Software Components: Interaction Protocols

Edelmiro Fuentes
Xavier Pintado

Abstract

Achieving software reuse to a significant extent has been an important goal for decades. However, as new techniques and methodologies are developed they seem invariably to fall short of expectations. Recently, component-based software construction has been promoted as an outstanding approach in terms of software reuse. Building applications by composing software components much in the same way as we put together Lego bricks seems appealing, indeed. Unfortunately, this is not as simple as the metaphor might suggest. This paper builds on our experience in building software components for financial institutions. Most of the components have been developed in C++ and rely on Microsoft OLE/COM mechanisms for inter-component communication and interaction. OLE/COM with their extensive support for programming language independence and standard inter-component communication mechanisms go a long way in promoting componentware as a practical discipline.

OLE/COM is based on a component model wherein a component appears essentially as a collection of interfaces. The interfaces expose the services provided by a component. As such they represent the essential resource for reuse. Interface-centered reuse has been the prevailing approach since the advent of object-oriented systems. The prominent role played by the components’ interfaces leads naturally to the standardization of interfaces as a way of expressing common functionality among a set of components. In this paper we explore another direction where the major role for reuse is played by protocols. A protocol is a mechanism that allows for the specification of patterns of interaction among components. Protocols have a state of execution and define the sequence of operations performed on the participating components. In order to be eligible to participate in activities governed by a protocol, components must expose interfaces that comply with the protocol’s requirements. In protocol-based systems, protocols become the unit of reuse. In fact, protocols express the dynamics of components interactions while interfaces represent components’ services. So by standardizing protocols instead of interfaces we set the focus on the reuse of interaction patterns rather than the reuse of services.

1 Introduction

In the late 1940's code reuse had already been recognized a major issue in software development. By those days, the notion of software componentry appeared with the invention of the subroutine library concept. The idea behind this concept was related to program productivity: how to avoid reinventing the wheel by reusing the software parts that have already been created to solve a similar problem. A similar concern was at the origin of Parnas [5] manifesto which paved the way for modern component-based software development techniques. His seminal paper promotes modular programming as a solution for both reuse and software maintainability and reliability. In 1982, Wasserman and Gutz [10] considered that the future of programming for the medium
term would include the development of certified software components (rigorously tested and well documented) that could be easily incorporated in new systems. The main objective is to create an environment in which code reuse would be the norm and not the exception.

It should be noted that significant levels of reuse have already been achieved in specific domains. Subroutine libraries, for example, have been extremely successful in terms of reuse, in particular, for programming mathematical applications. The reason for such a success is that most mathematical algorithms adhere naturally to a functional style which facilitates the reuse of libraries as collections of subroutines. Another example of successful reuse mechanism was introduced by the UNIX operating system with pipes and filters. Filters are stand-alone programs which perform transformations (filtering) on data streams. Such data streams are managed by pipes. By chaining filters through pipes a user can build a data transformation chain. This result is achieved by letting the output of a filter become the input to another.

![Diagram of data transform chain built with pipes and filters](image)

**Figure 1** A data transform chain built with pipes and filters

What seems to make filters and pipes such a useful reuse mechanism is that pipes assume a simple data type: a stream of bytes. This is the only common assumption but because it is so simple it is easy to reuse. The difficulty is that it is the responsibility of the user to provide data stream contents so that filter transformations produce meaningful results.

### 1.1 Interaction conventions

In our understanding one of the key issues in achieving reuse in the software development process is the definition of what is assumed to be common (standard) in a design framework. Interoperability among independent components implies some sort of common mechanism or common feature. Human interaction, for instance, requires a number of explicit and implied conventions such as a common language, common gestures and common thinking framework.

Any approach dealing with interoperability must face two aspects [4]:

1. control issues (coordination of the interoperating entities),
2. data issues (correspondences among the ways interoperating entities manipulate shared data).

