Chapter 12

Gluons and the Cooperation between Software Components

Xavier Pintado

Abstract A major problem in software engineering is how to specify the patterns of interaction among software components so that they can be assembled to perform tasks in a cooperative way. Such cooperative assembly requires that components obey rules ensuring their interaction compatibility. The choice of a specific approach to specifying rules depends on various criteria such as the kind of target environment, the nature of the software components or the kind of programming language. This chapter reviews major efforts to develop and promote standards that address this issue. We present our own approach to the construction of a development framework for software applications that make use of real-time financial information. For this domain, the two main requirements are (1) to facilitate the integration of new components into an existing system, and (2) to allow for the run-time composition of software components. The goal of the development framework is to provide dynamic interconnection capabilities. The basic idea is to standardize and reuse interaction protocols that are encapsulated inside special objects called *gluons*. These objects mediate the cooperation of software components. We discuss the advantages of the approach, and provide examples of how gluons are used in the financial framework.

12.1 Introduction

The advent of object-oriented techniques has brought many benefits to the field of software engineering. One notable benefit is that objects provide a higher degree of autonomy than obtained with the traditional separation of software into functions and data structures. This autonomy promotes component-oriented software construction, since autonomous objects can be reused in many different context with reasonable integration efforts. Component reuse can reduce development time and costs, and can lead to improved reliability, since reusable components will become thoroughly tested as a consequence of reuse.

Although component-oriented software is fairly promising in terms of its reuse potential some major problems remain to be solved. Among these, a salient problem is the definition of the patterns of cooperation between software components, to which considerable effort has already been devoted. We may notice, for instance, that a class interface condenses assumptions about the objects that can be instantiated from it, but not assumptions about the interactions that those objects may have with other objects.

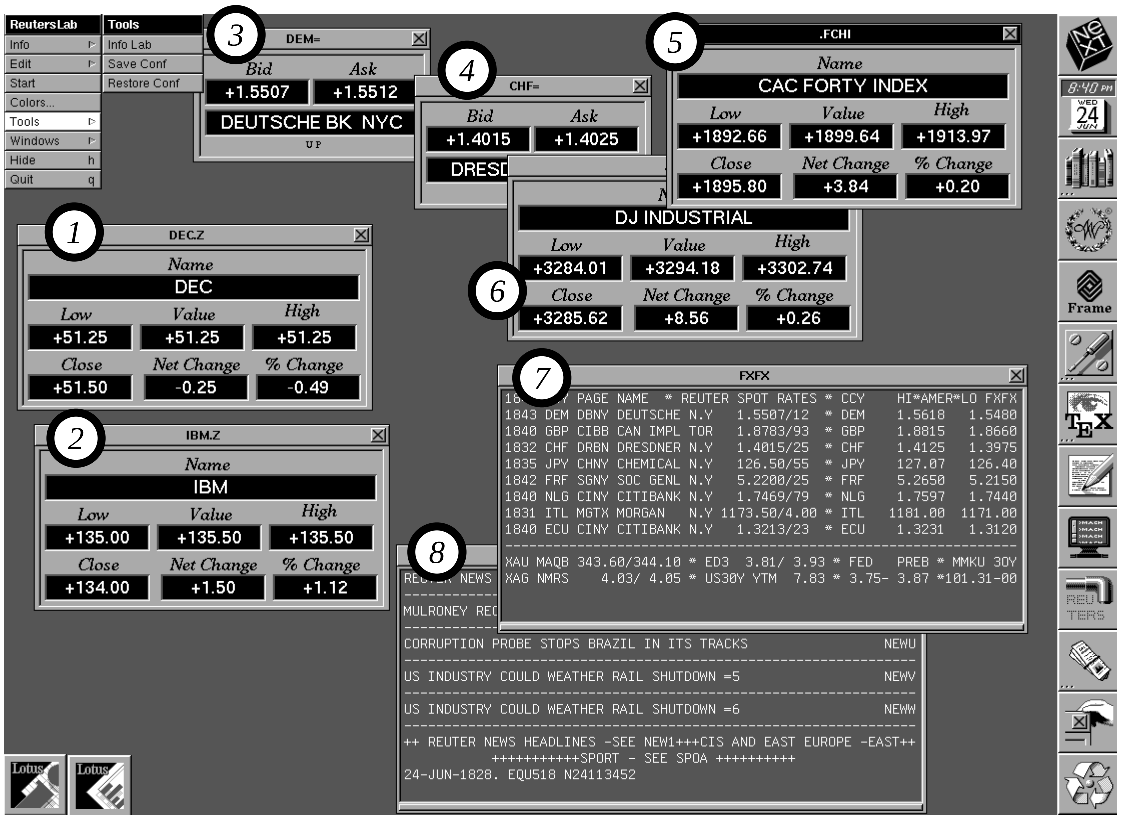
We may better capture the essence of the problem by observing that virtually any kind of cooperation requires agreement between the cooperating entities [29]. Cooperation agreements can take many forms, however. They can be specified, for instance, by a “law” to which all the cooperating entities obey. But cooperation can also rely on bilateral agreements each defining the cooperation between pairs of entities.

