Dimensions of Component-based Development

Colin Atkinson[†‡], Thomas Kühne[†] and Christian Bunse[‡]

[†]Universität Kaiserslautern
{atkinson, kuehne@informatik.uni-kl.de }

[‡]Fraunhofer Institute for Experimental Software Engineering
{atkinson, bunse@iese.fhg.de}

ABSTRACT

As the properties of components have gradually become clearer, attention has started to turn to the architectural issues which govern their interaction and composition. In this paper we identify some of the major architectural questions affecting component-based software development and describe the predominant architectural dimensions. Of these, the most interesting is the "architecture hierarchy" which we believe is needed to address the "interface vicissitude" problem that arises whenever interaction refinement is explicitly documented within a component-based system. We present a solution to this problem based on the concept of stratified architectures and object metamorphosis Finally, we describe how these concepts may assist in increasing the tailorability of component-based frameworks.

1 INTRODUCTION

Much of the recent debate on component-oriented software development has naturally revolved around the question: "*what* is a component?" Less attention has been given to the architectural issues related to the structure of component-based systems, and the nature of the key relationships which drive component-based development - in essence, to the question "*where* is a component"? Addressing this question we believe will not only help establish a cleaner and more general theory of components, but will also shed light on the "what" question by helping to clarify important characteristics of components.

We believe four fundamental hierarchies naturally dominate the structure of component-oriented software systems.

1. Containment hierarchy
2. Type hierarchy
3. Meta-hierarchy
4. Architecture hierarchy

The term hierarchy is used in a general sense here to represent a set of entities related by some transitive, partially ordered relationship.

The first three hierarchies may be termed "*intrinsic*", since they contain the actual components themselves. In other words, every component must be assigned a place in each of these hierarchies. This place is unique for each component and serves to define its properties and characteristics.

The fourth hierarchy, in contrast, can be thought of as "*extrinsic*" since it is not actually a hierarchy of components per se, but rather a hierarchy of "architectures" or "architectural strata." In other words, it is not the components themselves which are partially ordered, but the architectural strata in which they appear. This hierarchy therefore has more to do with describing how a component is used than on defining the nature of the component itself[1].

In the following sections we discuss each of these dimensions in more detail: section 2 describes the role of the component hierarchy, section 3 briefly talks about the type hierarchy, section 4 discusses the ramifications of the meta-level hierarchy, and finally section 5 introduces the concept of the architecture hierarchy and describes its potential benefits. Section 6 provides a summary of the key points, and an analysis of their implications.

2 CONTAINMENT HIERARCHY

The containment relationship is probably the most fundamental of those influencing the structure of component-based systems. It also has the largest number of different names, including "aggregation", "part-of", "includes", "embeds" and of course "composition". All these terms are used to convey the same underlying idea of "big" objects containing "small" objects. In fact, the very name component is intended to reflect the idea of containment.

Although simple in concept, containment is notoriously difficult to apply in practice. The problem is that 100% "pure" containment rarely occurs in the real word. Contained objects almost always have relationships to objects other than their container or fellow contained objects (i.e., they are shared by multiple containers or temporary clients), and often these can also represent some form of containment. Most object-oriented systems typically contain a tangled web of inter-object links, making the identification of a clear containment tree a non-trivial problem. In particular, it is often difficult to

[1] The containment hierarchy can actually can be thought of as playing a dual role in this sense, because as well as determining the nature of a component's interface it also plays a role in describing how it is deployed.

disentangle "containment" relationships from "uses" or "peer" relationships where no containment is intended.

Why not therefore simply de-emphasize (or ignore) the idea of containment in the structuring of object-oriented and component-based systems? To a certain degree this is the strategy adopted in the UML which views aggregation as a special case of association, and advises developers to use the latter whenever they are in any doubt as to the applicability of the former. While it may be possible to deemphasize containment between individual components, however, the idea of the eventual "system" containing the components from which it is created seems inescapable. This idea is as fundamental as the word component itself.

