Documenting Component and Connector Views with UML 2.0

James Ivers
Software Engineering Institute

Paul Clements
Software Engineering Institute

David Garlan
School of Computer Science, Carnegie Mellon University

Robert Nord
Software Engineering Institute

Bradley Schmerl
School of Computer Science, Carnegie Mellon University

Jaime Rodrigo Oviedo Silva
School of Computer Science, Carnegie Mellon University

April 2004

TECHNICAL REPORT
CMU/SEI-2004-TR-008
ESC-TR-2004-008

Carnegie Mellon
Software Engineering Institute

Pittsburgh, PA 15213-3890

Documenting Component and Connector Views with UML 2.0

CMU/SEI-2004-TR-008
ESC-TR-2004-008

James Ivers
Software Engineering Institute

Paul Clements
Software Engineering Institute

David Garlan
School of Computer Science, Carnegie Mellon University

Robert Nord
Software Engineering Institute

Bradley Schmerl
School of Computer Science, Carnegie Mellon University

Jaime Rodrigo Oviedo Silva
School of Computer Science, Carnegie Mellon University

April 2004

Software Architecture Technology Initiative

This report was prepared for the

SEI Joint Program Office
HQ ESC/DIB
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

Christos Scondras
Chief of Programs, XPK

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number F19628-00-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Table of Contents

List of Figures

List of Tables

Acknowledgements

We are indebted to those with whom we collaborated on earlier incarnations of this work. Andrew J. Kompanek, Shang-Wen Cheng, and David Garlan began with the paper “Reconciling the Needs of Architectural Description with Object-Modeling Notations” [Garlan 02]. Felix Bachmann, Len Bass, Reed Little, Judy Stafford, Paul Clements, David Garlan, James Ivers, and Robert Nord revisited this topic for the book titled *Documenting Software Architectures: Views and Beyond* [Clements 02]. Bran Selic answered numerous questions and helped us understand the implications of some of the UML 2.0 changes. Felix Bachmann, Linda Northrop, and Reed Little provided valuable feedback to improve the quality of this report.

Abstract

The widespread presence of the Unified Modeling Language (UML) has led practitioners to try to apply it when documenting software architectures. While early versions of UML have been adequate for documenting many kinds of architectural views, they have fallen somewhat short, particularly for documenting component and connector views. UML 2.0 has added a number of new constructs and modified some existing ones to address these problems. In this report, we explore how changes in this version affect UML's suitability as a notation for documenting component and connector views.

1 Introduction

Because architectures are intellectual constructs of enduring and long-lived importance, communicating an architecture to its stakeholders becomes as important a job as creating it in the first place. If others cannot understand an architecture well enough to build systems from it, analyze it, maintain it, or learn from it, all the effort put into crafting it is, by and large, wasted. Therefore, attention is now being paid to how architectural information should be captured in enduring and useful artifacts—that is, how it should be documented. Major trajectories in this work include the SEI's "Views and Beyond" approach for documenting software architectures [Clements 02], the IEEE 1471 recommended best practice for documenting architectures of software-intensive systems [IEEE 00], and Version 2.0 of the Unified Modeling Language (UML) [OMG 03].

1.1 Views and the "Views and Beyond" Approach

Modern software architecture practice embraces the concept of architectural *views* [Kruchten 01], [Kruchten 95], [Hofmeister 00]. A view is a representation of a set of system elements and the relations associated with them [Clements 02]. Views are representations of the many system structures present simultaneously in software systems. The complexity of modern systems make them difficult to grasp all at once. Instead, we restrict our attention at any one moment to one (or a small number) of the software system's structures, which we represent as views. Different views often carry different burdens with respect to a system's quality attributes. For instance, a view that shows a system structured as layers can be used to engineer certain kinds of modifiability (such as platform portability) into the system and can also be used to communicate that facet of the system's design to others. By contrast, a view showing how the same system is structured as a set of communicating processes can be used to engineer certain kinds of performance into the system and can also be used to communicate *that* facet of the system's design to others. Documenting a software architecture involves choosing and documenting the relevant views of that architecture and then documenting information that applies to more than one view. The relevant views are the ones that best serve the needs of the architecture documentation's most important stakeholders.

In fact, clearly documenting the views that best serve stakeholders' needs is the central tenet of the "Views and Beyond" approach [Clements 02]. In addition, the approach identifies a set of three fundamental view categories called *viewtypes:* (1) module, (2) component-and-connector (C&C), and (3) allocation. Module views show how a software system is structured as a set of implementation units, component-and-connector views show the system as a set of cooper-

ating units of runtime behavior, and allocation views show how the software is allocated to structures in the system's environment. The approach also recommends templates for documenting views and the information that applies beyond them, and a simple stakeholder-centric method for choosing the best views to document.

Like the "Views and Beyond" approach, the IEEE 1471-2000 standard recommends choosing those views that best serve the needs of the architecture's many stakeholders [IEEE 00]. By contrast, the Rational Unified Process (RUP) and similar approaches prescribe a predefined set of views for all systems [Kruchten 01].

1.2 UML and Software Architecture

The widespread presence of UML has led practitioners to apply it to the task of documenting software architectures. The results have been mixed and somewhat disappointing. For example, UML has no built-in way to represent the basic architectural concept of a *layer*. It also lacks a straightforward way of representing a *connector*, in the rich sense proposed by Shaw and Garlan [Shaw 96 , Ch. 7]. The modeling element (*component*) provided by UML 1.4[1] was more of an implementation concept than an architectural one. Nevertheless, because few other commercially accepted standards for architectural documentation exist, ways have been proposed to represent the graphical portion of several familiar architectural views using UML. Representative approaches include those that use UML "as is" and those that specialize UML to improve its suitability for architectural documentation [Selic 98], [Garlan 02], [Medvidovic 02]. Clements and associates explored several common notations for showing the elements and relations in 15 of the most familiar views and, in most cases, showed how UML 1.4 could be applied (or shoehorned) into helping with the task [Clements 02].

Most of the shoehorning took place in the area of representing components and connectors—the primary elements that occur in views in the component-and-connector viewtype. A *component* is a principal unit of runtime interaction or data storage; a *connector* is an interaction mechanism among components. For example, in a pipe-and-filter view, filters are components, and pipes are the connectors. In a shared-data view, the data repository and the accessors are the components, and the access mechanisms are the connectors. In a client-server view, the components are clients and servers, and the connectors are the protocol mechanisms by which they interact.

1.3 About This Report

This report revisits the work of Clements and associates on documenting C&C views using UML [Clements 02] and explains how the changes in UML 2.0 improve its suitability for

1. The statements made in this report referring to UML 1.4 [OMG 01] are generally applicable to all the 1.x versions of UML.

architectural documentation and how it still falls short. Section 2 reviews the options for documenting C&C views using UML 1.4 to provide a basis for comparison with the changes in UML 2.0. Though we assume readers are familiar with UML concepts, we do not assume they are familiar with the changes in UML 2.0. Those changes that affect UML's suitability for documenting software architecture are presented in Section 3. Section 4 presents the options for documenting C&C views using UML 2.0, and Section 5 summarizes how that language supports documenting those views and the support it still lacks.

2 Documenting C&C Views with UML 1.4

Before discussing how to document a C&C view, we need to review what needs to be documented [Clements 02]:

- components and component types: representing the principle runtime elements of computation and data storage such as clients, servers, filters, and databases
- connectors and connector types: representing the runtime pathways of interaction between components such as pipes, publish-subscribe buses, and client-server channels
- component interfaces (or ports): representing points of interaction between a C&C component and its environment. A given component may have many such interfaces, even some that provide or require identical services.
- connector interfaces (or roles): representing points of interaction between a C&C connector and the components to which it is connected. A given connector may have many such interfaces, even some that provide or require identical services.
- systems: graphs representing the components and connectors in the system and the pathways of interaction among them
- decomposition: a means of representing substructure and selectively hiding complexity. A component that appears as a single entity at one level of description might have an internal architectural structure that provides more detail on its design.
- properties: additional information associated with structural elements of an architecture. For C&C views, such properties typically characterize attributes (such as execution time or thread priority) that allow you to analyze the performance or reliability of a system.
- styles:[2] defining a vocabulary of component and connector types together with rules for how instances of those types can be combined to form an architecture in a given style. Common styles include pipe-and-filter, client-server, and publish-subscribe. (See the work of Clements and associates [Clements 02] or Shaw and Garlan [Shaw 96] for more details.)

2. Architectural styles are sometimes referred to as *architectural patterns*. We use the term *architectural style* to avoid any potential confusion with design patterns, such as those used in the object-oriented community [Gamma 95], which are often not architectural.

2.1 Documenting Components and Connectors with UML 1.4

Clements and associates [Clements 02, p. 148-159] present a variety of choices (based on Garlan, Cheng, and Kompanek's work [Garlan 02]) for using UML 1.4 to represent concepts of C&C views. Each choice has advantages and drawbacks, and none of them can be said to have the advantage of intuitive simplicity. To draw a contrast between UML 1.4 and UML 2.0 in this area—in particular, to show how UML 2.0 has simplified the picture considerably—this section summarizes the key choices when using UML 1.4.

Because components are the primary computational elements of a C&C view of a software architecture, they feature prominently in architectural documentation. In fact, how you choose to represent various C&C concepts, such as connectors and ports, often depends on how components are represented. Consequently, for the first choice that must be made—how to represent C&C components—the documentation options for using UML are organized around three broad strategies:

1. UML classes and objects
2. UML subsystems
3. the UML Real-Time (UML-RT) profile

At first glance, it seems like a glaring omission to ignore UML components because, in fact, they are quite similar to C&C components. UML components have interfaces, may be deployed on hardware, and are visually distinct from classes. UML components are often used in diagrams that depict an overall topology, and just as architectural components are mapped to hardware, components are assigned to nodes in UML deployment diagrams.

However, in UML 1.4, components are defined as concrete chunks of implementation—for example, executables or dynamic link libraries—that realize abstract interfaces. In the C&C viewtype, the notion of a runtime component is more abstract, frequently having only an indirect relationship to a concrete unit of deployment. While architectural components in some systems may ultimately be realized as components in the UML sense, in many cases, their semantics simply do not match.