In the context of component-oriented software design, the goal is to make software components cooperate through reliable and flexible mechanisms that appropriately support and enforce convenient interaction patterns. In this context, the interaction “law” or cooperation agreement is usually captured by the notion of an object-oriented development framework [9] [10]. An object-oriented framework is a collection of classes that are designed to work together. A framework is intended to provide a development environment that promotes reuse and reduces development effort by providing a comprehensive set of classes and development rules. Frameworks come in many different flavours: they can, for example, target a narrow application domain such as the development of device drivers (e.g. NeXTStep Driver Kit [19]), or they can address the requirements of a generic development environment (e.g. Visual C++ framework [4]) comprising multiple sets of classes and development rules.

The distinguishing characteristic of a framework is the design philosophy that pervades all aspects of the framework such as the definition of foundation classes, the rules for the design of new classes and the tools that support the development process. By applying a consistent design philosophy to all the aspects of the framework, designers attempt to provide the user with a uniform development model that reduces the learning effort and defines a generic architecture for applications developed with the framework.

In this chapter we develop a framework for the development of financial applications. The framework is intended for the development of applications that involve the retrieval of real-time financial data sources. The typical target environment for the framework is rapidly evolving, in the sense that the behaviour of the objects and the way they are related evolves at a fast pace to reflect the real world of finance. The framework focuses on runtime connection of software components and on capabilities that support the incremental development of applications. Figure 12.1 shows a typical display of an application developed with the financial framework.

The distinguishing feature of the framework is the introduction of a special family of objects, called *gluons*, which are responsible for the cooperation among software compo-



DEM=

Bid	Ask
+1.5507	+1.5512

DEUTSCHE BK NYC
UP

CHF=

Bid	Ask
+1.4015	+1.4025

DRESD

.FCHI

Name: CAC FORTY INDEX

Low	Value	High
+1892.66	+1899.64	+1913.97
Close	**Net Change**	**% Change**
+1895.80	+3.84	+0.20

DEC.Z

Name: DEC

Low	Value	High
+51.25	+51.25	+51.25
Close	**Net Change**	**% Change**
+51.50	-0.25	-0.49

IBM.Z

Name: IBM

Low	Value	High
+135.00	+135.50	+135.50
Close	**Net Change**	**% Change**
+134.00	+1.50	+1.12

DJ INDUSTRIAL

Low	Value	High
+3284.01	+3294.18	+3302.74
Close	**Net Change**	**% Change**
+3285.62	+8.56	+0.26

