T U M

INSTITUT FÜR INFORMATIK

Component Interface Diagrams: Putting Components to Work

Franz Huber, Andreas Rausch, Bernhard Rumpe

TUM-I9831
Dezember 98

TECHNISCHE UNIVERSITÄT MÜNCHEN

TUM-INFO-12-I9831-100/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Component Interface Diagrams: Putting Components to Work*

Franz Huber, Andreas Rausch, Bernhard Rumpe

email: {huberf,rausch,rumpe}@informatik.tu-muenchen.de

Technische Universität München
Arcisstr. 21, D-80290 München, Germany
Contact: Andreas Rausch
Tel: 49-89-289-28362, Fax: -28183

In this paper, we present *Component Interface Diagrams* as a notation to describe service access points (interfaces) of components, their structure, and their navigability. We give guidelines that allow to map the component model presented here to different technologies, like ActiveX, CORBA, and Java Beans. The framework FRISCO OEF, implemented in Java, illustrates the proposed component concept and proves its usefulness.

Keywords: Component Interface Diagrams, ActiveX, CORBA, Java Beans.

1 Introduction

The goals of ComponentWare are very similar to those of object-orientation. Software should be reusable in a convenient way, leading to various customization and configuration mechanisms. Also, implementation details should be hidden from the client as much as possible.

ComponentWare takes an even larger leap toward reusability, as components aim at a granularity much larger than single objects do. However, today the question what component concepts are, is still under investigation. This paper aims at clarifying the concept of components and demonstrates these ideas applied in the well structured framework FRISCO that was built using our component concept. The quality of that framework is considerably improved by the notion of components that we introduced.

To reuse as much as possible from already existing abstraction and encapsulation concepts, we build the component concept as an extension to object-orientation in Section 2. The notation of Component Interface Diagrams is introduced and applied to the

* This paper is joint work of the the project SYSLAB (supported by the DFG under the Leibniz Program, and by Siemens-Nixdorf), and the project "A1 Methods for Component-Based Software Engineering", supported by Siemens ZT, a part of "Bayerischer Forschungsverbund Software-Engineering (FORSOFT)".

FRISCO framework in Section 3. Finally, in Section 4 we discuss a mapping of our component concept to different object technologies, like ActiveX [Cha96], CORBA [OHE96] and Java Beans [Mic97].

1.1 A Brief Introduction into FRISCO OEF

FRISCO is a document-oriented software engineering tool prototype. It is based on a subset of UML notations [Gro97] but incorporates precisely defined refinement and transformation rules. FRISCO provides a variety of editors combining graphical and textual parts as well as tables within a single document. An example of a FRISCO editor is given in Figure 1.

To achieve flexibility we developed the OEF (**O**pen **E**ditor **F**ramework) as an open approach of nesting document parts into one compound document. The developed framework provides a standardized set of protocols for embedding documents. To structure these protocols, our notion of component interfaces is used.

For each document element, a specific kind of editor, called *PartHandler*, exists. Each *PartHandler* component consists of a possibly large set of internal objects implementing its functionality. A subset of these objects provides the protocol interface necessary for embedding it into the enclosing document frame. The interface objects hide the internal object structure of a *PartHandler*. They are the only way of communication with the environment. This framework, which has deliberate similarities to OPENDOC [App96], is implemented in Java, and the *PartHandlers* are realized as Java Beans.

2 A Model for Object-Oriented and Component-Based Systems

In this section we define an abstract model for object-oriented systems and extend this model to a component-based one, introducing the concepts of components and their structure. The model is used to clarify our notion of components and to give the notation proposed in Section 3 a semantics.

As a basic assumption, we regard an *object* to be an instance of a *class*. In a similar way we use the terms *component instances* and *component types* to refer to *instances* and to *property descriptions* that makes up components, respectively.

2.1 Properties of a Component-Based Model

The concept of components is built on top of object-oriented concepts, thus allowing to reuse them for components.

We do not enforce every entity of the system to be a component, but allow independent objects to live between components, just like global variables live between objects. Thus developers are free to choose what they want to be a component. Components may interact directly, but may also be glued together using independent objects.

Furthermore, the component concept must fit into the type system of the underlying language, such as in Java [GJS96]. As components are intended to be reused across