This brings us to a critical question -

> *should the assembly of a system be viewed as a different activity (i.e. use different concepts and techniques) from the assembly of a component?*

In other words, should the application (or use) of a component be viewed as involving different concepts and techniques than the creation of a component? Most approaches to component-based development do not explicitly address this question, but their terminology implies that they view the two as different activities. In other words, most approaches view a system as being a different kind of entity from a component.

We believe this to be fundamentally at odds with the philosophy of component-based development. There seems to be no good reason why an assembly of components developed to meet the requirements for a "system" should not at a later stage also be viewable as an individual component, should their collective services be useful in the creation of a larger system (i.e. as a component). However, if one accepts the metaphor:

"a system = a component"

one is compelled to provide a uniform component model which treats all components in the same way regardless of their location in the composition hierarchy or whether they are used as a system or as a part of a system. The only factor which should determine the activities and concepts applied to a component should be relevant the requirements (functional or non-functional).

3 TYPE HIERARCHY

Another hierarchy that plays a fundamental role in component-based development is the type hierarchy. As in object-oriented approaches, the basic idea of a type is to control the linking together and interaction of components based on some form of explicitly specified set of expectations (i.e. a contract). Like containment, the idea of a type also goes by various names, the chief among them being "role", "class", and "interface". These concepts all essentially serve to define a set of expectations that govern interactions and relationships between objects. They also can placed into hierarchies which organize such "expectation specifications" in terms of their commonalties and differences. These hierarchies also go by various different names, including type hierarchy, role hierarchy and interface hierarchy.

In most existing component technologies a component type (i.e. interface) is embodied by the set of operations that the component exports, and the information which these operations receive and return (i.e. parameters). Exception definitions are also sometimes included. While this provides a rudimentary way of defining expectations, it leaves a lot of information missing. For example, the typical interface specification says nothing about the expected effects of operations, or the expected interleaving of operations. Guaranteed substitutability of components, which is the underlying motivation for typing, requires that the client and supplier of a service be in complete agreement about the full nature of the expectations to be satisfied.

The "system = component" metaphor mentioned in the previous section, suggests one way of approaching this problem; namely, to model a component interface by a suite of UML diagrams as if the component were a system. Various analysis/design methods present ways of using UML (or equivalent) diagrams to described the requirements satisfied by a system, so it would seem reasonable that these might also be useful for modeling interfaces. At the Fraunhofer Institute for Experimental Software Engineering we are investigating an approach based on the diagram suite defined in the Fusion method [1] as adapted for the UML by FuML [2].

4 META HIERARCHY

Metamodeling has become fashionable. However, many of the approaches which claim to be based on meta modeling fail to follow through with the full implications. The best example is the UML [3], which ostensibly assumes the four level modeling framework illustrated in Figure 1. Each layer, except the bottom layer, represents a model (i.e. a class diagram) instantiated from the elements described in the layer above. The exceptions to this rule are the bottom layer, which contains the actual objects embodying the data of the user, and the top layer which is regarded as an instance of itself. Normal user class diagrams reside at the second level, immediately above the bottom "data" layer.

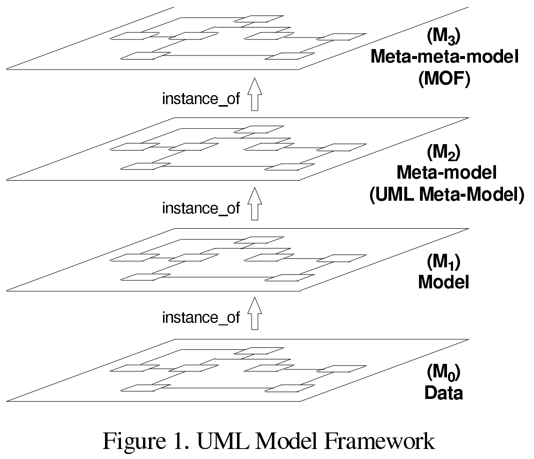


Figure 1. UML Model Framework

The main consequence of this approach for components is that elements in all but the bottom layer generally have the properties of both an object and a class (i.e. they are clabjects [4]). This is because they represent a template for instantiating instances at the level below, and at the same time they are themselves instances of templates from the level above. This dual faceted view of components is depicted in figure 2.