2.1.1 Strategy 1: Using Classes and Objects for C&C Components

A natural candidate for documenting component types in UML 1.4 is the class concept. Classes describe the conceptual vocabulary of a system just as component types form the conceptual vocabulary of an architectural description in a particular style. Additionally, the relationship between classes and objects is similar to that between architectural types and their instances. The full set of UML descriptive mechanisms is available to describe the structure, properties, and behavior of a class, making this strategy a good choice for depicting detail and using UML-based analysis tools. Properties of architectural components can be represented as class attributes or associations, behavior can be documented using UML behavioral models, and generalization can be used to relate a set of component types. The semantics of an instance or type can also be elaborated by attaching a standard stereotype; for example, the “process” stereotype can be attached to a component to indicate that it runs as a separate process.

With this strategy, you must make additional choices about how to document other C&C architectural concepts:

- Ports can be documented in five ways: (1) not at all, (2) as annotations, (3) as attributes, (4) as UML interfaces, or (5) as classes.
- Connectors can be documented in three ways: (1) as associations, (2) as association classes, or (3) as classes.
- Systems can be documented in three ways: (1) as UML subsystems, (2) as contained objects, or (3) as collaborations.

The choices available for documenting each concept vary in how much detail can be captured, the degree of semantic match between the corresponding UML and C&C concepts, and how each UML concept fits visually with the other choices. Consequently, while no one set of choices is appropriate for every system, each is a reasonable choice for some. See the work of Clements and associates for more specific information on the various choices [Clements 02].

2.1.2 Strategy 2: Using Subsystems for C&C Components

The second strategy for using UML to document components is to use UML subsystems, which are stereotyped packages. This approach is appealing because packages are an ideal way to describe coarse-grained elements as a set of UML models and because UML modelers are already familiar with the package construct as a way of bundling portions of a system. The subsystem construct is used in UML to group, or encapsulate, a set of model elements that describe a logical piece of a system, similar to components in architectural descriptions. Subsystems—indeed, any UML package—can include structures based on any UML model.

The advantage that identifying a component with a package has over documenting components as classes is that the former allows us to represent substructure using either classes (or objects) and behavioral models. This approach also has a visual appeal; substructure can be depicted as "embedded" in the package. Components and component types can be documented in essentially the same way—as packages—although you could also take advantage of the UML template mechanism when defining a type.

However, using subsystems to document components can be problematic. In UML, a subsystem has no behavior of its own, so all the communications sent to a closed subsystem must be redirected to instances inside that subsystem, and UML leaves that redirection unspecified as a semantic variation point. Another problem is that subsystem interfaces have the same set of issues as class interfaces (i.e., it is impossible to represent several interfaces of the same type on the same subsystem). A third problem is that representing substructures, such as ports, as elements contained by a subsystem is arguably counterintuitive. The fact that certain elements correspond to ports, others to properties, and others to hierarchical decompositions is likely to be misleading.

This scheme provides two natural choices for representing connectors: as dependencies (visually simple but lacking expressiveness) or as subsystems themselves. Dependencies have visual appeal but do not provide a way to define more detailed aspects of a connector. Using subsystems to document connectors (which is similar to using objects or classes to document connectors in the previous strategy) may be problematic because components and connectors are not readily distinguishable.

2.1.3 Strategy 3: Using the UML-RT Profile

The third strategy is to document C&C views using a profile, or specialization, of UML. A UML profile is a collection of stereotypes, constraints, and tagged values that can be bundled to form a domain-specific language specialization. UML-RT is a profile originally developed by the telecommunication industry to meet its software development needs [Selic 98]. The stereotypes defined in UML-RT provide a close semantic match for the architectural concepts documented as part of a C&C view.

In UML-RT, the primary unit for encapsulating computation is the capsule. Capsules can have interfaces and be hierarchically decomposed. Component types are documented using UML capsule-stereotyped classes; component instances are documented using capsule-stereotyped objects in a collaboration diagram.

C&C connectors are documented using UML-RT connectors because both represent interactions between the computational units. Connectors are documented using UML association classes, and connector instances are documented using UML links—instances of UML associations. UML-RT protocols represent the behavioral aspects of UML-RT connectors.

Likewise, UML-RT defines stereotypes that can be used effectively to document other C&C concepts such as ports, roles, and systems. Table 1 summarizes the relationship between UML-RT and the corresponding architectural concepts.

Table 1: Mapping Between C&C Concepts and UML-RT Constructs

Architectural Construct	UML-RT Construct
Component	<<Capsule>> instance
Component type	<<Capsule>> class
Port	<<Port>> instance
Port type	<<ProtocolRole>> class
Connector	<<Connector>> link
Connector type	association class
Connector's behavioral constraint	<<Protocol>> class
Role	No explicit mapping; implicit elements: LinkEnd
Role type	AssociationEnd
System	Collaboration

2.2 Shortcomings of Using UML 1.4

As we have seen, UML 1.4 provides many possible ways to document software architectures. In particular, there are several natural modeling constructs for components and component types. Connectors, however, are problematic because UML 1.4 does not have a first-class concept for representing them directly. Instead, connectors have to be encoded as associations or represented as components themselves. Interfaces are also problematic, since the interface concept in UML 1.4 does not allow the representation of multiple runtime points of interaction, many of which might have the same "signature" (and hence, the same UML 1.4 interface). Representing the hierarchical decomposition of components is also difficult because there is no way to provide a more detailed architectural model scoped to a single component and indicate that, when a component instance is created, it must have the indicated substructure. Finally (although we did not discuss this above), there is no specific construct within UML 1.4 for representing architectural styles, although you could use UML profiles to achieve a similar effect. UML-RT corrects some of these problems through a special profile and set of visual conventions that provide a more direct way to represent architectural structure. However, because the profile is not "core" UML, it is not as well understood or supported by tools.

3 UML 2.0 Changes

UML 2.0 introduces a number of new concepts and refines a number of existing ones.[3] This section provides a brief overview of those concepts most relevant to documenting C&C views. Our goal is not to define these concepts completely—the UML standard is a much better source for that—but, instead, to provide a basic description of them and how they fit together as a prelude to describing, in Section 4, how they can be used to document C&C views. Though some of these concepts can also be used to document other architectural views more effectively (e.g., structured classifiers improve the documentation of decomposition views in UML), in this report, we focus on their applicability to C&C views.

This overview explains UML concepts with minimal reference to the UML metamodel (the core of the formal language definition). However, in order to present the relationships among some UML concepts clearly, we must describe certain aspects of the metamodel. In particular, one concept from the metamodel that is used repeatedly is the *classifier*—a type that can have instances (e.g., a class is a kind of classifier whose instances are objects). This concept is important because two UML concepts that we discuss—classes and components—are both classifiers. Consequently, when we describe how concepts apply to classifiers, we are describing how the concepts apply to both classes and components.

Five areas of UML 2.0 warrant our attention, each of which is addressed below:

1. interfaces
2. ports
3. structured classifiers
4. components
5. connectors

3. As of the publication date of this report, UML 2.0 has not yet been released as an official standard. However, a version of the standard has been endorsed by the Object Management Group (OMG) as the final adopted specification, and significant changes are not expected during the finalization process.

3.1 Interfaces

UML 2.0 extends the interface concept to explicitly include *provided* and *required* interfaces. Interfaces from UML 1.4 are provided interfaces, describing the features (e.g., operations with public visibility) that constitute a coherent service provided by a classifier. Required interfaces were introduced to complement provided interfaces and describe the features that make up a coherent service that a classifier depends on in order to provide its functionality. Additionally, interfaces can now include attributes and be augmented with behavioral descriptions called *protocol state machines*. These descriptions define usage constraints (protocols) among the features of the associated interface.

Figures 1 - 3 show three different ways to represent the same interfaces in UML 2.0. The first way, shown in Figure 1, is the most descriptive. Interfaces are defined using a stereotyped classifier with compartments (much like a class) in which the interface's features are defined. In this representation, provided and required interfaces are not distinguished in the classifier itself but rather by the relations between the interface and another classifier. A dependency relation from a classifier to an interface denotes a required interface of that classifier (e.g., in Figure 1, Database is a required interface of Server). A realization relation between a classifier and an interface (with the triangle pointing to the interface) denotes a provided interface that is realized by the classifier (e.g., in Figure 1, HTTPRequest is a provided interface of Server).[4]

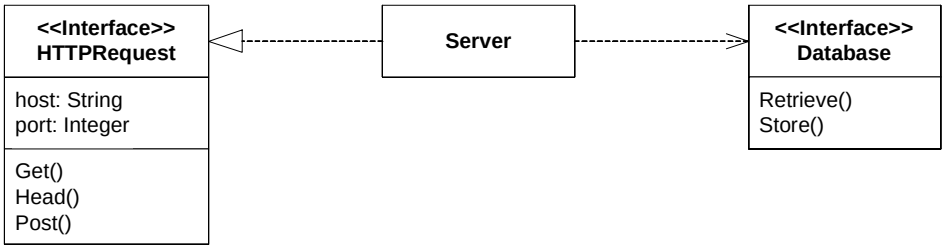

Figure 1: Interfaces as Stereotyped Classifiers

Figure 2 shows a form in which the provided and required interfaces of a classifier are listed in a compartment of the classifier. Each interface is identified as provided or required by use in a list of interfaces preceded by the appropriate stereotype: <<provided interfaces>> or <<required interfaces>>.

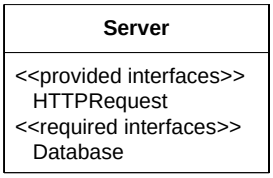

Figure 2: Interfaces in a Classifier Compartment

4. Required interfaces could be documented in UML 1.4 using dependencies, as shown in Figure 1, but the semantics were not as clear as those found in UML 2.0. For example, because UML 2.0 defines required interfaces as a first-order concept, it can define interface compatibility in terms of matching required interface signatures to provided interface signatures.

Figure 3 shows a form that represents interfaces in a graphically concise form useful for connecting classifiers. The ball-and-socket notation shows provided interfaces as balls at the end of a line (or "lollipops") and required interfaces as sockets. This notation labels the interfaces but does not list their features.

Figure 3: Interfaces Using Ball-and-Socket Notation

3.2 Ports

A *port*, a new concept in UML 2.0, is similar to an interface in that it describes how a classifier interacts with its environment, but is different in that each port is a distinct interaction point of its classifier. Ports can have types, and a classifier can specify the multiplicity of a port. When the classifier is instantiated, the corresponding number of port instances are created, each distinguishable from others of the same type.