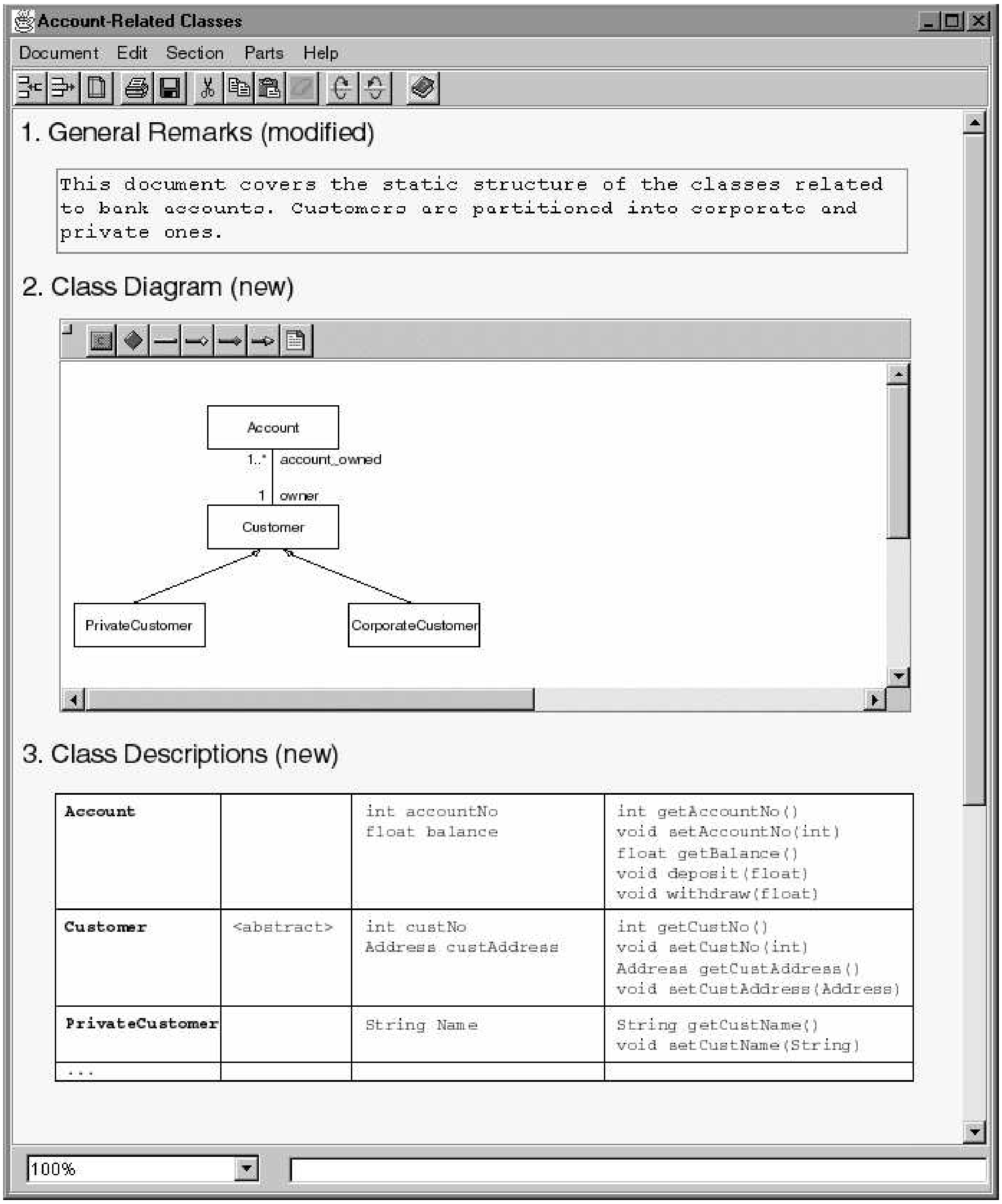

Figure 1: A Sample Screenshot of a Compound Document Editor in FRISCO

language boundaries, there should be a mapping of the component infrastructure into several type systems as, e.g., found in CORBA.

Components exhibit a characteristics similar to objects.

- Their instances can be dynamically created,
- they have a clearly defined interface, and
- they have a well structured state.

Beyond objects, they exhibit some additional features. A component has

- hierarchically structured interfaces,
- hierarchically structured states, and
- state and interface structure may change dynamically.

2.2 A Model for Object-Oriented Systems

In this section we present in an idealized and simplified form a model for an object-oriented system. Introducing it in a top-down way, we start with the definition for an object-oriented system and end with attributes, methods, and basic types, leaving out irrelevant details. Please note that this model for object-oriented systems is not complete, but sufficient for our purposes. It is defined in a way such that it fits different object-oriented languages.

Definition 1 Object Structure
An *Object Structure* $(Obj, \leadsto)$ is given by

- a set of objects $Obj \subseteq \mathbb{OBJ}$, and
- a relation $\leadsto: Obj \rightarrow Obj$, which denotes existing links between these objects.

In general $obj_1 \leadsto obj_2$ describes the existence of an unidirectional link from object obj_1 to object obj_2.

An object structure contains a set of objects Obj and links between them. These links are an abstraction and do not represent which attribute, parameter, or local variable is responsible, neither are multiple links represented. As links are unidirectional they describe accessibility. Since object-oriented systems change over time, an object structure describes a snapshot of a system.

The object structure need not be closed or complete. An object structure may contain a subset of existing objects and a subset of links. Therefore several object structures can describe different abstractions from an object-oriented system.

Definition 2 Object
An *Object* $(id, cl, Val) \in \mathbb{OBJ}$ can be represented by

- a unique identifier id for the object,
- the object's class $cl \in \mathbb{CLASS}$, and

- the valuation $Val \in \mathbb{VAL}$ for the attributes, local variables, parameters etc.