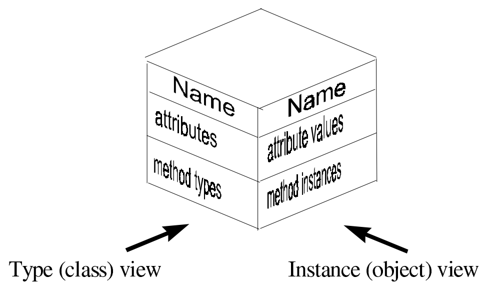


Figure 2. Class/Object View of a Component

Most approaches that adopt such a multi-layered model hierarchy, such as UML and OPEN, ignore this fact because it leads to some awkward consequences. Ironically, however, this dual object/class facet could actually help address a problem that has been central to the component debate for some time; namely "is a component an object or a template (from which objects can be created)"? Some authors, such as Orfali et. al. view a component as an object with certain additional properties [5], but others such as Szyperski, believe that a component is not an object, but can only be used to instantiate objects [6]. If one accepts the class/object duality implied by a rigorous multi-level modeling framework, the most general answer would be that a component is both.

The phrase "most general" is here because not all components will necessarily have both facets all of the time. However, the class/object duality occurs more often than might be expected. For example, components which are primarily intended to provide a template for instantiating objects typically tend to have some "static" information, such as a serial number, which essential corresponds to attribute values in the object facet. In the UML, such attributes are called "tagged values", while in programming languages they are called "static data members". The only difference from normal attributes is that that they are not usually changed at run-time.

Similarly, components which are primarily intended to serve as objects (e.g. CORBA objects), often have an associated reflection API which can be used to provide access to certain kinds of "static" information. Also, environments such as CORBA usually store "meta information" about running objects, typically in interface repositories. These both essentially correspond to the template facet of the components.

Even with existing component technologies, therefore, explicit class/object duality may provide a natural and clean unifying model for handling the various characteristics of components and the different, often separated, pieces of information that are maintained about them. However, the possibility also remains that a pure and fully object-oriented component model of the kind characterized by Smalltalk, in which every class has an explicit run-time presence, may offer one of the best long term strategies for promoting component-based software development.

5 ARCHITECTURE HIERARCHY

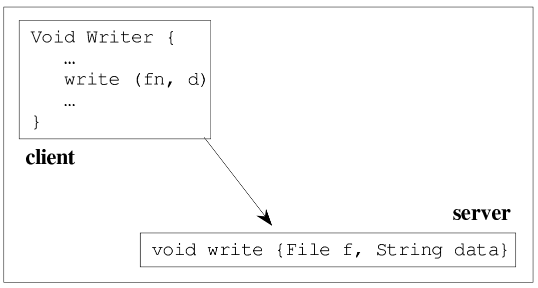
The three "dimensions" described in the previous sections are fairly conventional, and in one form or another appear in most existing component technologies. However, the fourth "dimension" described in this section is much less conventional, and as far we are aware does not exist explicitly in any of the current or proposed component-based development technologies. It also differs from the previous three hierarchies in that it is not a hierarchy of components per se. In order to explain precisely what it is rather than what it is not, we first need to elaborate upon the problem that it is aimed at solving.

5.1 The Problem

To illustrate the problem we will consider the classic scenario of communication between remote entities in a distributed system. The example will be based on a simple client/server scenario in which a file manger (server) supports requests to read and write strings to and from files. We will consider both function-oriented and object-oriented version of the system, in both localized and distributed forms.