Each port can be associated with a number of interfaces, provided and/or required, in whatever collection makes sense for the point of interaction. Like interfaces, ports can be associated with behavioral descriptions (protocol state machines) that define usage constraints. Unlike interface behavioral descriptions (which are restricted to the features of the interface), a port behavioral description can be used to show usage constraints among the features of all the port's interfaces.

Figure 4 shows a simple port (without associated interfaces). Port p is shown as a rectangle on the border of the Server classifier and has a 1..* multiplicity, indicating that each instance of Server will have one or more p ports. Figure 5 extends this example to show the association of specific interfaces (request and response) with the port.

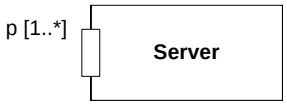

Figure 4: Port Without Interfaces

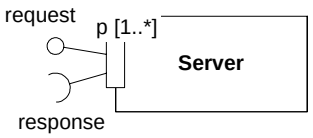

Figure 5: Port with Interfaces

3.3 Structured Classifiers

UML 2.0 provides a new way to represent the internal structure (decomposition) of classifiers. A *structured classifier* is defined, in whole or in part, in terms of a number of contained instances owned or referenced by the structured classifier. Those contained instances that are owned (with a similar meaning to a composition relation) are called *parts*. A structured classifier's parts are created within the containing classifier (either when the structured classifier is created or later) and are destroyed when the containing classifier is destroyed.

Contained instances are represented graphically as classifiers nested within a compartment of the containing classifier. Figure 6 shows an example in which the structured classifier and the contained parts are all classes.[5] In this example, the containing classifier, Car, is composed of multiple Wheel parts. Each label within the parts has three pieces of information: a role name (e.g., left), a classifier name (Wheel), and a multiplicity (2). The model indicates that a car is composed of four wheels—two left and two right—and that each left wheel is associated with exactly one right wheel by an axle association.

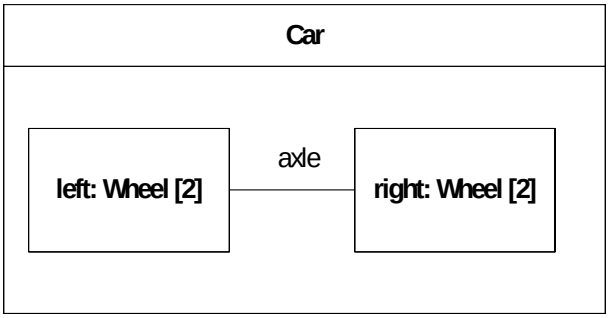

Figure 6: A Structured Class

A benefit of using a structured classifier, as opposed to a composition relation, is that constraints on the parts of a structured classifier only apply to instances of the parts that are within the scope of the structured classifier. For example, in Figure 6, the Wheel parts are constrained such that each Wheel instance must be associated with exactly one other Wheel instance. However, as this constraint is part of an occurrence within the structured classifier, it does not apply to Wheel instances that are not parts of Car instances (e.g., Wheel instances that are parts of Motorcycles).

5. The examples in Figure 6 and Figure 7 are adapted from examples in the UML standard [OMG 03, Figures 119 and 121].

Structured classifiers are instantiated in the same way as classifiers (e.g., classes). Figure 7 shows an instance of the Car structured class. The two instances of the left Wheel part are given names (l1 and l2), but the instances of the right Wheel part are anonymous.

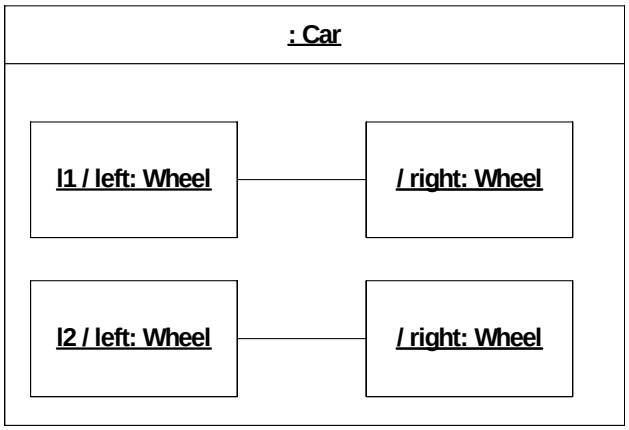

Figure 7: An Instance of a Structured Class

Additionally, classifiers, like classes and components, combine the descriptive capabilities of structured classifiers with ports and interfaces, permitting rich descriptions of structure, visibility, and precise interfaces. These concepts are illustrated in the following section in the context of components, but they can just as easily be used together with classes.

3.4 Components

In UML 2.0, the component concept has been generalized to be more meaningful throughout the development life cycle. In UML 1.4, *component* had a strong implementation bias as "a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces" [OMG 01, p. 2-31]. In UML 2.0, a component is "a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment" [OMG 03, p. 136]. This generalization permits the component concept to be used to describe component designs or implementations, without losing the ability to describe deployment information.

Additionally, component is now a subtype of class in the UML 2.0 metamodel. As such, components are as expressive as classes and have access to features such as subtyping through generalization relations, behavioral descriptions, internal structure, ports and interfaces, and instantiation. Components extend classes with additional features such as

- the ability to own more types of elements than classes can; for example, packages, constraints, use cases, and artifacts

- deployment specifications that define the execution parameters of a component deployed to a node

While the graphical representation of components from UML 1.4 (as shown on the right in Figure 8) is still supported for backward compatibility, two other representations are available in UML 2.0. The two examples on the left in Figure 8 show the new options for the external (black-box) view of a component. The leftmost representation (with the symbol in the upper right corner) is used in this report.

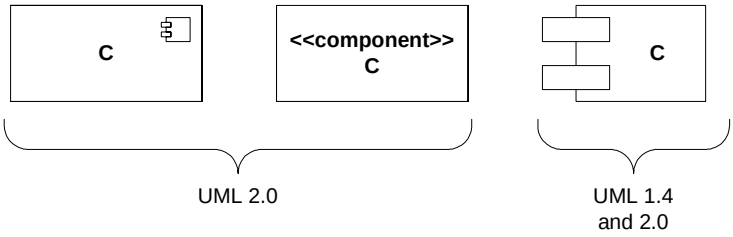

Figure 8: Three Ways to Represent Components

Figure 9 shows a sample component with ports. This figure shows an external view of the component in which its internal structure is not shown.

Figure 9: External View of a Component with Ports

Figure 10 elaborates on the example from Figure 9 by showing the internal (white-box) view of the component. The MergeAndSort component is composed of (using the structured classifier notation) two subcomponents: Merge and Sort. The external ports of MergeAndSort are connected to ports of the contained parts (Merge and Sort) by delegation connectors (see the next section for more information on connectors). The direction of the delegation connectors indicates that the in1 and in2 ports are associated with provided interfaces and the out ports are associated with required interfaces. The internal parts are connected by an unlabeled assembly connector.

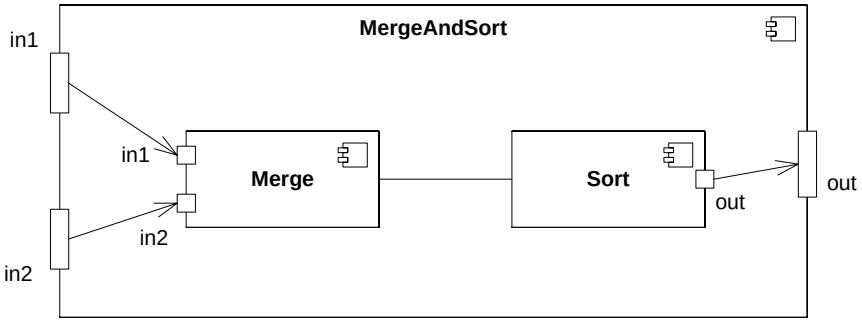

Figure 10: Internal View of a Component with Ports

3.5 Connectors

A *connector*, which is a new concept in UML 2.0, represents a communication link between two or more instances. Connectors can be realized by a variety of mechanisms, such as pointers or network connections—though the particular mechanism is not part of a connector definition. Formally, a connector is just a link between two or more connectable elements (e.g., ports or interfaces); it cannot be associated with a behavioral description or attributes that characterize the connection.

UML 2.0 specifies two kinds of connectors: *assembly* and *delegation*. An assembly connector is a binding between a provided interface and a required interface (or ports) that indicates that one component provides the services required by another. A delegation connector binds a component's external behavior (as specified at a port) to an internal realization of that behavior by one of its parts. For example, a delegation connector between the port of a component to the port of an internal part indicates that the behavior described at the component's port is realized by the internal part's behavior.

UML 2.0 defines some sensible but incomplete compatibility constraints over connectors. For example, delegation connectors must be used between the same kinds of ports or interfaces (provided or required), and assembly connectors can be defined only between a provided interface and a required interface that are compatible. The definition of *compatible*, however, is a semantic variation point, though signature compatibility and complete coverage (i.e., the features in the provided interface must be a superset of those in the required interface) are minimal criteria.

Connectors can be represented in different ways, depending on the type of the connector and the type of elements being joined. Figure 11 shows an assembly connector (the unlabeled line) between the ports of two components. Figure 12 shows an assembly connector between a provided interface of one component and a required interface of a second component using the ball-and-socket notation; in this form, the connector adds no new symbols and is represented by positioning the ball inside the connected socket. Connectors can also be defined directly between two components, without connecting ports or interfaces, though it is clearest to use explicit ports or interfaces.

Figure 11: An Assembly Connector Between Ports

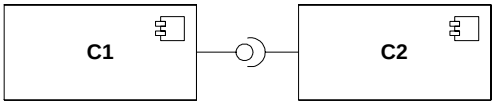

Figure 12: An Assembly Connector Between Interfaces

Figure 13 shows two delegation connectors between a component's external ports and the ports of its internal parts (components in this case). Delegation connectors are shown as arrows whose direction is dependent on the kinds of ports or interfaces being connected. The left delegation connector is drawn from an external port to an internal port because the external port has a provided interface; the delegation connector indicates that the realization of that interface is provided by the internal part. The right delegation connector is drawn from an internal port to an external port because the latter has a required interface; the delegation connector indicates that the internal part's requirement is not satisfied within the outer component and is passed on to the outer component's environment.