An object system contains a set of objects that may change over time, as objects are created or deleted. The valuations can be used to determine the linkage $\leadsto$ of the object structure. This definition of objects imposes several requirements, e.g., an object structure may not contain more than one object with the same identifier.

Definition 3 Class
A *Class* $(name, Meth, Attr) \in \mathbb{CLASS}$ is characterized by

- a unique *name* for the class,
- a set *Meth* of public accessible methods, and
- a set *Attr* of private accessible attributes.

In addition $\leq: \mathbb{CLASS} \times \mathbb{CLASS}$ is the inheritance relation for classes.

A class has a unique name, a set of public methods, and a set of private attributes. Public attributes can be simulated by methods. Private methods are used in programming languages to avoid re-writing code in several public methods. Hence there is no need for private methods or public attributes in our model.

With $\mathbb{VAL}$ the set of valuations for attributes and parameters are denoted. They are in essence mappings of variable names (attributes etc.) to values of appropriate type, characterizing the state of objects.

We do not elaborate on the underlying type system here, but assume an appropriate one to be given. In addition, to add a precise characterization of behavioral concepts, a mapping of the above given definitions into a system model as given in [KRB96] using state machines as behavioral entities [PR97, GKRB96] could be defined.

2.3 A Model for Component-Based Systems

Our model for a component-based system is introduced on top of the model for object-oriented systems.

Definition 4 Component
A *Component* $(name, os, pr, If, Int)$ is given by

- a unique *name* for the component,
- an underlying object structure $os = (Obj, \leadsto)$,
- the principal object $pr \in If$ of the component,
- a set of interface objects $If \subseteq Obj$, and
- a set of internal objects $Int = Obj \backslash If$.

A component denotes a snapshot of an object structure os, characterizing the internal structure, linkage etc. os contains a set of internal objects *Int* and a set of interface objects *If* that are referenced from the environment.

The lifecycle of the component instance is exactly the lifecycle of the the *principal* object pr. Other components and objects can access a component via the principal object. From the principal object they can receive links to other interfaces of the component. This way, a complex interface structure to the component can be obtained.

Once a reference of an internal object given to the environment, this object is no longer internal, but belongs to the interface of the component. Thus, the interface of the component is dynamically changing. The set of interface objects *If* denotes an snapshot of the component interface.

A *Component-Based System* is now characterized by a set of components, and an underlying object structure. Each component's internal object structure is a subset from that global object structure and objects internal to a component are not referenced from outside.

Definition 5 Component-Based System
A *Component-Based System* (Cp, os) is characterized by

- a set of components $Cp \subseteq \mathbb{COMP}$, and
- an underlying object structure $os \in \mathbb{OS}$.

We impose several requirements for meaningful component-based systems:

- Each component $c \in Cp$ has an internal object structure os_c that is an abstraction from the underlying object structure: $os_c \subseteq os$.
- Objects internal to a component are not referenced from outside.

Our experiences show that, in many cases, it is not necessary to use concepts of object migration between components. Since component-based systems usually have a rather static structure, it is sufficient to allow objects that have been internal to a component to "emerge" to the interface, thus allowing their access from outside. In general, it is not necessary for components to be tightly connected.

Objects that are created within a component belong to this component during their lifetime. We assume that objects are not explicitly destroyed but garbage collected which allows us to disregard dangling references and related problems.

3 Describing Components

So far, we have focused on providing a model for components. Now we introduce notations for describing them. As the UML [Gro97] provides a rich set of techniques for describing different views, we use and adapt these techniques for our purposes. Especially useful for describing components are the following notations:

Interaction Diagrams describe interactions either between objects in a component, or between components.

State Machines and hierarchical StateCharts [Har88] characterize the behavior of single objects within a component, but also of an abstraction of the entire component's behavior.

Interface and Class Declarations describe the methods and attributes, together with their types and access rights.

Class Diagrams are used to describe the possible structure of a system or a component.

Object (Structure) Diagrams define the static part of the internal structure of a component.

Our experiences show that a larger subset of the objects within a component has the same lifecycle as the principal object and does not change its linkage. Thus, the internal structure of a component is rather static and can be described by an Object Diagram.

Beyond the given UML notations, we propose an adapted version of Class Diagrams – the *Component Interface Diagrams* – that allows us to cope with the extended capabilities of component interfaces.

3.1 Frisco OEF Interfaces

In Frisco OEF several kinds of components are used. We now introduce and briefly describe a subset of the interfaces that *PartHandler* components provides, as Figure 2 illustrates.

BasicPartHandler is the principal interface that every *PartHandler* must provide. It covers rudimentary content and embedding functionality and allows to access additional interfaces of a *PartHandler*. To allow the enclosing document frame access to part information relevant for embedding, a number of methods are available to obtain information about content and size. Please note that this interface does not provide services for editing documents, since it is desirable that certain document parts should be displayed read-only.