FXFX

```
18     Y PAGE NAME   * REUTER SPOT RATES * CCY     HI*AMER*LO FXFX
1843 DEM DBNY DEUTSCHE N.Y   1.5507/12   * DEM    1.5618   1.5480
1840 GBP CIBB CAN IMPL TOR   1.8783/93   * GBP    1.8815   1.8660
1832 CHF DRBN DRESDNER N.Y   1.4015/25   * CHF    1.4125   1.3975
1835 JPY CHNY CHEMICAL N.Y   126.50/55   * JPY    127.07   126.40
1842 FRF SGNY SOC GENL N.Y   5.2200/25   * FRF    5.2650   5.2150
1840 NLG CINY CITIBANK N.Y   1.7469/79   * NLG    1.7597   1.7440
1831 ITL MGTX MORGAN   N.Y 1173.50/4.00 * ITL   1181.00  1171.00
1840 ECU CINY CITIBANK N.Y   1.3213/23   * ECU    1.3231   1.3120
----------------------------------------------------------------
XAU MAQB 343.60/344.10 * ED3  3.81/ 3.93 * FED   PREB * MMKU 30Y
XAG NMRS    4.03/ 4.05 * US30Y YTM  7.83 * 3.75- 3.87 *101.31-00
```

```
RE    NEWS
-------------
MULRONEY REC
-------------
CORRUPTION PROBE STOPS BRAZIL IN ITS TRACKS                 NEWU
-----------------------------------------------------------------
US INDUSTRY COULD WEATHER RAIL SHUTDOWN =5                  NEWV
-----------------------------------------------------------------
US INDUSTRY COULD WEATHER RAIL SHUTDOWN =6                  NEWW
-----------------------------------------------------------------
++ REUTER NEWS HEADLINES -SEE NEW1+++CIS AND EAST EUROPE -EAST++
          +++++++++++SPORT - SEE SPOA ++++++++++
24-JUN-1828. EQU518 N24113452
```

Figure 12.1 *Display presenting some of the visualization tools available for the display of real-time information. Windows 1 and 2 display real-time information about DEC and IBM stocks in the Zurich stock exchange. Windows 3 and 4 provide transaction information about foreign exchange rates. Window 5 and 6 display index values (French Cac 40 and Dow Jones Industrial). Finally, window 7 displays information in page format, and window 8 offers news highlights.*

nents. Although gluons essentially encapsulate communication protocols, they play a prominent role at the design level by promoting a protocol-centered design.

This chapter is organized as follows: the next section provides an overview of how different frameworks address the issue of object cooperation and the patterns of cooperation that they promote. We focus on standardization proposals promoted by major software houses since they will most likely have a significant impact on the future architecture of software applications. Section 12.3 discusses the requirements for the financial framework. Such requirements cannot be easily satisfied with the previously described approaches and we therefore introduce a new protocol-centered approach. Section 12.4 discusses gluons as special components that enable a protocol-centered approach. Section 12.5 presents the financial framework, focusing on the illustration of commonly used gluons. We conclude with a summary of the advantages of protocol-centered frameworks.

12.2 An Overview of Cooperation Patterns

The development of mechanisms that support communication between software components is hardly a new problem. A significant effort has been devoted in the past, for instance, to interapplication communication. A typical mechanism is the remote procedure call (RPC), which allows an application to invoke routines belonging to another application. RPC is the kind of cooperation mechanism one expects in software environments where the principal entities are functions and data structures. In a word of objects, however, we might expect to have remote message capabilities since the message is the inter-object communication mechanism.

To the best of our knowledge the first commercially available implementation of remote messages came bundled with NeXTStep AppKit framework[19]. However, remote messaging only provides a communication layer. For software components to cooperate in a dependable and flexible way we need to define the laws of cooperation. In what follows we provide an overview of various standardization efforts that address, in a broad sense, the problem of defining laws of cooperation in the context of software development frameworks.

12.2.1 Object Management Group

The Object Management Group (OMG) promotes a standard to support the interaction of software components within a framework called the Object Management Architecture (OMA). One of the main goals of OMA is to achieve object distribution transparency, which means that the interaction between a client component and a server component through the server's interface should be independent of its physical location, access path, and should be relocation invariant. This standard relies on a common object model, the OMG Object Model which is used by all OMG-compliant technologies.