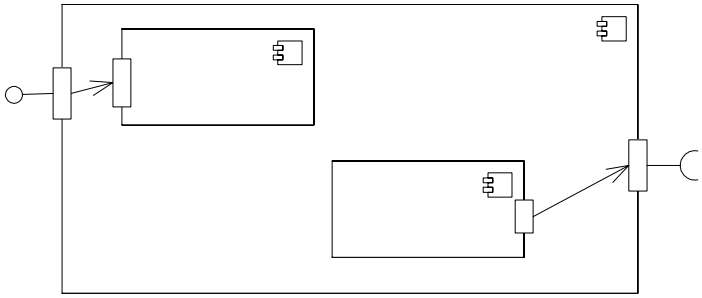

Figure 13: Delegation Connectors Between External and Internal Ports

4 Documenting C&C Views with UML 2.0

The new and modified features in UML 2.0, combined with those inherited from earlier versions of UML, provide a rich vocabulary for documenting software architectures and help solve several of the problems involved with using those earlier versions. In particular, though you can encode architectural concepts in many ways in UML 1.4, none were completely adequate. UML 2.0, on the other hand, provides natural modeling constructs for most architectural concepts.

However, as described below, even with UML 2.0, there are several ways for some architectural concepts to be represented naturally, so users must still choose a documentation strategy. Moreover, some aspects of architectural documentation continue to be problematic. In this section, we describe the main documentation strategies that can be used to document C&C views in UML 2.0 and note the modeling inadequacies that still exist.

When considering documentation strategies it is important to be clear about criteria for choosing one way of using UML 2.0 over other possibilities. These criteria allow us to determine when a particular documentation strategy is likely to be appropriate. They also allow us to point out architectural concepts that continue to be difficult to document using UML. The criteria (the first three of which are derived from the work of Garlan and associates [Garlan 02]) are as follows.

1. semantic match: The UML constructs should map intuitively to the architectural features being documented.
2. visual clarity: The UML description should bring conceptual clarity to a system design, avoid visual clutter, and highlight key design details.
3. completeness: All relevant architectural features for the design should be represented in the UML model. The ability to capture this information is also referred to as expressiveness.
4. tool support: Not all uses of UML are supported equally by all tools (particularly when specializing UML), which can limit documentation options.

In practice, choosing a strategy involves making tradeoffs among the strengths and weaknesses of each possible strategy, and no one strategy is the best choice for all documentation purposes.

When choosing strategies, it is also important to understand how the documentation will be used. Sometimes, it is even beneficial to use different strategies at different points in the software development life cycle as the design is elaborated and refined.

Design elaboration is the process of gradually adding information to a given view, often starting with a sketch and adding information over time as additional decisions are made until the view is complete. In such an approach, it may be appropriate to begin with documentation strategies that have a high visual clarity, but are not complete. Over time, the set of documentation strategies could evolve to improve the completeness of the documentation, perhaps at the expense of visual clarity.

Design refinement is the process of gradually disclosing information across a series of descriptions. It might be used when the architect is designing the architecture or as a way to structure the documentation to represent the information that is understandable to particular stakeholders. Decomposition refinement is the elaboration of a single element that reveals its internal structure as documented in the same architectural style. In implementation refinement, many or all of the elements and relations are replaced by new, typically more implementation-specific elements and relations, as documented in a different architectural style. The documentation of different refinements, particularly for different styles, might use different strategies matched to the level of completeness appropriate at each level of decomposition.

4.1 Documenting Components

The changes in UML 2.0 provide a more natural representation for C&C components than UML 1.4 allowed. In particular, the introduction of ports and structured classifiers in UML 2.0 provides a clear means for representing component ports (interfaces) and component decomposition that is not dependent on specializations of UML.

4.1.1 Strategy 1: Using UML Classes

A natural candidate for representing C&C component types in UML is the class concept. The changes in UML 2.0 that support the representation of classes that may have ports and internal structure improve UML's ability to completely capture the details of C&C components in a natural way.

Classes describe the conceptual vocabulary of a system just as C&C component types form the conceptual vocabulary of architectural documentation in a particular style or view. Additionally, the relationship between classes and objects is similar to that between component types and their instances. Figure 14 illustrates the general idea, showing component types on the left

and a component instance on the right, as they can be documented using UML classes and objects.

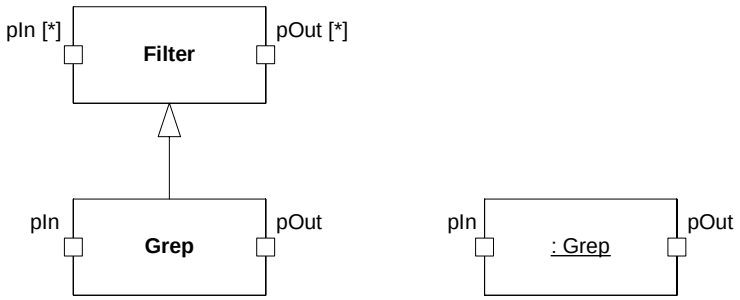

Figure 14: C&C Types as UML Classes and C&C Instances as UML Objects

The type/instance relationship in architectural descriptions is a close match to the class/object relationship in a UML model. UML classes, like component types in architectural descriptions, are first-class entities and rich structures for capturing software abstractions. The full set of UML descriptive mechanisms is available to describe the structure, properties, and behavior of a class, making this strategy a good choice for depicting detail and using UML-based analysis tools.

Structural properties of architectural components can be represented as class attributes or associations, behavior can be described using UML behavioral descriptions (e.g., statecharts), and generalization can be used to relate a set of component types. The semantics of an instance or a type can also be elaborated by attaching one of the standard stereotypes; for example, the <<process>> stereotype can be attached to a UML class representing a C&C component to indicate that the component runs as a separate process. The details of representing substructure are described in Section 4.4.

As noted by Clements and associates [Clements 02], the typical relationship between classes and instances is not identical to that between architectural components and their instances. A component instance might refine the number of ports specified by its type or associate an implementation in the form of an additional structure that is not part of its type's definition. In UML 2.0, as well as UML 1.4, an object can include only those parts defined in its class.

A more natural way to capture the component type/instance relationship in UML is to define an intermediate component type that specializes the general component type. The instance's structure (as represented by a UML object) matches the intermediate type's structure (as represented by a UML subclass of the class representing the general component type) with the usual UML type/instance meaning. This approach is shown in Figure 14 where a Grep subtype of Filter is defined. The component instance is shown as an anonymous instance of the Grep subtype, and its structure and behavior matches that of the Grep subtype.

4.1.2 Strategy 2: Using UML Components

The same UML 2.0 changes that improve the suitability of a UML class for representing a C&C component also apply to UML components because UML components were changed to be a subtype of classes in the metamodel. UML components have an expressive power similar to that of classes and can be used to represent C&C components as in the first strategy. To represent the same information, only the graphical depiction needs to be changed (as shown in Figure 15) to replace the class symbols with component symbols.

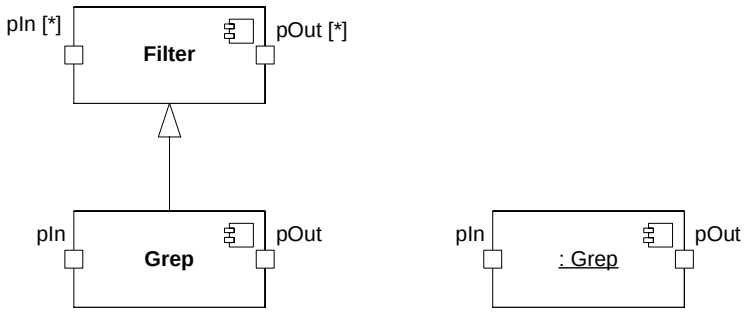

Figure 15: C&C Types as UML Component Types and C&C Instances as UML Component Instances

The main difference between UML components and classes is that, in addition to the elements a class can own, a component can also own packageable elements. This distinction allows a designer to attach such elements as deployment descriptors, property files, or any kind of document to a component. It also supports the definition of technology-dependent details and UML technology-specific profiles such as those for EJB or .Net.

These extra capabilities are not always necessary in an architectural design, and most of the time, classes are sufficient to represent all the relevant information in an architectural component.

4.1.3 Choosing

With respect to most of the information typically documented in a C&C view, either strategy is adequate since UML classes and components have nearly the same expressiveness (i.e., they are complete with respect to representing all relevant C&C component features in the UML model). In such cases, the choice of which strategy to use may rely more on the semantic match offered by the strategies. Is a UML component or a class a better (or worse) match to a C&C component?

The UML component may seem like it is always a better choice—it has the same name, after all. However, UML components have a history that is not entirely compatible with architec-

tural documentation. As mentioned earlier, in UML 1.4, a UML component was not considered to be a useful means of documenting C&C components because UML components were closely associated with implementation artifacts, particularly those associated with component technologies. For documentation readers familiar with the semantics of UML 1.4 components, an unintentional implementation bias may be difficult to overcome when using UML components to represent C&C components.

On the other hand, in some settings, such a bias could be beneficial. For an organization that has a development history of building component-based systems (implementations for some component technology), the UML approach to components is a good fit. The UML 2.0 lifecycle philosophy of component-based development uses the same element—the component—as both a design abstraction and a realizing implementation. A component is modeled throughout the development life cycle and successively refined into deployment and runtime.

While the UML specification enables the use of components throughout the development life cycle, there are potential problems with using the same construct for representing both a design abstraction and implementation, particularly when there is no one-to-one mapping between the two. If implementation artifacts do not map directly to architectural abstractions, or if you are not yet sure whether they will, it might be better to document the abstract C&C components using classes and reserve UML components for documenting the resulting implementation units. Using two different modeling elements will avoid confusion when describing how the artifacts relate to each other.

Visually, the two strategies are nearly identical, differing only in the presence or absence of the component symbol in the classifier boxes. However, choosing strategies for components that are visually compatible with those for connectors may be a concern. For example, one strategy for documenting C&C connectors is to use classes. When this strategy is combined with the strategy for documenting C&C components using classes, the visual distinction between components and connectors suffers. In such a case, using UML components to represent C&C components provides visual clarity by not using the same symbol (a UML class) for both C&C components and connectors.

4.2 Documenting Ports

Documenting C&C ports is one area in which the changes in UML 2.0 really shine. The newly introduced port concept in UML 2.0 is so well suited for documenting C&C ports that we no longer consider any other strategies.