Edit interfaces can be obtained invoking the *getEdit* method. This interface is provided only if the part is editable. It basically provides the services to externalize (save) its content and to activate and deactivate editing capabilities.

Toolbar interfaces allow access to the *PartHandler*'s toolbar. Two toolbars are allowed (one attached to the part, the other to the frame).

Undo allows a *PartHandler* to participate in the OEF Undo/Redo mechanism. After an *ActionListener* registers at the component, it receives a *UndoableAction* each time a change occurs.

Connection allows to access the interconnections between *PartHandlers* in the compound document, e.g., to propagate changes in order to ensure consistency between parts.

3.2 Motivation of Component Interface Diagrams

At the beginning of the lifetime of a component, the principal object (in FRISCO an instance of *BasicPartHandler*) is the only object that is accessible from the environment. Thus the interface of the component is initially given by the principal object. Over time, this may change. More objects may be created inside the component, and a reference to them may be given to the environment, leading to a dynamic extension of the component interface (see Section 2.3). This provides an important component property: being able to provide additional interfaces during runtime if required. The purpose of a *Component Interface Diagram* (CID) is to give clients a concise knowledge of the possible set of interfaces they may use.

Due to the requirement of strong typing, these interfaces may be created during runtime, but their type must be known initially. A CID gives information about the externally visible interfaces, their inheritance relations, and navigation paths between these interfaces. Furthermore, methods and multiplicities of these interfaces are shown.

CIDs are adapted from UML *Class Diagrams*. Figure 2 shows an *extended* CID for the *PartHandler* component. Let us forget about the arrows' labels for the moment and talk about the simple variant first.

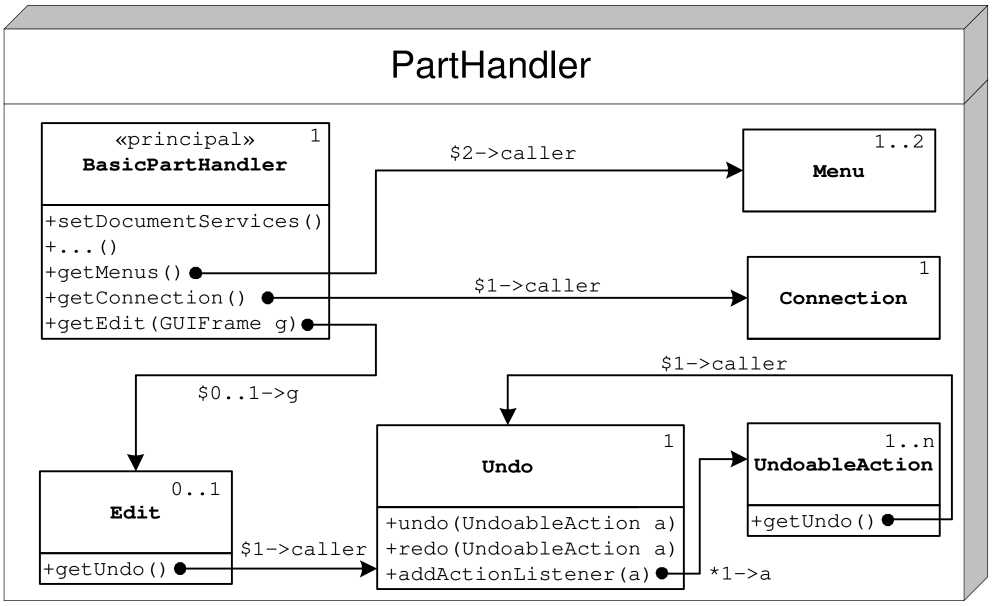

Figure 2: A FRISCO Component Interface Diagram

Disregarding labels, a CID contains externally visible classes, their inheritance relations, visible methods, and, in addition, multiplicities of possible instances. The multiplicity determines the maximum allowed set of interfaces during runtime. In addition, navigation paths are introduced as a concept to indicate the possible paths where to navigate from one interface to another. Such navigation is usually done by calling an appropriate method, which results in a reference to a new interface (see Section 3.4). Please note that these navigation paths are not associations, although an association might be the component's internal way to implement navigation.

The *PartHandler* in Figure 2 offers six externally visible interfaces, among them the principal interface marked with the appropriate stereotype. It also shows, *what* navigation paths between interfaces are possible, but not *how* navigation is done. It tells us, e.g., that from the *Edit* interface, the *Undo* interface can be obtained, and each component provides one or two menus (one is context-dependent, the other is optional).

The most important capability of components is the possibility to provide an entire set of individual and standard interfaces. Therefore, a classification of interfaces is a point of interest following two main goals:

- Separation of concerns for the component developer ending up with a more modular implementation than one monolithic interface could provide.
- Clearly separate individual and standard interfaces to give component users a more natural way of understanding the different purposes of the entire component.