12.2.1.1 The OMG Object Model

The OMG Object Model defines a way to specify externally visible characteristics of objects in an implementation-independent way. The visible characteristics of an object are described as a collection of operation signatures called the object's interface. The OMG Object Model definition of an operation signature extends in interesting ways the typical definition of a method's signature in order to make it more convenient for distributed computing environments. The optional **oneway** keyword specifies an exactly-once operation semantics if the operation successfully returns results or a at-most-once semantics if an exception is returned. Each parameter is flagged with one of the three qualifiers — **in**, **out** or **inout** — to specify the write access to the parameter of the client, the server or both. An exception is an indication that the request was not performed successfully. The **raises** keyword introduces the list of possible exceptions that can be raised by the operation. Finally,

```
[oneway] <return_type> <operation>(in|out|inout param1, ..., in|out|inout paramK)
        [raises (except1, ..., exceptL)]
        [context (name1, ..., nameM)]
```

Figure 12.2 *The OMG Object Model operation signature.*

the **context** keyword allows for the specification of additional information that may affect the performance of the operation. These extensions address issues related to distributed environments such as unreliable communications, and the need for appropriate mechanisms for exception handling.

12.2.1.2 Object Request Broker

The communication between objects is mediated by an Object Request Broker (ORB). The ORB is responsible for finding the object implementation for the requested operation, to perform any preprocessing needed to perform an operation, and to communicate any data associated with the operation. The functionality of object request brokers is defined in the Common Object Request Broker Architecture (CORBA)[21]. In order to ensure language independence, CORBA defines a Interface Definition Language (IDL) that obeys the same lexical rules as C++, although additional keywords are introduced essentially to support distributed environments. However, IDL differs from C++ in that it is only a declarative language. In order for object implementations to communicate with the ORB they need to implement a Basic Object Adaptor (BOA) which deals with such aspects as interface registration, implementation activation, and authentication and access control. An important component of the ORB is the interface repository which provides access to a collection of object interfaces specified in IDL.

To summarize, the OMG provides a standard for the communication of objects in distributed environments. The standard focuses on interoperability of heterogeneous systems, where interoperability is achieved through a request broker that defines standard interface rules which the interacting agents need to obey.

12.2.2 Microsoft DDE and OLE

Microsoft provides two mainstandards for interapplication cooperation: DDE (Dynamic Data Exchange) and OLE (Object Linking and Embedding). DDE is much simpler than OLE since it addresses essentially the exchange of data between applications that run on the same computer. On the other hand, OLE is an ambitious standard that encompasses many aspects related to the structures of software components.

12.2.2.1 Dynamic Data Exchange

DDE focuses on data exchange between applications based on a client–server model. In DDE parlance, a client is any application that initiates a DDE connection. Usually a client requests data after establishing a connection with a server. The connection establishes a

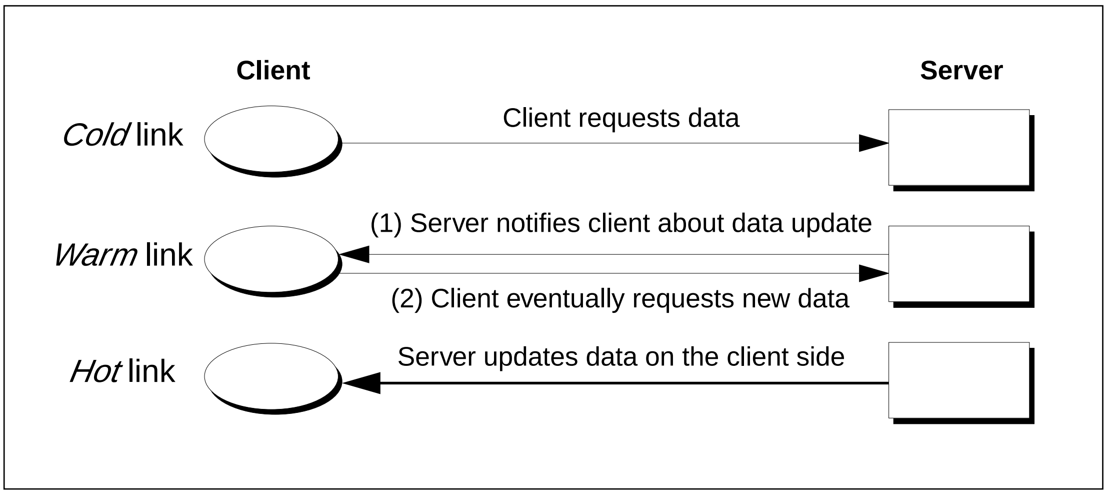

Client
Server
Cold link
Client requests data
Warm link
(1) Server notifies client about data update
(2) Client eventually requests new data
Hot link
Server updates data on the client side