UML ports are explicit interaction points for classifiers (specifically, for classes and components—covering both of our strategies for documenting C&C components). UML ports with public visibility are a better semantic match for C&C ports if we add the constraint that all

interactions of the component with its environment are achieved through ports. This constraint allows the component to be used in any context that satisfies the constraints specified by its ports.

UML ports provide all the expressiveness needed to document C&C ports. Multiple ports can be defined for a component type, enabling different interactions to be distinguished based on the port through which they occur. Multiple instances of the same port type are allowed and can be distinguished. Ports can be associated with multiple interfaces (provided and required) that completely characterize any interaction that may occur between the component and its environment at a port. Ports can be typed by an interface or a class. The latter case allows attributes, substructure, and a more elaborate specification of the communication over a port such as a behavioral description.

A port is shown as a small rectangle symbol overlapping the boundary of the rectangle symbol denoting the component (see Figure 16). The name of the port is placed near the square symbol (e.g., `pIn` and `pOut`). A provided interface is shown using a "lollipop" symbol attached to the port, and a required interface is shown using a socket symbol attached to the port.

Figure 16: C&C Ports as UML Ports

The amount of detail included in component interface documentation can increase over time as the design is elaborated; for example, starting with no ports, then adding ports with no interfaces, then adding provided and required interfaces to the ports, and finally supplementing ports with information about behavioral restrictions.

Leaving ports out leads to the simplest diagrams. This choice might be reasonable if the components have a single port, if the ports can be inferred from the system topology, or if the component is refined elsewhere. Identifying the ports of a component allows different interactions to be distinguished based on the port through which they occur.

Adding interfaces to the ports characterizes aspects of how a component can interact with its environment and clearly distinguishes between services that the component provides to, and requires from, its environment.

4.3 Documenting Connectors

While the changes in UML 2.0 improve its suitability for documenting components and ports, similar improvements supporting C&C connectors are missing. The UML 2.0 connector con-

cept, which is new, is too lacking in expressiveness to be a good solution for documenting C&C connectors. In particular, it lacks any ability to associate semantic information with a connector (e.g., a behavioral description) or to clearly document C&C connector roles. As a result, in this section, we describe three strategies for documenting connectors[6] using UML 2.0. The strategies are roughly the same as those presented for documenting connectors using UML 1.4 but with some incremental improvements from other UML changes, such as the introduction of UML ports. The three strategies, which are described below, are

1. using UML associations
2. using UML association classes
3. using UML classes

4.3.1 Strategy 1: Using UML Associations

The first strategy, which does not improve on the equivalent strategy for UML 1.4, is to represent C&C connectors using UML associations or assembly connectors (for our purposes, these concepts offer the same expressiveness). Figure 17 shows an example in which two C&C components (represented by UML objects) are connected with a UML link (an instance of an association) representing a pipe connector.

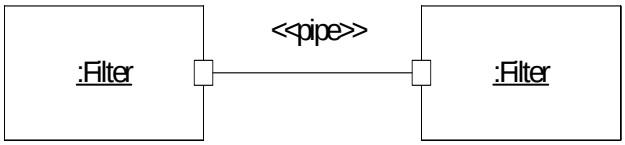

Figure 17: A C&C Connector as a UML Association (Link)

The strategy is visually appealing, allowing a quick visual distinction between C&C components and connectors. Different types of C&C connectors can be distinguished by labeling each association with a stereotype that names the type of the C&C connector. (Figure 17 identifies the connector as a pipe.)

However, there is a limit to the expressiveness of this strategy. C&C connectors are used to coordinate component behavior and have semantics of their own. For example, how a pipe connector behaves when its buffer is full has an impact on system behavior and constrains, or at least influences, component behavior. Documenting connector semantics is important because it provides architects and analysts with the information needed to understand how the choice of a particular connector type will impact system behavior.

---

6. For the sake of simplicity, from this point forward, the term *connector* refers to a *C&C connector*. Whenever a UML connector is meant, UML is mentioned explicitly.

In contrast, UML associations represent a potential for interaction between two classifiers but do not have any behavior of their own. Consequently, a number of C&C connector characteristics cannot be expressed using this representation in UML. Specifically

- The roles (interfaces) of C&C connectors cannot be defined because UML associations cannot have UML interfaces or ports, the most appropriate means to represent C&C roles.
- C&C connector semantics cannot be defined because UML associations cannot own attributes (e.g., to record buffer size) or have behavioral descriptions (e.g., to define queuing or blocking policy).

Even with these limitations, this strategy is still useful in the right circumstances. If the purpose of your documentation is simply to identify where different types of connectors are used in a system, this strategy works well.

C&C views documented using box-and-line notations often represent connectors as stylized lines, so this UML representation provides equivalent visual aesthetics.

4.3.2 Strategy 2: Using UML Association Classes

The second strategy, which improves on the equivalent strategy for UML 1.4, is to represent C&C connectors using UML association classes. Figure 18 recasts our example by using an association class (actually, an instance of an association class—a link object) to represent a C&C pipe connector.

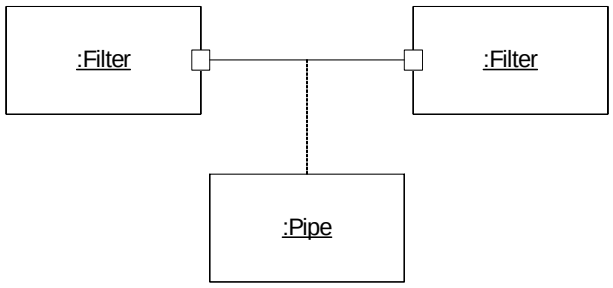

Figure 18: A C&C Connector as a UML Association Class (Link Object)

Using association classes addresses many of the limitations of using associations. The class portion of the association class allows rich semantic descriptions, including attributes and behavioral descriptions. The class could also have substructure if decomposition of the connector is useful (e.g., to show more details of how the connector is to be implemented). This strategy allows connector types to be defined independently of usage; the association class could even be part of a type hierarchy, as documented in a class diagram.

This strategy also allows C&C connector roles to be represented using UML ports on the association class, as shown in Figure 19—a definite improvement over the options available in UML 1.4, all of which made role documentation difficult. Using UML ports, C&C connector roles can be documented as completely as C&C component ports (see Section 4.2 for more information).

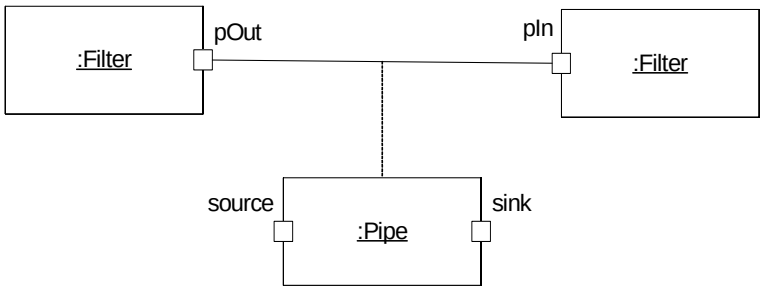

Figure 19: A C&C Connector as a UML Association Class with Ports

However, attaching C&C component ports to C&C connector roles is problematic. In Figure 19, it is not clear whether the `source` C&C connector role is attached to the `pOut` or `pIn` component port. This ambiguity could be addressed in one of two ways, as shown in Figures 20 and 21. In Figure 20, attachment is shown using UML role names on the link that correspond to the C&C role names (which are represented by UML port names on the link's object). This convention is more suggestive than formal, however, and tool support to ensure that labeling is consistent or even that there is one UML port on the object for every end of the link is unlikely without custom tools or extensions.

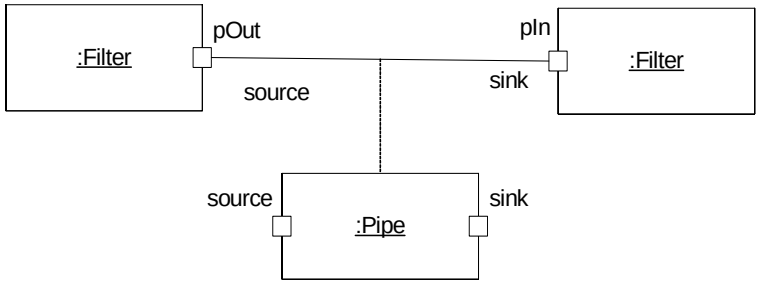

Figure 20: Link Role Names Are Used to Represent Attachments

In Figure 21, attachment is shown explicitly using UML assembly connectors between the ports of the object representing the connector and the ports of the objects representing the components. While unambiguous, the association portion of this diagram is redundant—it shows only information that could be derived from the UML assembly connectors used to con-

nect the C&C ports and roles. Worse, this approach introduces substantial visual clutter; a C&C connector, often visualized as a single line, is represented by a box and four lines.

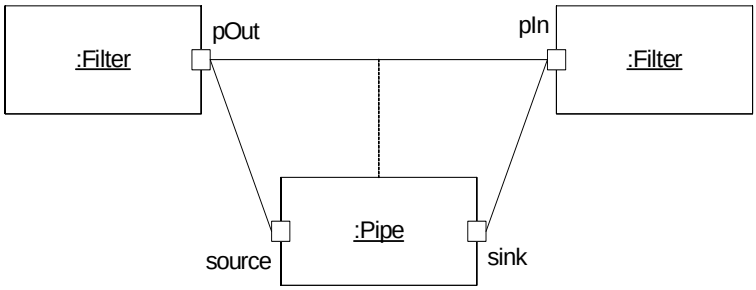

Figure 21: Assembly Connectors Are Used to Represent Attachments

Using an association class, rather than an association, to represent a C&C connector allows greater semantic expressiveness and the representation of connector roles, but at the cost of visual aesthetics. While C&C views documented using box-and-line notations often represent connectors as stylized lines, this UML representation requires additional visual elements (at least a box and sometimes additional lines) that can lead quickly to visual clutter as the number of connectors in a system grows.

4.3.3 Strategy 3: Using UML Classes

The third strategy is to represent C&C connectors using UML classes. Figure 22 recasts our examples using an object instance (of a class representing the C&C connector type) to represent a C&C pipe connector.

