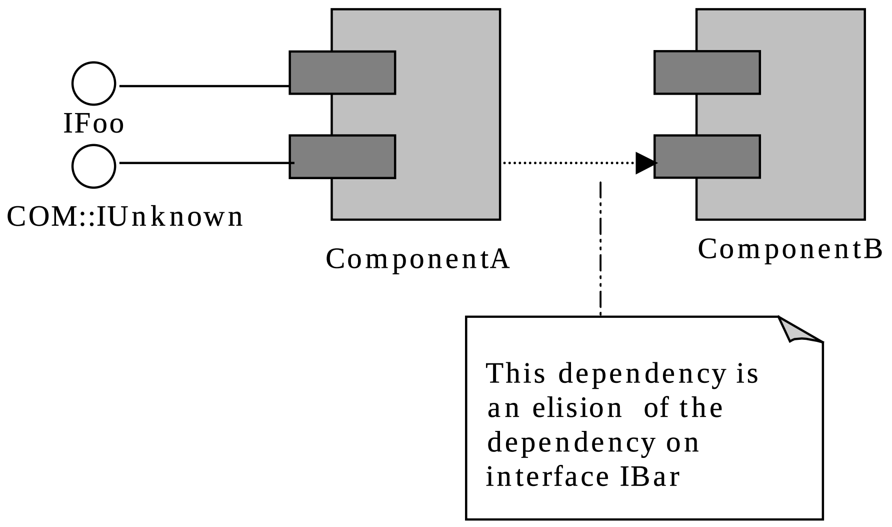
The component is also dependent on the interface IBar, as shown by the dashed line. Dependency of a component upon an interface means that the component requires the services of other components which realize the interface. A component may be dependent upon any number of interfaces.

Expressing dependencies in terms of interfaces rather than specific components provides one of the key benefits of CBD: the substitutability of components which realize the same interface. Interfaces allow complete separation of specification from implementation.

Implicit Component Dependencies

It is often convenient to speak of dependencies between components. Although in a pure sense, components are only dependent on interfaces, it is simpler in many cases to speak of components being dependent on one another, although what we really mean is that 'a component is dependent on an interface realized by another component'.

For simplification and visualization of component dependencies, the following depictions are equivalent:

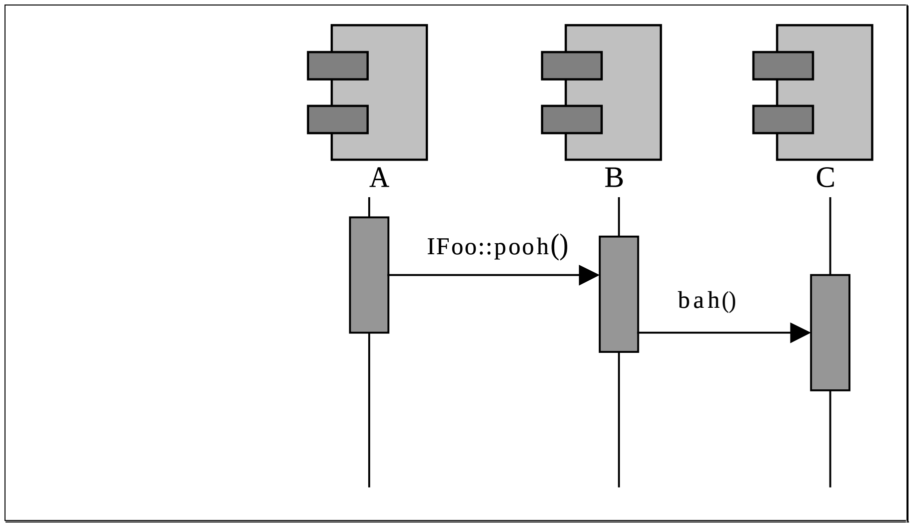

IFoo
COM::IUnknown
ComponentA
ComponentB
This dependency is an elision of the dependency on interface IBar

If component A is dependent of interface I, and component B realizes interface I, there is an elided dependency between A and B.