Consequently, software component’s interaction requires the reuse of common interaction mechanisms. Different systems rely on different conventions to define such a common ground. Apple events [14] is an attempt to standardize typical operations on documents such as data bases, spreadsheets, and text editors. Standard verbs (syntactic keys) are associated with typical operations. Application designers are encouraged to reuse these verbs where appropriate so that services can be reused. As a consequence, components are interchangeable provided they ex-
pose the same set of verbs required by a client application. In this approach interoperability is grounded on service standardization. Microsoft OLE/COM follows, to some extent, a similar approach although less ambitious in term of services standardization. OLE/COM emphasizes on standardization of lower level interoperation mechanisms such as embedded documents, in-place activation and remote service invocation (OLE automation). However, both Apple events and OLE/COM provide a fairly desirable feature in terms of reuse: programming language independence. Programmers can choose the programming languages and tools that best fit their problem and their components are nevertheless guaranteed to interoperate with others provided they conform to the language-independent standard.

1.1.1 Object-oriented model

Object-oriented languages such as C++, Eiffel, and Java offer many mechanisms to support reuse such as inheritance and genericity. These programming languages promote a software components model where the components’ interfaces play the prominent role. In terms of reuse, a component boils down to the services exported by its interface. The interface plays the role of a contract between the component and its clients which states the services it can grant them. A component usually exports only one interface.

1.1.2 OLE/COM model

The components model promoted by OLE is slightly different. Components expose multiple interfaces. Furthermore, all OLE components are required to export at least the standard interface IUnknown. This interface provides rather rudimentary services related to component lifetime management and interface querying. Through IUnknown clients can ask a component for a specific interface. The component can either return a reference to the requested interface if it offers that interface or tell the client the interface is unavailable. However, there is no standard OLE interface that returns the set of interfaces exposed by a component.

The main reason why OLE components export multiple interfaces instead of a unique interface, as is common in most object-oriented programming languages, is that OLE defines standard reusable interfaces which are meant to be reused in different components. For example, all components intended to accept remote service invocation will adopt the standard IDispatch
interface. In general, an OLE component defines a set of interfaces which are specific to the services it provides (custom interfaces) and implements a set of standard OLE interfaces to manage standard interaction with other OLE components.

The factorization of a component’s services into distinct interfaces is thus dictated by the reuse of partial interface specifications defined by the OLE standard.

![OLE component model](image)

Figure 3 OLE component model. Each component exposes multiple interfaces. These interfaces are either specific to the component (custom interfaces) or standard OLE interfaces.

Although Microsoft OLE/COM is by any means the most widely used software component standard, other standards with more or less similar scope exist as well. For example, OpenDoc is supported by Apple, IBM, Novell and many other software vendors. CORBA is another component interoperation standard which is less vendor dependent. Although its main focus has been on remote service invocation it is gradually evolving towards a standard similar to OLE and OpenDoc. Finally, it is worth mentioning the forthcoming Java Beans as a language-dependent (Java) framework to build components. These standards focus on component interfaces as the essential reusable resource. By standardizing commonly used interfaces they promote the reuse of standard interface specifications. Component developers benefit from the adoption of standard interfaces because their components can be more easily reused.

### 1.2 Interface-centered reuse drawbacks

The interface-centered approach for software reuse has some major drawbacks, however. First, interface standardization omits one essential aspect of component-based software: components interaction. In fact, interfaces specify the services provided by components in isolation. The complexity of making independent components work together is simply ignored. Component integrators are presented with a set of interfaces but there is a lack of support for the description and organization of the possible interactions. However, developers know that the dynamics of component interaction is complex. Most interactions are more elaborate than simply requesting a service from a component. Interactions may involve more than two components. Interactions may encompass multiple exchanges from start to end. Complex interactions often go through many states before completion. Depending on the interaction’s state, different services provided by a component may not be available or they may not be meaningful in the context. These complexities make the components integrator task difficult and reduce the benefits of software reuse.
A practical problem related to component-based programming is that a substantial part of a component’s code deals with the interaction with other components. Putting interaction functionality inside the components that provide the services substantially reduces reusability and maintainability. This problem is particularly critical in object-oriented programming environments such as C++ and Java since the language does not offer mechanisms to state interactions outside the components. Interaction management functionality should appear outside components that provide other kinds of services. Interaction functionality can be specified, for example, by scripting languages or by building special purpose interaction components.