The designer of a CID should structure the interfaces with respect to some methodical guidelines. This could be expressed in UML stereotypes for standard interfaces. For example, special interfaces for storage, printing, the undo/redo-mechanism, security, configuration, online help, testing and debugging are often useful. These standard interfaces are especially needed for component-based systems supporting plug-in of components, like, for instance, editors with exchangeable spell checkers.

The proposed CIDs give a first flavor of the interfaces of a component, but their expressiveness is limited. Therefore, we have enhanced CIDs to allow, e.g., to describe which methods are used to obtain new interfaces. However, this makes CIDs more complex, and it is therefore useful to work with both variants.

We introduce a transition labeling to describe how new interfaces can be obtained, whether we iteratively receive the same interface, or a new one for each request.

For example, calling *getMenus* on the principal interface returns one or two menu interfaces to the caller (`$1..2->caller`). Iterative calls result in the same interface for all callers (indicated by "`$`"). To indicate the creation of a new interface "`$`" is replaced by "`*`" (see method *addActionListener*).

A call of *getEdit* does not return an interface to the caller but to the method's parameter (`$1->g`) via another call. Please note, that such a "call back" need not take place immediately, but can be delayed (e.g. done by another thread). Furthermore, repeated "call backs" are allowed, as it is in the *Undo* interface, that allows to register *UndoActionListeners* (method *addActionListener*) that will receive a reference to an *UndoableAction* each time an undoable change occurs.

3.3 Precise Definition of Component Interface Diagrams

We now give a precise characterization of CIDs (the set of labels $\langle Lab \rangle$ used here is defined below):

Definition 6 Component Interface Diagram (CID)
A Component Interface Diagram $(\mathit{Ifc}, \sqsubseteq, \rho, \twoheadrightarrow, \rho)$ consists of a

- a set of interfaces $\mathit{Ifc} \in \mathbb{CLASS}$,
- an inheritance relation $\sqsubseteq: \mathit{Ifc} \times \mathit{Ifc}$
- a multiplicity mapping, $\rho : \mathit{Ifc} \to \langle \mathit{Multiplicity} \rangle$, and
- a labeled navigation relation $\twoheadrightarrow \subseteq \mathit{Ifc} \times \langle Lab \rangle \times \mathit{Ifc}$.

By $if_1 \overset{l}{\twoheadrightarrow} if_2$ we denote that there is a label $l \in \langle Lab \rangle$ in interface type $if_1 \in \mathit{Ifc}$ that allows clients to obtain an instance of interface $if_2 \in \mathit{Ifc}$ from this component, and the label tells how.

Definition 7 Labeling of a CID
The labels $\langle Lab \rangle$ of a CID are given by the following grammar:

$$
\begin{array}{ll}
\langle Lab \rangle ::= & \mathbb{METH}\ \boxed{\texttt{(}}\ \{\ \langle Param \rangle \parallel \boxed{\texttt{,}}\ \}^*\ \boxed{\texttt{)}}\ \langle Details \rangle \\
\langle Details \rangle ::= & [\langle Modifier \rangle]\ [\langle Multiplicity \rangle]\ [\ \boxed{\texttt{->}}\ \langle Receiver \rangle] \\
\langle Modifier \rangle ::= & \boxed{\texttt{*}} \mid \boxed{\texttt{\$}} \\
\langle Multiplicity \rangle ::= & [\ \mathbb{N}\ \boxed{\texttt{..}}\ ]\ \{\mathbb{N} \mid \boxed{\texttt{n}}\} \\
\langle Receiver \rangle ::= & \mathbb{VAR} \mid \boxed{\texttt{caller}}
\end{array}
$$

Whenever a modifier, multiplicity or receiver is missing, no constraint is assumed. Please note, that in the diagram the $\mathbb{METH}$-part of the label is attached to the source node, as this denotes the interface, where the method belongs to. Some straightforward context conditions apply and some combinations are useless, e.g., the multiplicity of the interface itself must at least equal the multiplicity of the labels of incomming arrows.

CIDs specify, which references to its objects a component can give to the environment. A careful flow analysis, as done for other purposes already in Java compilers, could proof correctness of the component implementation.

There are basic objects, such as Java Strings, that are publicly available (see Section 2.1). It is useful to exclude such basic classes from the component concept, but to let them float through component borders freely, regardless, where they have been created. However, such exclusion has to be done carefully, being aware of implicit communication via shared objects which could lead to a behavior that is not derivable through observation of component interfaces.

Given the technique of Component Interface Diagrams and the already mentioned notations of UML, we can define different views of components. With CIDs, we can define the *Black-Box View* of components. Class Diagrams are useful to specify the internal structure of a component, the so called *Glass-Box View*. With object diagrams we can specify run-time behavior of components as a object structure snapshot. The connection between these views is shown in Figure 3. Note that an interface in the CID can be implemented through several classes in the class diagram as well as an class can implement several interfaces.