Component Dynamics

Representing components and their dependencies captures only the static nature of components. Assembling systems based on components requires understanding and depiction of the behavior of components.

Component dynamics can be shown using *sequence diagrams,* as shown below:



This diagram depicts a set of interactions of components A, B and C. Vertical bars represent the focus of control of the components. Messages (horizontal arrows) represent invocation of operations on the interfaces realized by the components. The vertical dimension of the diagram represents time. Messages can be labeled with the specific operation being invoked; if the operation name is ambiguous or the component realizes more than one interface, the operation name may be qualified with the name of the interface.

A message from component A to component B implies a dependency between A and B. Technically, components are only dependent upon interfaces of other components. The implicit dependency of two components A and B can be regarded as an elision of the dependency between a component A and the interface realized by component B.

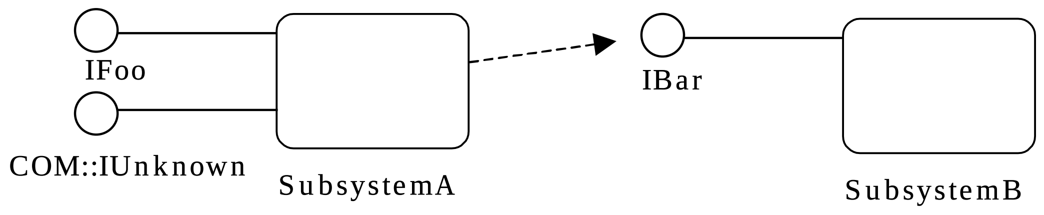
Depicting the interaction of components using sequence diagrams eases the creation of new systems by assembling existing components and viewing their interactions visually.

Designing with Components

Despite our best intentions of building new systems only out of existing components, there is usually a need to either modify an existing component or to create new components. Since components are run-time elements, we need a way to represent the design perspective of a component.

The design perspective of a component encompasses more than a single class: it represents a number of classes which interact to provide a set of services. In UML, this can be represented as a subsystem: a type of package which realizes one or more interfaces. To clarify use of a subsystem for particular use as the design representation of a component, the UML stereotype «component subsystem» can be applied. There is typically a 1:1 relationship between components and component subsystems, though for complex designs, subsystems may be nested to represented composite components.

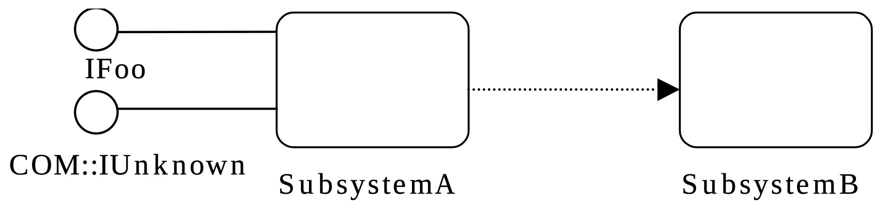
Subsystems are depicted as follows:



Like a component, a subsystem realizes one or more interfaces and is dependent on zero or more interfaces. It cannot be directly dependent on other subsystems, except in the form of an elided dependency.

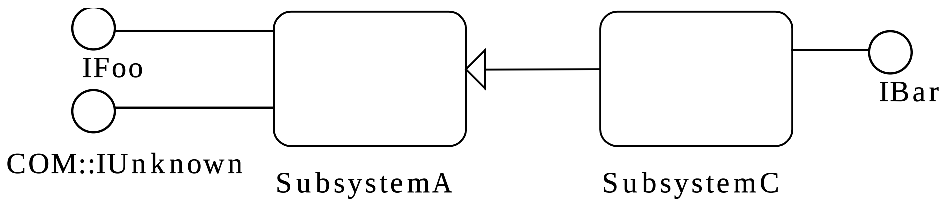
The visibility rules on subsystems are simple: it completely encapsulates its contents. Like a component, the only behavior a subsystem offers is that of the interfaces it realizes; there is no other way to invoke functionality on a subsystem than through its interfaces. This is important, as it mirrors the semantics of components: in order to achieve true substitutability of components realizing the same interface, only the interface should be visible when the component is represented in design.