Figure 12.3 *DDE involves three types of links between clients and servers. The variety of links reflects the different requirements of applications on how to maintain client's data consistent with the corresponding server's data.*

link that according to the way the link deals with data updates on the server side can be one of three types: *cold*, *warm* and *hot*. These three links are illustrated in figure 12.3. With *cold* links the server plays a passive role: it takes no action whenever data is updated. The client is, therefore, responsible for implementing the update policy by issuing data requests when appropriate. With *warm* links the responsibility for data update is shared between the client and the server: the server notifies the client upon a data update but the data request to perform the update on the client's side is initiated by the client. Finally, with *hot* links the server is responsible for the whole update process on the client's side.

The three types of links allow for the implementation of data consistency policies between the client and the server that appropriately reflect the requirements of the client application. The actions on both the client and the server side are carried out through call-back functions.

The data organization at the server end follows a three-level hierarchy that recognizes three entity types: services, topics and items, as illustrated in figure 12.4. Typically, a topic corresponds to a document (e.g. an open document in a wordprocessor server) but it can also represent a relation in a relational database since the DDE standard does not specify what a topic should be. Items are the smallest entities that can be addressed through DDE. Items can be of any type and format recognized by the Windows clipboard. In order for a client to request data from a server it needs to know the name of the service provided by the server, the name of the topic and the name of the item it is looking for. A client can connect to multiple servers and a server can be linked to multiple clients. Although DDE is essentially a mechanism for data exchange among applications it also provides limited capabilities that allow a client to execute commands on the server side. These capabilities can be used to implement cooperation mechanisms that are, to some extent, similar to remote messaging in other environments.

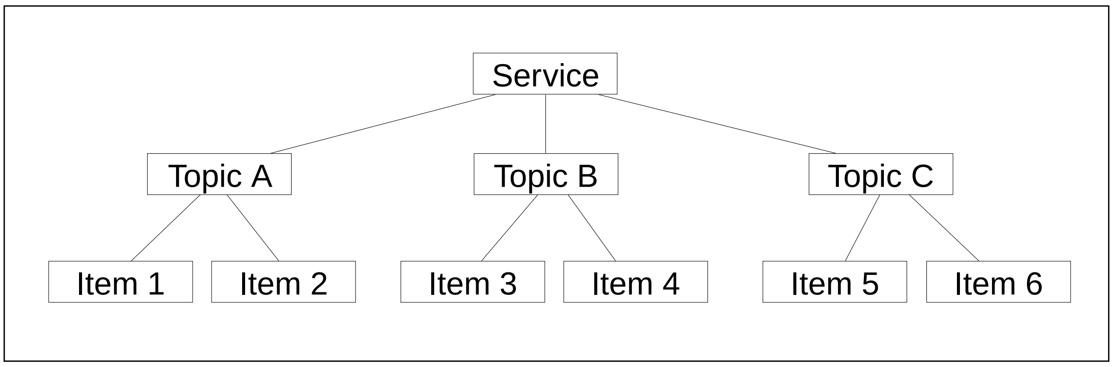


Figure 12.4 *DDE hierarchy showing the service provided by a server and how it is hierarchically organized in topics and items.*

12.2.2.2 OLE 2.0

OLE is another standard defined by Microsoft that enables the cooperation of applications. In its current 2.0 version [17][18] it shares many similarities with OpenDoc that we will describe in section 12.2.4.2. For instance, both standards comprise a set of cooperation protocols and a definition for compliant structured documents. OLE 2.0 is relatively hard to summarize briefly. In fact OLE 2.0 is much more than a application cooperation standard; it is the foundation for a Microsoft strategy to make MS-Windows migrate to object-oriented technology. As such, OLE 2.0 comprises a set of apparently loosely related standard definitions, models and implementations which provide, as a whole, a coordinated platform for future object-technology. OLE 2.0 provides standard definitions and implementation support for compound documents, drag-and-drop operations, name services, linking and embedding of documents, and application interaction automation.