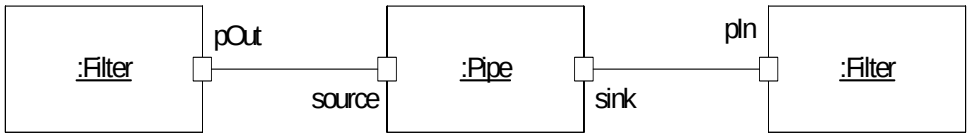

Figure 22: A C&C Connector as a UML Class (Object)

This strategy resolves the component and connector attachment problems of the second strategy while retaining its expressiveness. The class representing the C&C connector allows the same semantic descriptions, independent definition of types, and use of UML ports to represent C&C roles. However, C&C attachments are always represented explicitly using UML assembly connectors, removing the potential ambiguity of the second strategy. Using UML classes offers essentially the same solution as the association class variant shown in Figure 21, but without the redundant association portion of the association class and its associated visual clutter.

Unfortunately, this solution presents the poorest visual distinction between C&C components and connectors (because both are represented as boxes) and dilutes one of the benefits of a C&C view—the ability to quickly identify the principle computational elements and understand the patterns of interaction among them. This problem can, to some extent, be mitigated by using different UML concepts to represent C&C components and connectors. If C&C components are represented by UML classes, UML components can be used to represent C&C connectors and vice versa. Figure 23 shows a variation in which C&C components are represented by UML components and C&C connectors are represented by UML classes.

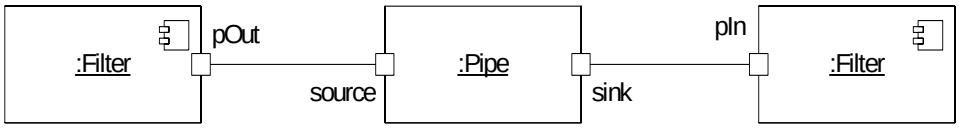

Figure 23: Using a UML Class for a C&C Connector and a UML Component for a C&C Component

However, the variation above is only a little better in terms of visual distinctiveness. The ideal approach would be to combine the expressiveness of this strategy with a different visualization (such as that from the first strategy) by using UML's stereotype mechanisms, which permit the visualization of stereotyped elements to be customized. For example, Figure 24 shows an example in which a C&C connector is represented using a stereotyped UML class. The stereotype customizes the visualization of the stereotyped class to be a thick line segment with ports.[7]

Figure 24: A C&C Connector as a UML Stereotyped Class with Custom Visualization

Unfortunately, this use of stereotypes requires graphical support not offered by most UML tools, so its practical application is limited.

4.3.4 Choosing

Which strategy is best suited to documenting C&C connectors for a particular system depends on a number of factors, and different strategies may be more effective at different points in the development life cycle. Some questions to guide your decision are

7. In this report, we have restricted ourselves to minimal specializations of UML, with this change in visualization being the most drastic one. More precise architectural specifications can be produced using more sophisticated specializations of UML [Selic 98, Medvidovic 02] but doing so creates a need to educate a potentially large group of stakeholders. Managing the tradeoffs between using a more precise specialization and using the widely understood standard "as is" is a topic beyond the scope of this report.

- Are you identifying which types of connectors are used, describing what effect the connectors have on component interaction, and/or providing enough design information to guide the connector implementation?
- Where are you in the system's development life cycle (i.e., which decisions have been made and which have been deferred)?
- How are you representing components? Are their visualizations distinct from those of the connectors?
- What tool support is available?

The first strategy—representing C&C connectors using UML associations—is a good choice when the goal of the documentation is to identify where different types of connectors are used in a system, particularly if the connector types are well known and understood (e.g., procedure call connectors). The first strategy is a poor choice if connector semantics need to be documented or connector roles need to be shown.

The second strategy, representing C&C connectors using UML association classes, is a good choice when connector semantics need to be described, but specific component port and connector role attachments are not important. Such a situation might occur because a connector offers only one type of role (making distinctions among specific roles unnecessary) or because precise interface decisions have not yet been made. While the third strategy (using UML classes) could also be used in such cases, using UML association classes provides better visual distinctiveness between components and connectors than the third strategy and may be preferable if specific attachments are not important.

The third strategy, representing C&C connectors using UML classes, is a good choice when connector semantics and/or specific component port and connector role attachments are important. This strategy is a good choice in situations where you would normally use UML association classes (the second strategy) and when tool support is available to visualize stereotyped classes differently (e.g., as a line segment with ports, as shown in Figure 24).

From a life-cycle perspective, the first strategy can be a good choice for "first drafts" in which specific connector semantics have not been defined, but crude choices should be identified by name. Using an association leads to the simplest of diagrams. Another representation can be chosen when the need arises and more semantic information is available. Refinements of the view could gradually add more semantic information, moving to the second strategy of representing C&C connectors using UML association classes when state is needed, and then moving to the third strategy of representing C&C connectors using UML classes when unambiguous attachments are needed.

4.4 Documenting Systems

C&C views of systems depend on two types of information: 1) definitions of component and connector types and 2) topologies of instances of component and connector types forming the system. Each type of information is described below.

4.4.1 Component and Connector Types

Types and type hierarchies (if used) are documented using class or component diagrams, as shown in Figure 25. Generalizations are used to show subtypes, and classes can be used to define the decomposition of types. For a complete C&C view, interfaces, attributes, and behavioral models should be fully defined for each type. Interfaces should be documented in terms of UML ports and interfaces; the interface representation shown in Figure 1 on page 12 is a good choice, as it allows the most complete interface descriptions.

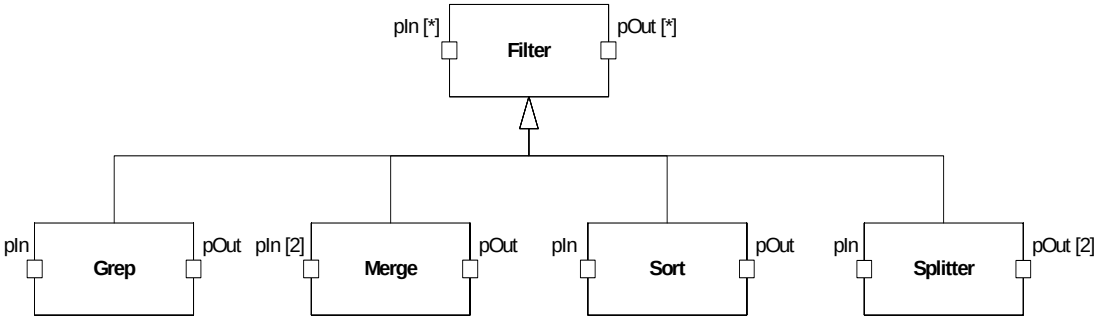

Figure 25: Documentation of Component Type Hierarchy

4.4.2 Topologies of Component and Connector Instances

Topologies of component and connector instances forming a system are documented using instance diagrams, as shown in Figure 26. This approach retains the benefits of representing systems as contained objects that were noted by Clements and associates [Clements 02] for UML 1.4. Objects and component instances provide a strong encapsulation boundary and carry with them the notion that each instance of the classifier will have the associated sub-structure. This approach also overcomes some of the problems noted for UML 1.4. Using that version, Clements and associates represented the substructure of MergeAndSort as a package and the relationship between MergeAndSort and its substructure using a dependency relation [Clements 02]. UML 2.0 allows internal structure to be nested within the class, as shown earlier in Figure 10. Associations used to represent connectors between contained components are now scoped by the containing structured classifier, allowing us to constrain only the contained instances of a component type without constraining all such instances.

In some cases, details found in the component or connector type definitions can be suppressed in the instance diagrams to improve visual clarity. For example, if a component type's ports each have only one interface, it is not necessary to explicitly show the interfaces for instances of the component type; the interface information is derivable from the type specification. In such cases, the concise interface representation shown in Figure 3 on page 13 is a more appropriate choice.

Likewise, interfaces of a port need not be shown when connecting two ports that are compatible (i.e., all provided and required interfaces of one port have a complement in the connected port). If multiple interfaces of a port are connected to different ports or different classifiers, it may be helpful to show the interface details.

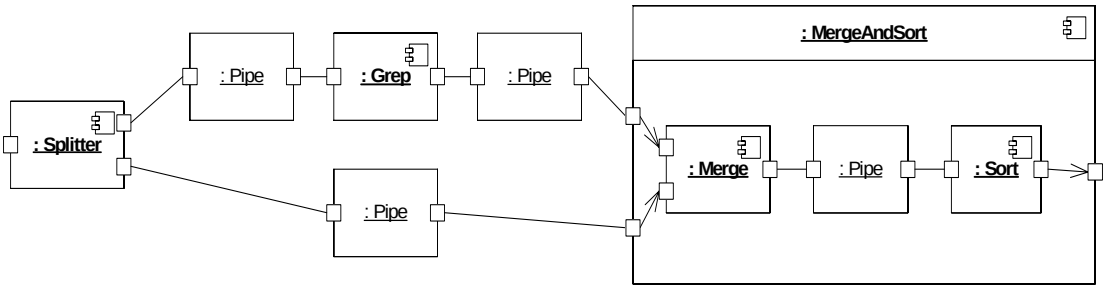

Figure 26: Documentation of a System

Regardless, consistent use of documentation conventions is important. If ports are documented, all interaction between components (via connectors) should only occur at ports. Attaching some connectors to component ports and others directly to components, as allowed by UML, introduces confusion.

4.5 Documenting Properties

Architectural properties are used to capture *semantic* information about a system and its elements that goes beyond structure [Bass 03]. Some properties capture quality attributes such as a connector's throughput, a component's response time, or a component's mean time to failure. Other properties capture other information needed for architectural analysis or as hints for developers, such as thread priority. Many properties, however, are often not represented in the eventual implementation.

UML 2.0 also has a concept called property, but it represents a structural feature [OMG 03]. For example, when a class defines a property, that property is defined as an attribute, and the attribute is expected to appear as a field in the corresponding implementation. When an association owns a property, that property is not a good semantic match for an architectural property because it represents a non-navigable end of the association. These UML uses of the term

property, along with other UML specializations of the concept, are poor choices for representing the semantic information contained in a C&C property.

Thus, UML 2.0 properties cannot be used directly to represent architectural properties. Instead, we introduce three strategies that can be used to document architectural properties in UML 2.0: 1) using tagged values, 2) using attributes to represent type-level properties, and 3) using stereotypes to represent type-level properties.