As with components, it is often convenient to elide the dependency relationship, showing the implied dependency between subsystems:



Generalization of Subsystems

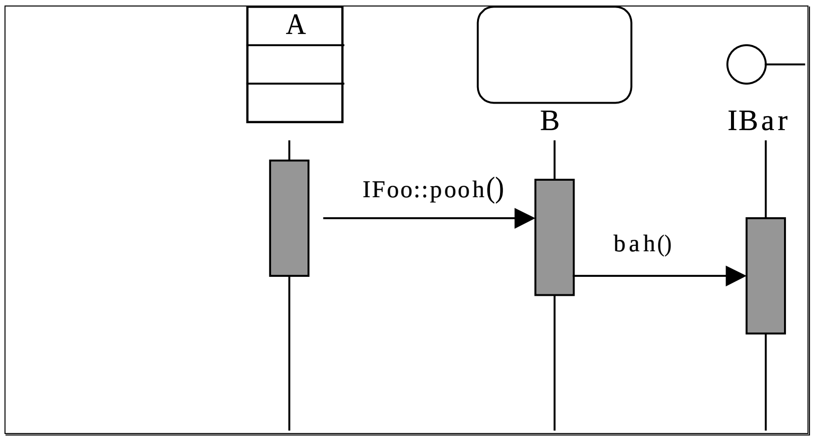
A subsystem may also be generalized from other subsystems. The practical use of this is to depict implementation inheritance or component specialization.



In the figure above, Subsystem realizes IFoo and COM::IUnkown as well as IBar. Rather than creating a new subsystem which realizes the same interfaces, has the same dependencies, and possesses the same contents, we can inherit these from an existing subsystem and then specialize its behavior in the new subsystem. As an example, we may want to create a new kind of TransactionManager component which adds functionality to an off-the-shelf transaction manager.

Subsystem Dynamics

Formally, subsystems in UML as classifiers, as are classes. As a result, they can be used generally anywhere a class can be used: in class diagrams, as the end-points of associations from classes, and more importantly, in diagrams depicting system dynamics, such as sequence diagrams:



The figure above depicts a typical depiction of the interaction of classes, components and interfaces. Classes can send messages to other classes, just as always, but now we can show invocation of behavior on a subsystem, or the exercise of an interface.

Interface Realization Semantics

Even though a subsystem may realize an interface, it is the contents of the subsystem which really carry out the behavior specified by the interface; subsystems have no behavior on their own.

When an interface is realized by a subsystem, it implicitly means that some model element within the subsystem actually realizes one or more of the operations of the interface. In the simple case, the mapping is 1:1 between some contained class and an interface realized by the subsystem; this mapping is expressed as a realization association between the contained class and the interface. A realization association between a class and an interface means that for every interface operation there is a compatible operation on the class. Compatibility means that the signatures match, although the class operation can have additional parameters if they are null or defaulted.

A single interface can also be realized by more than one class. In this case, there is a realization association between methods of the class and operations of the interface. Though more complicated, this allows for N:M mapping between operations on interfaces and classes.

Conclusion

This is a snapshot of a work in progress within Rational Software Corp. Much more work is necessary to refine the definitions and the mapping to UML, as well as describing the specific workflow, activities and steps related to CBD within the Rational Process.

Acknowledgments

For their help in shaping the ideas we presented in this paper, I thank my many colleagues at Rational, in particular Kurt Bittner and Grady Booch.

References

1. *Rational Objectory Process*, version 4.1, Rational Software Corp., Cupertino, Ca, 1997.
2. *Unified Modeling Language*, version 1.1, OMG, 1997. See also http://www.rational.com/uml/
3. Ivar Jacobson, Martin Griss, Patrik Jonsson, *Software Reuse*, Addison-Wesley, 1997.
4. Alan Brown (ed.), *Component-Based Software Engineering*, IEEE Computer Society, 1997.
5. David Chappell, *Understanding ActiveX and OLE—A guide for developers and managers,* Microsoft Press, 1996.