1.3 Specification of the dynamic aspects of an interaction

The specification of interactions is a well understood problem in the telecommunications realm. In telecommunications interactions are governed by communication protocols which specify the actions involved, the possible states of the interaction, and the actions that are triggered by state transitions. Although many aspects of telecommunication protocols cannot be meaningfully applied to software component interactions (e.g., time-outs), others such as interaction states, transitions, and actions seem fairly appropriate for the task. They allow for the specification of the dynamic aspect of component interactions. Next section discusses our proposal to introduce interaction protocols as a mechanism to manage interactions between software components. Our goal is to create a reusable interaction protocol layer above components interfaces that can be standardized so that instead of focusing on interface reuse we emphasize on interaction protocol reuse.

2 Protocols

In general terms, a protocol is a support that offers facilities to define compatibility and provides mechanisms to specify the rules of interaction between components.

An interaction protocol defines:

- the roles supported by the protocol (i.e. client, server). Roles are specified in terms of interfaces.
- possible states of the protocol, if any.
- actions associated with state transitions. The actions are specified in terms of messages sent to the interfaces of the participating software components.

The roles accepted by a protocol specify which kind of objects are allowed to participate in the interaction governed by the protocol. For example, an interaction protocol which defines the roles client and server specifies which are the components interfaces that are accepted to fulfill each role. It should be noted that a protocol does not prescribe the type of participating components. It defines a partial functionality requirement stated as interfaces. For instance, if the client role is associated to the standard interface IClient, any component that exposes this interface can participate in the protocol as a client.

Most protocols go through various states from start to completion. Protocol states allow for the definition, in terms of a finite state automaton, of which transitions and which actions are
allowed at each state. For example, a dragging protocol will not allow sending messages to a
destination component while the dragged icon is not inside a drag-sink candidate.

The rational behind interaction protocols stems from the observation that protocols can usually
be defined at a higher level of abstraction than component interfaces. The consequence is that
we can define a small set of typical interactions that can be reused in many different contexts.
Another advantage of protocols is that they encapsulate the complexity of dynamic aspects of
interactions.

2.1 Protocol definition

A protocol specifies the interaction between software components. A protocol \( P = (R, I, F) \) con-
sists of a set of roles, \( R \), an interplay relation, \( I \), and a finite state automaton, \( F \).

\[ P \] defines a set of roles:

\[ R = \{ R_1, R_2, ..., R_r \} \]

Each component that is \( P \)-compliant plays one or more roles A typical example of roles are
the client and server roles in a client-server protocol, where components can play either the cli-
ent’s role, the server’s role, or both depending on the specific responsibility assigned to the com-
ponents. In general, the number of roles defined by a protocol is small. A protocol also defines
an interplay relation that specifies the interaction compatibilities allowed by protocol \( P \). The in-
terplay relation is defined by a set:

\[ I = \{ I_1, I_2, ..., I_i \} \]

Moreover, if \( R = \{ r \} \), then \( I = \{ \{ I \} \} = \{ \{ r, r \} \} \). In words, it is always assumed for a one-role
protocol that all the software components obeying \( P \) are compatible in the sense that they are
able to interact under \( P \). Referring to the previous example, \( I = \{ \{ \text{server}, \text{client} \} \} \) specifies that
the protocol allows for the interaction between objects that play a server’s role and objects that
play a client’s role. To specify that the protocol also allows for the interaction between objects
that play the role of servers, the interplay relation should be specified as:

\[ I = \{ \{ \text{server}, \text{client} \}, \{ \text{server}, \text{server’} \} \} \]

Each object of the environment \( O_i \) eventually conforms to roles of one or many protocols. Lets roles (\( O_i \)) denote a function that returns the set of all roles component \( O_i \) conforms to. A
protocol \( P \) together with an element (i.e. a set of roles) \( I_k = \{ x_1, ..., x_i \} \), \( x \in R \) of its interplay
relation defines a domain of interaction compatibility \( D = (P, I_k) \). The compatibility defined by
a domain of interaction extends not only to the components that exist at the time the protocol is
defined and implemented, but also to all future components that obey the same protocol and are
compatible through an interplay relation. Finally, each protocol is associated with a finite state
automaton that specifies valid sequences of interaction between participants in the protocol.

3 Examples

This section illustrates interaction protocols with two examples. The first one is a generic drag
and drop protocol. The second example translates into a protocol the dynamic patterns of inter-
action of an existing windows application which relies on OLE mechanisms to offer dynamic extensibility.