4.5.1 Strategy 1: Using UML Tagged Values

In UML, a *tagged value* is an explicit definition of a property as a name-value pair [OMG 03].

Tagged values can be used as an extension mechanism to document information that is semantically relevant to a classifier, but is not part of its structure. Tagged values can therefore be used to document architectural properties.

Figure 27 illustrates how a tagged value can be attached to an object. The InvoiceServer instance has a tagged value named “multithreaded” with a value of yes.

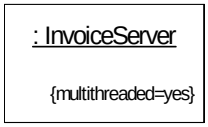

Figure 27: Architectural Properties as UML Tagged Values

However, there are limitations to using tagged values to represent the architectural properties of a component:

- Since the tagged value is a name-value pair, there is no explicit documentation of the value’s type.
- A tagged value (like the one in Figure 27) is defined only for the instance; there is no notion of a property shared by all server instances.

Some architecture description languages (e.g., Acme [Garlan 00]) overcome these limitations by allowing types to be defined that specify the properties each instance must have. In such languages, it is possible to define an InvoiceServer type where each instance requires the property *multithreaded*. This ability is important when properties are analyzed by tools, and those tools expect certain properties to be defined.

4.5.2 Strategy 2: Using UML Attributes

The easiest way to represent properties in UML is as attributes of a class or component (see Figure 28). This approach is very easy to understand and supported well by typical UML tools. However, this representation has limitations with respect to semantic match; as mentioned above, architectural properties are not structural elements but rather semantic ones. Using this approach could cause misinterpretation that could, for example, result in code that is generated for the semantic properties.

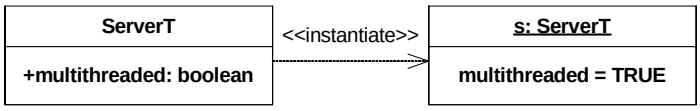

Figure 28: Architectural Properties as UML Attributes

A variation that overcomes the possible misinterpretation caused by using attributes directly is to use a stereotype denoting that an attribute is semantic, not structural. For example, Figure 29 shows the same component shown in Figure 28 (the ServerT component) with its *multi-threaded* attribute stereotyped as <<semantic>>.

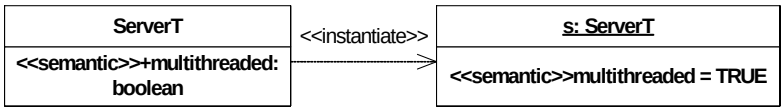

Figure 29: Architectural Properties as UML Stereotyped Attributes

This mapping is intuitive and has some level of tool support. However, tools that do not recognize the stereotype may produce inappropriate results, such as the generation of code for those attributes as if they were structural.

4.5.3 Strategy 3: Using UML Stereotypes

A more accurate, but more complex, way to represent semantic information is through stereotypes. Stereotypes provide a way to extend concepts in the UML metamodel and allow new semantics to be incorporated into UML models, enabling the use of domain-specific terminology and notation in addition to those used for the extended metaclass.

One way to extend a concept is to define a stereotype that includes a *tag definition*. Like an attribute of a class, a tag definition is a named definition of a type of data; the PipelineFilter stereotype in Figure 30 includes a throughput tag definition. When the stereotype is applied, as shown in Figure 31, the value of a tagged definition is called a *tagged value*. Unlike class attributes, tagged values are not structural and are not carried through to implementations.

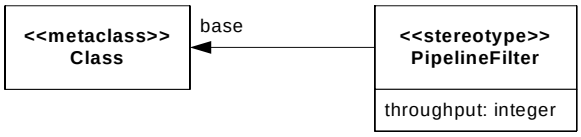

Figure 30: Architectural Property Captured in a UML Stereotype

A stereotype is defined in UML in much the same way as a class. It is drawn as a classifier with a separate compartment for its tagged definitions. An extension relation (line with a solid arrowhead) is drawn from the stereotype to the class in the metamodel that is being extended.

Figure 31 shows a use of the stereotype defined in Figure 30 for an EncryptionFilter component (documented using the UML class strategy). The note attached to the component contains the throughput tagged value.

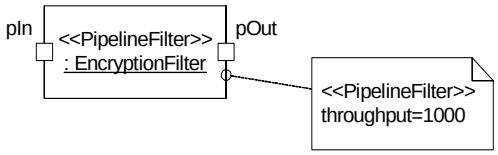

Figure 31: Stereotype Applied to an Instance

Although the throughput tagged value is available in the EncryptionFilter class definition, there is no requirement that a value must be assigned to it. Doing so would give semantic information that may not make sense at this level of abstraction because the stereotype may extend multiple metaclasses. Instead, we can add a new extension that allows the stereotype to extend not only the EncryptionFilter class but also its instances (where assigning value to the throughput tagged definition does make sense). We can refine the stereotype definition using the Object Constraint Language (OCL) to say that the throughput tagged value can have a name only if the stereotype is applied to an instance of a class. Figure 31 shows how the stereotype can be applied to an instance of the EncryptionFilter.

4.5.4 Choosing

If analysis tools are not used and properties are not required for all instances of a class, the first strategy is adequate. These values might be used to guide the design or convey information to developers on selected components.

The second strategy is a good choice if the properties are mandatory to support analysis, but the implementation consequences are not terribly detrimental (e.g., if there are no memory constraints and there is good documentation regarding the unnecessary program variables).

The second strategy ensures that all components consider the values for the specified architectural properties and provides explicit documentation of the property type.

The third strategy also provides explicit documentation of the property type, but lacks the semantic mismatch and potential implementation consequences of the second strategy. Although the third strategy does not require that values be supplied, it provides a stronger hint than the first strategy does by providing a placeholder and associated semantics about the type.

5 Conclusion

UML 2.0 provides many ways to document architectures. We have outlined the principle strategies and their variants, and provided a number of guidelines for determining which strategies to select.

Comparing UML 2.0 to its predecessors, we can see that the situation has improved considerably. Several of the problems with documenting C&C concepts in UML 1.4 have been fixed, and, for other problems, the number of reasonable strategies for solving them has been reduced, in particular

- The concept of structured classifiers now permits natural representation of architectural hierarchy.
- The concept of ports now provides a natural way to represent runtime points of interaction.

However, some problems still remain that lead to a variety of encoding mechanisms that continue to make documenting architectures complex, specifically

- UML connectors are not first class, and it is difficult to associate detailed semantic descriptions or representations with them. Consequently, they are a poor choice for representing C&C connectors, and less natural representations must be used.
- Properties can be represented, but there is no completely natural way to do it.

An important topic not addressed in this report is how to represent architectural styles. Although profiles can still be used (as in UML 1.4), a more direct representation does not appear to be available in UML 2.0. However, further investigation is warranted.

Appendix A C&C View Examples

This appendix shows how to document the same system using the C&C pipe-and-filter style using Acme and two different combinations of strategies for using UML 2.0.

A.1 Using Acme

Acme is an architecture description language designed for component and connector views [Garlan 00]. As such, it has a modeling element that is a semantic match for each C&C element listed in Section 2. Though Acme is not as widely used as UML, it can be helpful to compare how software architectures are documented using a special-purpose language (Acme) and a general language (UML).

Figure 32 shows a graphical representation of an Acme description of the system, as documented using the AcmeStudio tool.

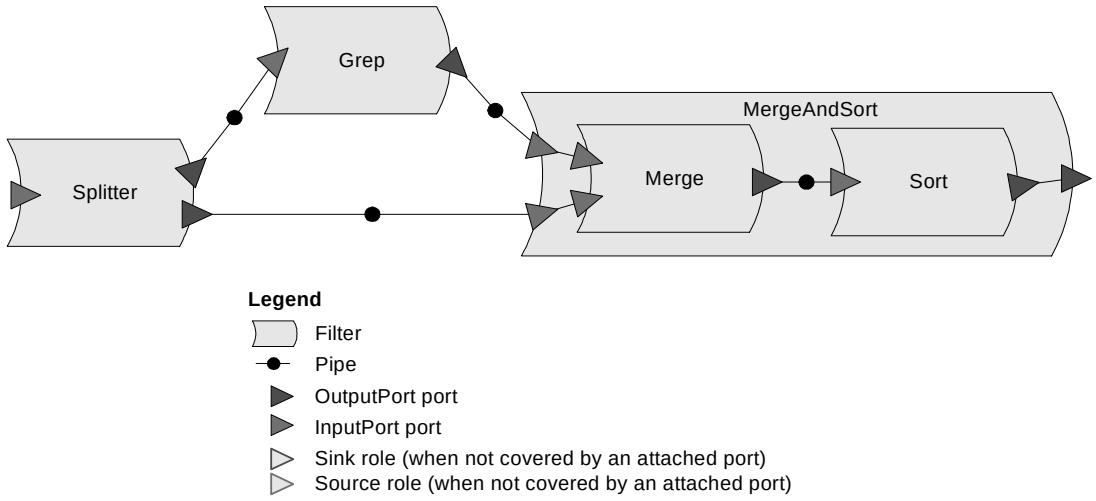

Figure 32: System Documented Using AcmeStudio

A.2 UML 2.0 Variation 1

In this variation, the UML 2.0 documentation strategies listed in Table 2 are used to document the system in Figures 33 - 35.

Table 2: Summary of UML 2.0 Strategies Used in Variation 1

C&C Element	UML Representation
Component (Type)	Object (Class)
Port	Port
Connector (Type)	Link Object (Association Class)
Role	Port
Attachment	not shown

Figure 33 shows the style key for this approach to using UML to document a view in the pipe-and-filter style.

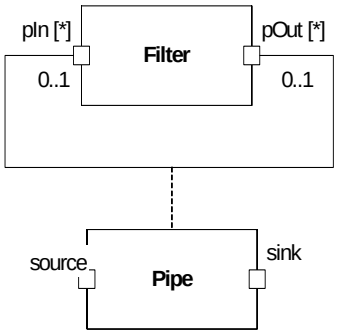

Figure 33: Style Key

Figure 34 shows the component types used to document the system. Each subtype of the general Filter component type binds the number of ports to a fixed number and could have a different behavioral description describing its computation.