5.1.1 Localized File Management System

As might be expected, the localized, function-oriented version of the system is the simplest. Figure 3 illustrates a client function, Writer, issuing a call to a server function, write.



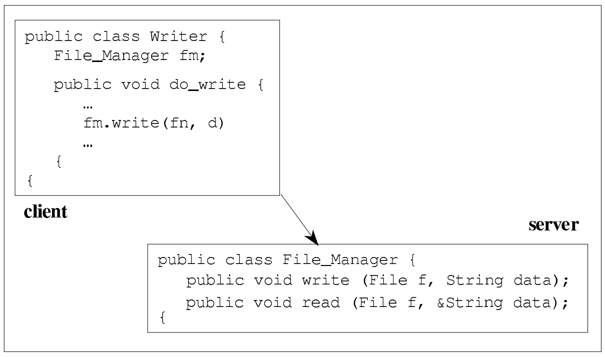
```
Void Writer {
   …
   write (fn, d)
   …
}
```

```
void write {File f, String data}
```

Figure 3. Localized Function-Oriented Form

The write function takes two parameters: a reference of type File serving to identify the target file and a String representing the data to be written to the file. The read server function is similar, but obviously the String parameter would have to be passed by reference in order to return the value. In this example, the actual file reference is fn and the actual string to be written is d.

In an object-oriented system all functions have to belong to objects. The basic difference in the object-oriented version of the system, therefore, is that the write and writer functions have to be defined as part of a class definition, as illustrated in figure 4 (write becomes do_write). The basic interaction is the same, however.



```
public class Writer {
   File_Manager fm;
   public void do_write {
      …
      fm.write(fn, d)
      …
   {
{
```

```
public class File_Manager {
   public void write (File f, String data);
   public void read (File f, &String data);
{
```

Figure 4. Localized Object-Oriented Form

The Writer and File_Manager classes in the object-oriented version of the system can also be depicted graphically. Figure 5 is an equivalent UML collaboration diagram which indicates that an instance of Writer, called w, sends a message write() to an instance of File_Manager called fm.

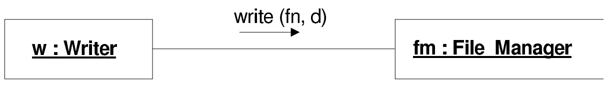
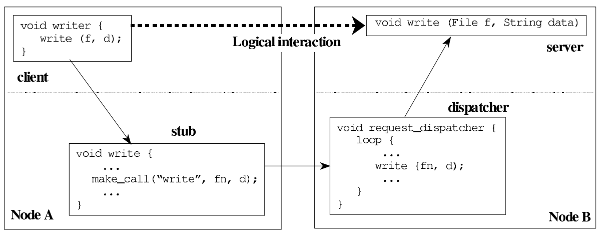


Figure 5 Localized UML Collaboration Diagram

5.1.2 *Distributed File Management System*

Whether written in a function-oriented or object-oriented style, if the client and server are on the same machine the compiler can simply link all the appropriate components into a single program, and the interaction between them will be implemented directly as a normal, local function (or method) call.

However, if the file manager and writer need to execute on different machines, things get a little more complicated. It is now necessary to arrange for the communication to be implemented via the network.



```
void writer {
   write (f, d);
}
```

```
void write (File f, String data)
```

```
void write {
   ...
   make_call("write", fn, d);
   ...
}
```

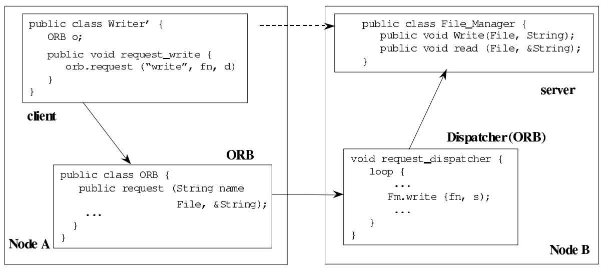
```
void request_dispatcher {
   loop {
      ...
      write {fn, d);
      ...
   }
}
```