### 3.1 Drag and Drop protocol

A drag operation is an operation initiated by a component, the dragging source, that attempts to find a partner component to cooperate with. The choice of the partner, the destination component, is performed by the user with the visual assistance of the windowing system. Both the dragging source and the dragging destination need to be associated with a visual representation since dragging is a visual operation. Figure 4 illustrates the finite state automaton associated with the dragging protocol, while table 1 shows the events that fire each state transition and the associated actions.

To simplify the understanding of how the protocol works it is useful to consult simultaneously figure 4, which shows the state transitions, and table 1, which exhibits the events that trigger a state transition together with the actions executed during the transition. The three boxes in the lower left corner of figure 4 represent the roles of the components that participate in the dragging process.

![Finite automata for the dragging protocol. The ellipses represent the states while the arrows represent state transitions. The three boxes at the lower left corner represent the roles of the components that participate in the interaction.](image)

The server is the component that initiates the interaction by sending the message `startDragging` to the protocol with its object identifier as parameter (see table 1, transition 0). Upon receipt of this message the protocol enters state `Start` followed by the execution of an action that makes the protocol send the message `startDragging` to the component that plays the `WindowManager` role, and assigns object identifiers to the destination and the source roles. The destination is assigned the void object identifier since at this stage the object that will play the
<table>
<thead>
<tr>
<th>State</th>
<th>Transition</th>
<th>State</th>
<th>Event/Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>0</td>
<td>Start</td>
<td>Source: startDragging{Source}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>startDragging{source} $\rightarrow$ WindowManager</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Source := source</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Destination := none</td>
</tr>
<tr>
<td>Start</td>
<td>1</td>
<td>In</td>
<td>WindowManager: dragCandidateEntered{candidate}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Destination := candidate</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>dragEnter{Source} $\rightarrow$ Destination</td>
</tr>
<tr>
<td>In</td>
<td>2</td>
<td>Pre</td>
<td>WindowManager: endDragging</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>preOperation{Source} $\rightarrow$ Destination</td>
</tr>
<tr>
<td>Pre</td>
<td>3</td>
<td>Oper</td>
<td>Destination: ACK{destination} $</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>operation{source} $\rightarrow$ Destination</td>
</tr>
<tr>
<td>Oper</td>
<td>4</td>
<td>Post</td>
<td>Destination: ACK{destination} $</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>postOperation{Source} $\rightarrow$ Destination</td>
</tr>
<tr>
<td>Pre</td>
<td>5</td>
<td>End</td>
<td>Destination: NACK{destination}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>slideDragViewBack $\rightarrow$ WindowManager</td>
</tr>
<tr>
<td>Oper</td>
<td>6</td>
<td>End</td>
<td>Destination: NACK{destination}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>slideDragViewBack $\rightarrow$ WindowManager</td>
</tr>
<tr>
<td>Post</td>
<td>7</td>
<td>End</td>
<td>Destination: ACK{Destination}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>operationComplete{destination} $\rightarrow$ Source</td>
</tr>
<tr>
<td>In</td>
<td>8</td>
<td>Out</td>
<td>WindowManager: dragCandidateExit{candidate}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>dragExited $\rightarrow$ Destination</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Destination := none</td>
</tr>
<tr>
<td>Out</td>
<td>9</td>
<td>In</td>
<td>WindowManager: dragCandidateEntered{candidate}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Destination := candidate</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>dragEnter{Source} $\rightarrow$ Destination</td>
</tr>
<tr>
<td>Out</td>
<td>10</td>
<td>End</td>
<td>WindowManager: endDragging</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>dragAborted $\rightarrow$ Source</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>slideDragViewBack $\rightarrow$ WindowManager</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Source := none</td>
</tr>
</tbody>
</table>

*Table 1* Dragging protocol transition table
destination role is not yet determined. The WindowManager responds to the first message by sending back to the protocol the dragCandidateEntered message. The reception of this message triggers state transition 1 on the protocol. The candidate object identifier that is sent as parameter corresponds to the source component since at the beginning of the drag operation the mouse is over the visual representation of that component. Consequently, the first component that is assigned the destination role is always the same component as the one that plays the source role. Later, the assignment will change as the user drags the mouse out of the source visual representation to enter another visual representation (i.e. icon) that is associated to a software component that accepts dragging. In the process of finding the appropriate destination component, the user may move the mouse in and out of visual representations that accept dragging. This process corresponds to alternations between state IN and state OUT.