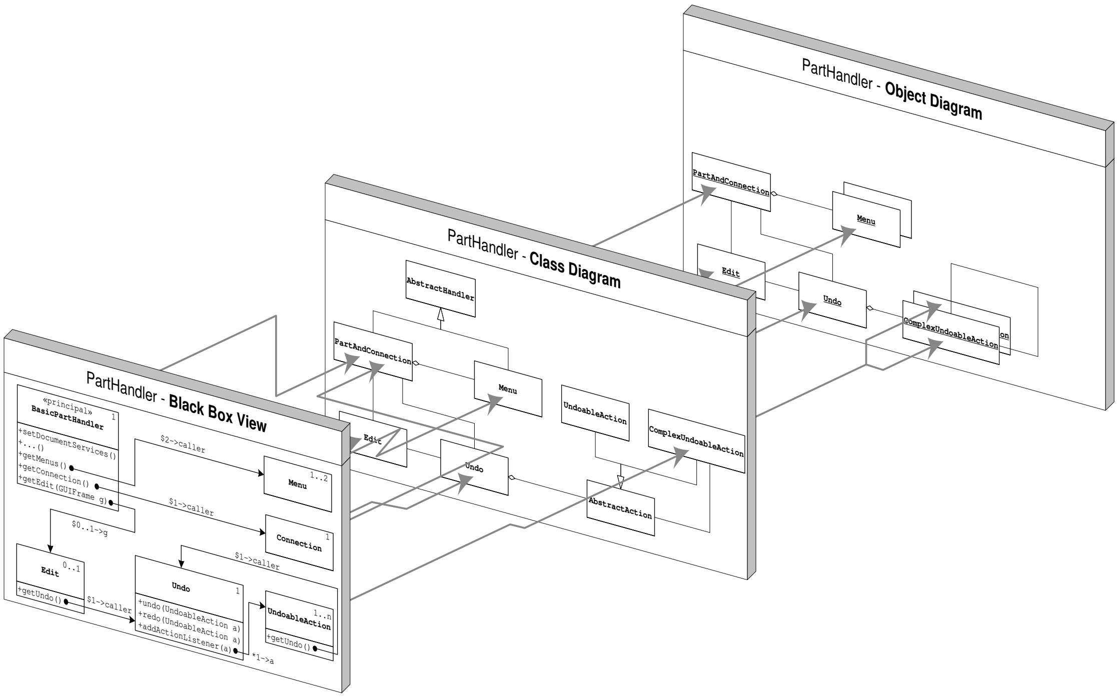

Figure 3: A Mapping between the Glass-Box and Black-Box View

As Figure 3 indicates, the semantics of a CID can be given as a mapping of the CID into an embedding Class Diagram, where all component interfaces map to classes, the inheritance relation and the multiplicities are preserved, and the navigation relation is mapped to method calls accordingly.

3.4 Guidelines to Map Components to Objects

Based on our experiences, we suggest the following guidelines for a mapping. In general there are three kinds of possibilities to implement navigation between interfaces.

We have focused on the preferable *method call*. But it is also possible to use public readable *attributes* for interface access if they are available, or a dynamic *cast* of a given interface into another interface. The latter is, e.g., possible in Java, where failed casts can be caught by an exception.

Component interface types are mapped either into Java classes or Java interfaces. The former has the disadvantage that classes are not abstract and thus can be instantiated from the environment, the latter cannot be used if attributes are publicly available in the interface. As we prefer methods for navigation, we suggest to use Java interfaces to implement CID interfaces.

When the desired multiplicity of an interface is 1 or a link has modifier `$`, then the interface needs to be stored after creation to be repeatedly exported. Its creation can either be done when the component is created, or in a lazy manner, upon the first request. Anyhow, these interfaces should be implemented following the singleton pattern [GHJV94].

If multiplicity is restricted, at least the number of already created interfaces needs to be stored. A proper reaction for too many requests is necessary: either returning `nil` or throwing an exception. The standard for too many requests is the latter one, the former one should be used to cope with optional interfaces.

The creation of a component goes along with the creation of its principal object. For that purpose, the creator must know the actual class of the principal. It is a good design principle to use equal names for the component and the pricipal class. Furthermore, there should be a global name service or an object factory (see [GHJV94] for clients to instantiate components).

Similar to aggregation of objects, we conceptually allow the hierarchical composition of components. However, our experiences show, that in practice, components will not be deeply nested. The composition of components is done by creating and using a component within another one.

4 Mapping the Component Model to Component Infrastructures

Today, three main component infrastructures are in practical use: Microsoft's *ActiveX*, based on OLE and DCOM [Cha96], several *CORBA* implementations [OHE96], and SUN's *Java Beans* [Mic97]. Since it is difficult to estimate at this time which technology will dominate in the future we subsequently characterize a mapping of CIDs in all three technologies.