Figure 6. Distributed Function-Oriented Form

A well-known and widely used strategy for implementing remote communication is to use a "stub, as depicted in figure 6. Instead of calling the server function directly, as in the localized system, the client instead calls the special "stub" which arranges for the interaction to be implemented in terms of the communication services supported by the network. Notice that the name of the function to be called now has to be passed as a parameter to the remote dispatcher to enable it to decide which of its local functions to call. In some circumstances such stubs can be generated automatically, but in others it may have to be coded by hand. In either case, the stub is linked into the client's program instead of the original implementation of the server function.

On the server's side, some form of request dispatcher (a.k.a. entry port) is needed to receive incoming messages and call the original server function on the remote client's behalf. This is the request_dispatcher illustrated in figure 6. The job of this entity is to respond to incoming service request by decoding the message and invoking the appropriate function.

This same idea can of course be applied in the object-oriented version of the system. In fact, this is the basis of the ubiquitous "request broker" technology underlying CORBA and other distributed object environments.



```
public class Writer' {
   ORB o;
   public void request_write {
      orb.request ("write", fn, d)
   }
}
```

```
public class File_Manager {
   public void Write(File, String);
   public void read (File, &String);
}
```

```
public class ORB {
   public request (String name
                    File, &String);
    ...
   }
}
```

```
void request_dispatcher {
   loop {
      ...
     Fm.write {fn, s);
      ...
   }
}
```

Figure 7. Distributed Object-Oriented Form

As illustrated in figure 7, the job that previously fell to the stub[2] in the function-oriented version of the system now falls to a method of the ORB. In this example the method is called request(). The body of this method is essentially equivalent to the stub, and sends the appropriate information over the network in order to implement the required interaction.

The job of the request dispatcher at the other end is also played by an orb. ORBs therefore play the general role of mediators between remote objects which wish to interact. The example is a little artificial since the ORB methods have parameters which are specific for this application, whereas in general of course they would be more generic. Figure 8 provides a UML interaction diagram for the implementation illustrated in figure 7.

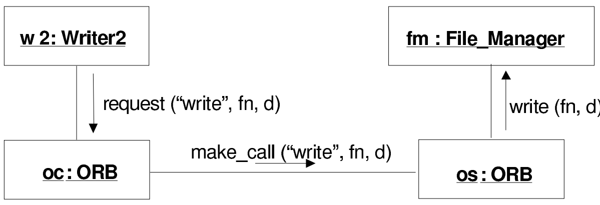


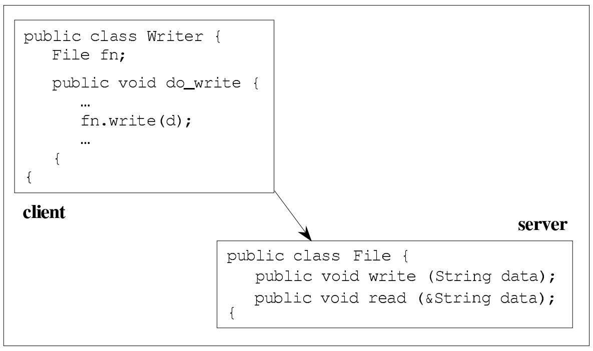
Figure 8. Distributed UML Collaboration Diagram

[2] Different distributed object technologies use words such as "stub" and "proxy" in non-standard ways. In this discussion we use the word in a general sense, not in the technical sense of any particular distributed object standard (e.g. COBRA, Java RMI etc.).

5.1.3 Interface Vicissitude[3]

So what is the problem? The basic issue is that in the object-oriented (and hence component-oriented) version of the system, the interface between objects can change depending on the level of abstraction at which the interaction or relationship between them is described. This can be seen by comparing figures 4 and 7, or their graphical UML equivalents, 5 and 8. In figures 4 and 5, the client, Writer, has an interface with File_Manager in which it invokes the operation write(). In a distributed implementation, this interaction might be referred to as the *logical* interaction. However, in figures 5 and 8, by contrast, the client, Writer, has no interface with File_Manager at all, but instead has an interface with ORB, in which it invokes the request() operation.