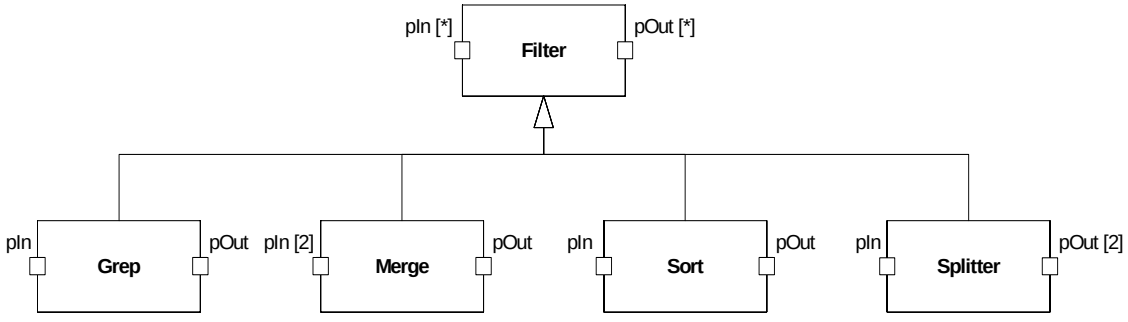

Figure 34: Component Types

Figure 35 shows the primary presentation for the pipe-and-filter view of this system.

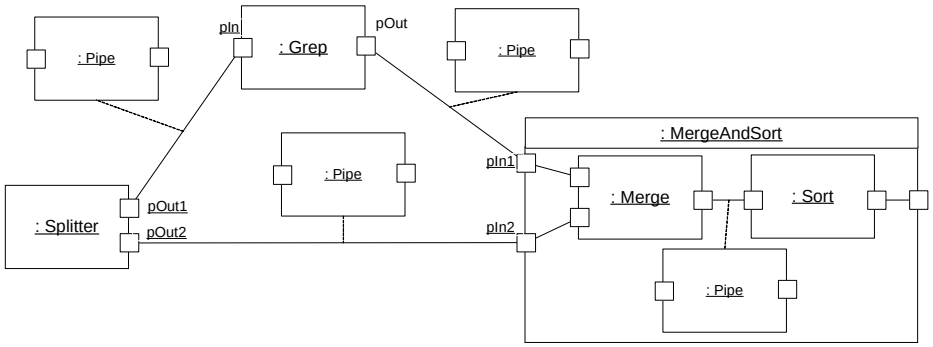

Figure 35: System

A.3 UML 2.0 Variation 2

In this variation, the UML 2.0 documentation strategies listed in Table 3 are used to document the system in Figures 36 - 38.

Table 3: Summary of UML 2.0 Strategies Used in Variation 2

C&C Element	UML Representation
Component (Type)	Component Instance (Component Type)
Port	Port
Connector (Type)	Object (Class)
Role	Port
Attachment	Assembly Connector

Figure 36 shows the style key for this approach to using UML to document a view in the pipe-and-filter style.

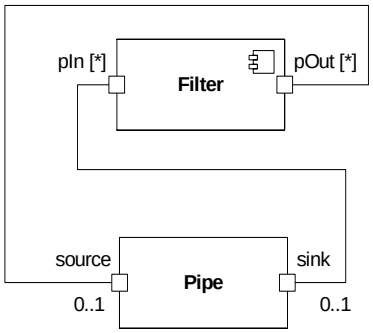

Figure 36: Style Key

Figure 37 shows the component types used to document the system. Each subtype of the general Filter component type binds the number of ports to a fixed number and could have a different behavioral description describing its computation.

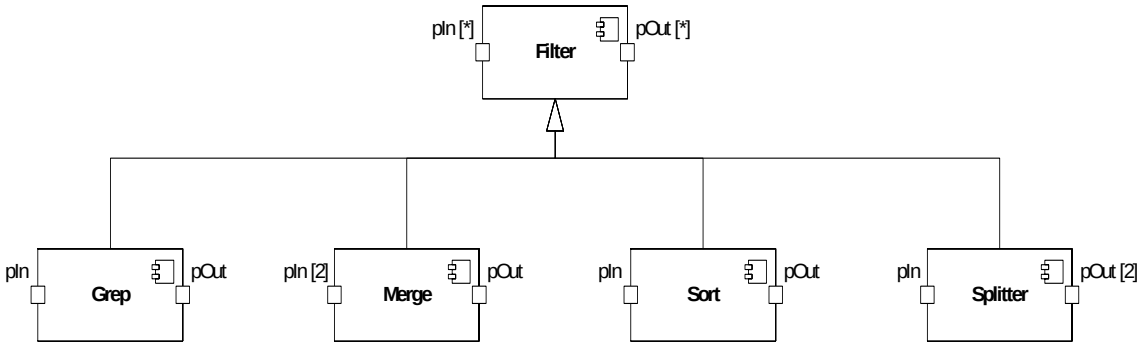

Figure 37: Component Types

Figure 38 shows the primary presentation for the pipe-and-filter view of this system.

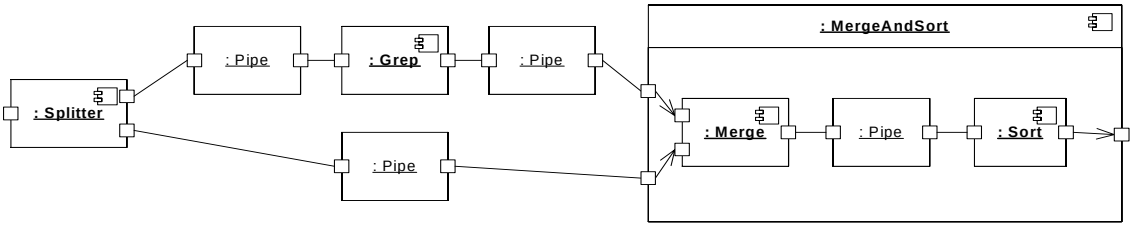

Figure 38: System

References

URLs are valid as of the publication date of this document.

Bass 03 Bass, L.; Clements, P.; & Kazman, R. *Software Architecture in Practice, Second Edition.* Boston, MA: Addison-Wesley, 2003.

Clements 02 Clements, P.; Bachmann, F.; Bass, L.; Garlan, D.; Ivers, J.; Little, R.; Nord, R.; & Stafford, J. *Documenting Software Architectures: Views and Beyond*. Boston, MA: Addison-Wesley, 2002.

Gamma 95 Gamma, E. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

Garlan 00 Garlan, D.; Monroe, R.; & Wile, D. Ch. 2, "Acme: A Basis for Architectural Integration," 47-68. *Foundations of Component-Based Systems*. Leavens, G. & Sitaraman, M., eds. New York, NY: Cambridge University Press, 2000.

Garlan 02 Garlan, D.; Cheng, S.; & Kompanek, A. J. "Reconciling the Needs of Architectural Description with Object Modeling Notations." *Science of Computer Programming, Special UML Edition 44,* 1 (July 2002): 23-49.

Hofmeister 00 Hofmeister, C.; Nord, R.; & Soni, D. *Applied Software Architecture*. Boston, MA: Addison-Wesley, 2000.

IEEE 00 Institute of Electrical and Electronics Engineers. *Recommended Practice for Architectural Description of Software-Intensive Systems* (IEEE Std 1471-2000). New York, NY: Institute of Electrical and Electronics Engineers, 2000.

Kruchten 95 Kruchten, P. "The 4+1 View Model of Architecture." *IEEE Software 12,* 6 (November 1995): 42-50.

Kruchten 01 Kruchten, P. *The Rational Unified Process: An Introduction, Second Edition.* Boston, MA: Addison-Wesley, 2001.

Medvidovic 02 Medvidovic, N.; Rosenblum, D.; Redmiles, D.; & Robbins, J. "Modeling Software Architecture in the Unified Modeling Language." *ACM Transactions on Software Engineering and Methodology (TOSEM) 11,* 1 (January 2002): 2-57.

OMG 01 Object Management Group. *OMG Unified Modeling Language Specification V. 1.4.* http://www.omg.org/docs/formal /01-09-67.pdf (September 2001).

OMG 03 Object Management Group. *UML 2.0 Superstructure Specification: Final Adopted Specification.* http://www.omg.org/docs/ptc /03-08-02.pdf (August 2003).

Selic 98 Selic, B. & Rumbaugh, J. "Using UML for Modeling Complex Real-Time Systems." http://www-106.ibm.com/developerworks /rational/library/content/03July/1000/1155 /1155_umlmodeling.pdf (March 1998).

Shaw 96 Shaw, M. & Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline*. Upper Saddle River, NJ: Prentice-Hall, 1996.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. **AGENCY USE ONLY** (leave blank)	2. **REPORT DATE** April 2004	3. **REPORT TYPE AND DATES COVERED** Final
4. **TITLE AND SUBTITLE** Documenting Component and Connector Views with UML 2.0		5. **FUNDING NUMBERS** F19628-00-C-0003
6. **AUTHOR(S)** James Ivers, Paul Clements, David Garlan, Robert Nord, Bradley Schmerl, Jaime Rodrigo Oviedo Silva		
7. **PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)** Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213		8. **PERFORMING ORGANIZATION REPORT NUMBER** CMU/SEI-2004-TR-008
9. **SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)** HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116		10. **SPONSORING/MONITORING AGENCY REPORT NUMBER** ESC-TR-2004-008
11. **SUPPLEMENTARY NOTES**		
12.a **DISTRIBUTION/AVAILABILITY STATEMENT** Unclassified/Unlimited, DTIC, NTIS		12.b **DISTRIBUTION CODE**

13. **ABSTRACT** (maximum 200 words)

The widespread presence of the Unified Modeling Language (UML) has led practitioners to try to apply it when documenting software architectures. While early versions of UML have been adequate for documenting many kinds of architectural views, they have fallen somewhat short, particularly for documenting component and connector views. UML 2.0 has added a number of new constructs and modified some existing ones to address these problems. In this report, we explore how changes in this version affect UML's suitability as a notation for documenting component and connector views.

14. **SUBJECT TERMS** software architecture documentation, UML, architecture view, components and connectors		15. **NUMBER OF PAGES** 58
		16. **PRICE CODE**

17. **SECURITY CLASSIFICATION OF REPORT** UNCLASSIFIED	18. **SECURITY CLASSIFICATION OF THIS PAGE** UNCLASSIFIED	19. **SECURITY CLASSIFICATION OF ABSTRACT** UNCLASSIFIED	20. **LIMITATION OF ABSTRACT** UL