For each technology, we discuss possible implementations of the component-based system shown in Figure 4. This system presents an abstraction of two Frisco components: The *PartHandler* (see Section 3.3, Figure 2) and a new component, the *DocManager*. The purpose of the *DocManager* is to observe its *PartHandlers* and propagate changes to related *PartHandlers*. If the method *registerAtPartHandler* is called the *DocManager* receives a pointer to the *Connection* interface (*getConnection*) and registers itself (*registerDocManager*). Afterwards, if a user edits any diagram, the corresponding editor component (*PartHandler*) notifies the *DocManager*, which then ensures that all other affected *PartHandlers* are informed of the change, eventually disallowing it, if it leads to inconsistent documents.

As all three technologies support a composition concept and provide an interface definition language – MS-IDL, IDL, and Java Interfaces – , a CASE tool supporting CIDs or similar description techniques could generate interface definitions for each technology.

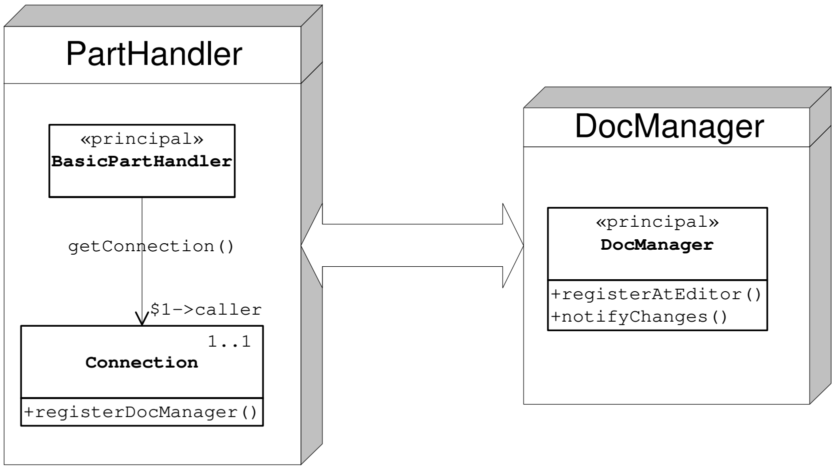

Figure 4: Interacting OEF Components

Hence, a mapping from our component based model to these technologies is basically possible.

4.1 ActiveX, OLE and DCOM

ActiveX controls, formerly known as OLE or OCX controls, are DCOM objects supporting a couple of standard interfaces. Minimally, OLE controls support two interfaces: One to search for additional interfaces, called *IUnknown*, the second to create new OLE controls, called *IClassFactory*. An ActiveX control supports several additional interfaces including initialization security, scripting security, run-time licensing, and digital certification [Cha96]. Moreover, DCOM offers additional standard interfaces, which can be implemented by DCOM objects, e.g., persistence interfaces, transaction interfaces, or drag & drop interfaces.

DCOM specifies a way of accessing objects via interfaces. Each DCOM object must provide at least the *IUnknown* interface, which allows clients to query and get access to other interfaces of the DCOM object. CID components are directly mapped to DCOM objects, whereas the DCOM object provides a DCOM interface for each CID interface. We also suggest to implement the CID navigation methods within the corresponding DCOM interfaces. Otherwise, DCOM's query interface mechanism must be used, thus sacrificing static type checking.

DCOM interfaces do not offer a concept for subtyping. Therefore, the subtyping mechanism for interfaces should not be used in CIDs if the target is DCOM.

In DCOM interface (types) have a unique identifier, but objects do not. To close this gap, DCOM introduces *Monikers* which allow to map DCOM objects to names. However, Monikers are insufficient for our purpose, as they are a crude way to establish connections between components (cf. [OH97]). Hence, we suggest to implement an own name service on top of DCOM or use standardized implementations, as, e.g., provided in CORBA.

To implement the example given in Figure 4 using DCOM each component is mapped into a DCOM object. Besides the standard DCOM interfaces *IUnknown* and *IClassFactory* each DCOM object has to provide its specific DCOM interfaces (*BasicPartHandler*, *Connection*, and *DocManager*). Clients can create the components by creating the principal DCOM interface via the class factory supported by DCOM.

4.2 CORBA

An ORB is a software bus: It allows objects to transparently request other objects, even if the target objects reside on ORBs of different vendors. Besides the distributed and language-independent, transparent access to objects, ORBs may offer a rich set of enhanced services. For instance, standard interfaces are specified for object and interface browsing, dynamic method invocation, object persistence, transaction management, or GUI services, which makes CORBA especially suited for our component concept.

CORBA interfaces are described using CORBA's *Interface Description Language* (IDL). A CORBA interface allows multiple inheritance, but a CORBA object cannot implement more than one interface. Instead, CORBA offers a module concept where interfaces can be grouped together into a specifc namespace, given by the surrounding module. CID components can have several interfaces. Hence a CID component has to be mapped to a CORBA module including all CID interfaces and navigation methods. In CORBA, CID components are thus reduced to simple name spaces.

Since CORBA provides a global name service, links between components and objects can be implemented in a straightforward fashion.