The phenomenon is not confined to the implementation of distributed communication, or to just two architecture levels. On the contrary, it occurs whenever an abstract interaction is refined into a more detailed description involving lower level components and less abstract interactions. Examples include transactions, security, persistence etc. - in fact, almost any service provided by component-based environments such a CORBA. The idea can also obviously be generalized to multiple levels. In fact, the interaction described in this example can easily be generalized to a third level by viewing the type, File, as a "persistent" class type rather than as a simple reference type and treating the write() operation as a method of this class rather than File_Manager. This would give the following view of the interaction illustrated textually in figure 9 and graphically in figure 10.



```
public class Writer {
   File fn;

   public void do_write {
      …
      fn.write(d);
      …
   {
{
```

```
public class File {
   public void write (String data);
   public void read (&String data);
{
```

Figure 9. "Persistent Class" Object-Oriented Form

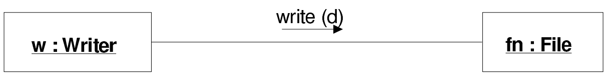


Figure 10. "Persistent Class" UML Collaboration Diagram

If we think of the structure and interactions described by the preceding figures as representing the architecture of the system (which in essence is what is meant by "architecture"), this means that the system can be considered to have different architectures at different levels of abstraction. Figures 4 and 5 represent descriptions of the architecture of the system (the first textual, the second graphical) which are equally as valid as figures 5 and 8 (and figures 9 and 10), the only difference is the level of abstraction at which the interaction to write information to a file is described. This would perhaps not be such an issue if the properties of the component involved in each view remained constant, but this is not the case — the interface[4] of the user component Writer is completely different in each case. In other words, the interface of Writer changes depending on which architectural perspective it is viewed from. This is what we refer to as "interface vicissitude".

Why is this a problem? In this small example we have shown three equally valid views of the architecture of the system, each with different interfaces for the Writer component. This begs the question as to which of the architectures is the correct (or best) one, or alternatively which of the interfaces of Writer is the correct (or best) one? If only one is to be considered <u>the</u> architecture, which one is it and how is it chosen?

Of course, it is always possible to place a wrapper around an ORB in the style of the Adapter pattern to make it have the appearance of the final server. In this example, this would mean placing a "proxy" on the client side to present the File_Manager interface to Writer instead of the ORB interface. But this essentially represents an attempt to simulate one architecture in terms of another, and implies that for some reason one architecture (or interface) has been chosen as preferable to another. However, unless superior tools are available at the higher abstraction level, or the translation to the lower level is fully automated, inserting such proxies only serve to complicate the lower-level architecture and decrease its efficiency. The issue is one of architecture modeling, or conceptualization, rather than interface adaptation.

It is interesting to consider why this problem does not arise in function-oriented software architecture. The reason goes right to the heart of what differentiates function-oriented approaches from object oriented approaches; object identity. In the function-oriented versions of the system (figures 3 and 6) the interface between the writer and the File_Manager is not at all affected by the identity of the communicating partners. As a consequence, the real write() method can be replaced by a stub (to handle remote communication) without in any way affecting the original communicating parties. This facilitates the creation of layered architectures of the kind characterized by the ISO Open Systems Interconnection model illustrated in figure 11. Because interactions at a given level can be refined without affecting the original communicating parties, clean layers can be established in which each module occupies one and only one layer.

[3] Vicissitude; n: regular change or succession of one thing to another, alternation; mutual succession, interchange (Webster's Unabridged Dictionary).

[4] The interface involved is often called the "required" or "imported" interface since it defines facilities used by the component rather than services provided for use by others.