If the user releases the mouse button when the protocol is in state OUT, then the dragging operation stops with no side effects since the mouse has been released outside a visual representation that accepts dragging. Conversely, if the mouse is released when the protocol is in the IN state, the protocol undergoes state transition 2 which puts the protocol in state PRE. This state corresponds to a pre-operation that is usually a negotiation between the source and destination components to agree on an operation to be performed. If both agree, the protocol transits to state OPER, which corresponds to execution of the agreed operation between the source and the destination. If no agreement is reached, then the dragging operation will end through transition 5. State POST allows for post-operation cleanup before the interaction ends.

We may notice that state IN and state OUT correspond to the visual process of establishing a relationship between two software components: the source and the destination. Likewise, states PRE, OPER, and POST manage the negotiation and execution of an operation between two components. The dragging protocol illustrates the generality and usefulness of an interaction protocol specified as a finite state automaton.

### 3.2 File Viewer

An interesting example of protocol reusability is given by the QuickView shell extension supported in Windows ®95 and Windows NT ®4.0 [15]. The purpose of this extension is to let users visualize the contents of an arbitrary file directly from the shell. The idea underlying QuickView is to have a generic application display any kind of file. To achieve this result, the architecture of the application is such that it can be automatically extended without any modification. QuickView adopts the following schema: First, it analyzes the type of the selected file (based on its extension). Second, it looks up in the system registry to find the location of the appropriated viewer. Third, it creates the viewer object and makes a series of calls to have the viewer object display the file. From the user’s point of view, QuickView looks like a single application (figure 5). From the point of view of its functionality, it is an empty application that implements a protocol.

Technically speaking, a viewer is a COM object that implements the OLE interfaces IPersistFile (methods that concern file storage) and IFileViewer (methods for displaying a file) in a DLL (Dynamic-Link Library) that is loaded at runtime. Once the corresponding DLL is loaded,
the protocol implemented in QuickView calls some of the methods supplied by those interfaces in the order shown in table 2.

<table>
<thead>
<tr>
<th>Order</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IPersistFile::Load</td>
<td>Opens the specified file and initializes an object from the file contents.</td>
</tr>
<tr>
<td>2</td>
<td>IFileViewer::ShowInitialize</td>
<td>Determines whether a file can be displayed and performs initialization operations.</td>
</tr>
<tr>
<td>3</td>
<td>IFileViewer::Show</td>
<td>Displays a file</td>
</tr>
</tbody>
</table>

Table 2  Order and description of methods called by the QuickView application.

4 Conclusions

This paper presents an approach for component-based software design and implementation based on protocols as a mechanism to specify patterns of interaction among software components. We take as a starting point Microsoft OLE/COM where the most commonly used component interfaces have been standardized to provide a coherent, language-independent framework that promotes interface reuse. We go a step further and define on top of component interfaces a mechanism called an interaction protocol. Interaction protocols define the role of participating components, the allowed states of the protocol and the actions associated with each state transition. Interaction protocols address the dynamic aspects of component interaction. An interesting feature of protocols is that they can be defined at a high level of abstraction allowing for the definition of a small set of protocols that can be reused in many different contexts.

We are currently implementing interaction protocols on top of Microsoft OLE/COM. The implementation relies essentially on COM for services such as the registry access and the dynamic instantiation of software components which are implemented as dynamic link libraries (DLL). One of our goals is to translate complex interaction patterns into protocols. Typical examples of complex interaction patterns are: interaction between an ActiveX component and its
container and interaction between an in-place edited document and its container document. These interactions are very complex and poorly documented in the sense that the only sensible way to implement them is to start with an existing implementation (this seems to be the way they have been designed). The translation of such complex interactions into protocols will allow us to formalize the design of the interaction patterns and identify possible design flaws such as redundant or incoherent states, too many states. We also expect to identify simplifications of the interaction scheme.

References