Mapping the example in Figure 4 to CORBA means to write two IDL modules–one for each component–including the corresponding IDL interfaces *BasicPartHandler*, *Connection*, and *DocManager*. After implementing the interfaces the two principal CORBA objects have to be registered at the CORBA name service, thus clients can access the components.

4.3 Java Beans

According to its creators from JavaSoft "A Java Bean is a reusable software component that can be manipulated visually in a builder tool" [Mic97, JT98]. This covers a wide range of different possiblities. The scope of functionality reaches from simple GUI parts, like buttons, up to full-featured database access adaptors.

In technical terms, a bean is a Java object. The specific characteristics of beans are:

A Public Interface offers *Properties*, *Methods*, and *Events* for clients to access the bean.

Introspection allows a builder tool to explore the bean's interfaces and present it to programmers. For that purpose, the *Java Reflection Technique* is used.

Customization allows developers to change the properties of beans during design-time.

Persistence is used to store the bean's state permanently and restore it later.

Beans can support additional features, such as, e.g., security, drag & drop, or remote invocation. To support several of these features, beans have to obey some conventions.

As beans are just Java objects, beans can implement several Java interfaces. This fits directly into our component concept, as we also allow several interfaces for each component and inheritance between interfaces. Beans also support single inheritance, which is not yet used for components in our model.

Beans are packaged in so-called JAR files that include, among code and other resources, optionally serialized bean instances. As the standard Java name service is a crude circumvention to establish links between bean instances in different JAR files, it is again necessary to define an own name service, or to use the new *Java Naming and Directory Interface* [Jav98], or even to use a bean-conformant infrastructure supporting a global name service, like, e.g., IBM's ComponentBroker [IBM98].

In our example, each CID interface is mapped into a Java interface. Two Java Beans–one for each component–must be realized. They should be registered at the global name service to allow clients access to them, particularly to enable other components to obtain links to them.

5 Conclusion

The proposed concept of components was defined as a result of designing and implementing the FRISCO framework for document editing. The high quality of FRISCO shows the suitability of the component concept. Although several extensions are imaginable, e.g., allowing object migration or defining a notion of inheritance on components (not only its interfaces), we expect the given notion of components to be sufficient for a large class of applications.

We feel that it is more important that language and tool support allow to conveniently define component types and automatically translate them into object-oriented implementations. This would considerably boost component technology.

6 Biographies

Franz Huber has been working in the area of software engineering tools since 1995. He is heading a project developing a tool for component-based development of distributed and embedded systems which combines the usage of informal and formal techniques. Additional research areas include object-oriented system modeling and development as well as methodical aspects of software and systems engineering.

Andreas Rausch is working on a research project aiming to develop methods for component-based software engineering. He has been heading in several industrial projects developing distributed information systems. Additional research areas include software architecture, distributed and component based systems, object-oriented modeling and development, and methodical aspects of software engineering.

Dr. **Bernhard Rumpe** is heading a research project aiming to narrow the gap between formal methods and practical modeling techniques. He has developed an approach including precise guidelines for refinement and composition of diagrams on a graphical basis, contributed to several papers about benefits and ways to formalize UML, and co-organizes workshops about similar themes e.g. at ICSE, ECOOP and OOPSLA.

References

[App96] Apple Computer Inc. *OpenDoc Programmer's Guide for the MacOS.* Addison-Wesley, 1996.

[Cha96] D. Chappell. *Understanding ActiveX and OLE.* Microsoft Press, 1996.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley, 1994.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[GKRB96] R. Grosu, C. Klein, B. Rumpe, and M. Broy. State Transition Diagarams. Technical Report TUM-I9630, Technische Universität München, 1996.

[Gro97] UML Group. Unified Modeling Language. Version 1.1, Rational Software Corporation, Santa Clara, CA-95051, USA, July 1997.

[Har88] D. Harel. On Visual Formalisms. *Communications of the ACM*, 31(5):514–531, May 1988.

[IBM98] IBM. *Component Broker Technical Overview.* IBM report, 1998.

[Jav98] JavaSoft. JNDI: Java Naming and Directory Interface. Version 1.1, Sun Microsystems, January 1998.

[JT98] H. Jubin and Jalapeno Team. *Cooking Beans in the Enterprise.* IBM report, 1998.

[KRB96] C. Klein, B. Rumpe, and M. Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model - . In J.-B. Stefani E. Naijm, editor, *FMOODS'96 Formal Methods for Open Object-based Distributed Systems*, pages 323–338. ENST France Telecom, 1996.

[Mic97] Sun Microsystems. Java Beans. Version 1.01, Sun Microsystems, July 1997.

[OH97] R. Orfali and D. Harkey. *Client/Server Programming with JAVA and CORBA.* John Wiley and Sons, 1997.

[OHE96] R. Orfali, D. Harkey, and J. Edwards. *The Essential Distributed Objects Survival Guide.* John Wiley and Sons, 1996.

[PR97] B. Paech and B. Rumpe. State based service description. In J. Derrick, editor, *Formal Methods for Open Object-based Distributed Systems.* Chapman-Hall, 1997.