The unifying concept underlying the OLE 2.0 platform is the Component Object Model (COM). All the other pieces of OLE 2.0 either rely on the COM definitions or use COM objects, usually called Windows objects [17]. Windows objects differ slightly from the objects proposed by commonly used programming languages such as C++ or Eiffel. A Windows object is fully defined by its set of *interfaces*. An *interface* is a collection of function pointers and there is no such notion as references to Windows objects. When we obtain a reference to an object it is in fact a reference to one of its *interfaces*. Another interesting aspect of Windows objects is that there is no inheritance mechanism, but because Windows objects provide multiple *interfaces*, it is easy to encapsulate Windows objects with programming languages that offer either single or multiple inheritance. The COM presents Windows objects essentially as collections of functions [7][17] (i.e. *interfaces*), which can be fairly confusing for readers acquainted with object-oriented concepts. The main reason, we believe, is that the OLE 2.0 is to be implemented with many different programming languages, such as BASIC, C, C++, which may or may not endorse object-oriented techniques. With different programming languages the binding between the object's data and the object's methods may be implemented in different ways that are not specified in OLE. Microsoft offers an OLE 2.0 software development kit for C++ environments.

A key feature of OLE 2.0 is the definition of structured documents. Structured documents contain *storages* and *streams* that are organized in a similar way to traditional file systems: *streams* are analogous to files while *storages* act as directories. So, *storages* contain either *streams* or *storages*. Storages and streams provide support for structured or composite documents that are organized in a hierarchical structure. OLE 2.0 provides a standard definition for the document's structure and also a set of functions that support the standard operations on structured documents.

The best-known features of OLE 2.0 are probably embedding and linking. A typical compound document (e.g. a text with graphics, sound, data in spreadsheet format, etc.) contains data objects that have been created by different applications. The owner of the compound document, say a wordprocessor, may know how to display most of these items but cannot deal with the full complexity of retrieving and modifying them. An OLE *container* is any application that can incorporate OLE objects. *Containers* usually display the OLE objects and accept commands for them. However, *containers* are not intended to process the objects. Objects retain an association with *server* applications that are responsible for servicing the requests addressed to the objects. The idea here is that clients do not need to be aware of the internals of the objects they contain. The object (data) together with its associate server corresponds to the usual notion of object in object-oriented terminology which encapsulates both data and operations on the data. *Servers* accept commands, called *verbs*, that correspond to actions that can be applied to the objects. An interface is the set of operations that can be applied to an object via its server.

OLE 2.0 offers two ways to integrate an object into a compound document: *linking* and *embedding*. Embedding is most frequently used. The *container* application owns and stores each embedded object, but the server retrieves the object. The server plays an anonymous role by processing the object on behalf of the container application. Conversely, an object can be linked into a document. A linked document belongs to a given document (and is stored in the document's file) but it is referenced in another document. In this way several *containers* can share a single linked object.

Additionally, OLE 2.0 provides a standard for data transfer called Uniform Data Transfer (UDT) and a standard for scripting called Automation. Automation allows objects associated with one application to be directed from another application, or to perform operations on a set of objects under the control of a macro language [18].

To summarize the OLE 2.0 standard suite we may say that the Component Object Model standardizes how an object and an object's client communicate; compound documents standardize document structure and storage; Uniform Data Transfer standardizes data exchange capabilities and Automation provides a support for remote control of applications.

It should be noted that with OLE version 2.0 the interapplication cooperation primitives are restricted to the scope of the same machine. However, these mechanisms could easily be extended to provide the same capabilities across networks and serve, therefore, as a foundation for distributed computing.

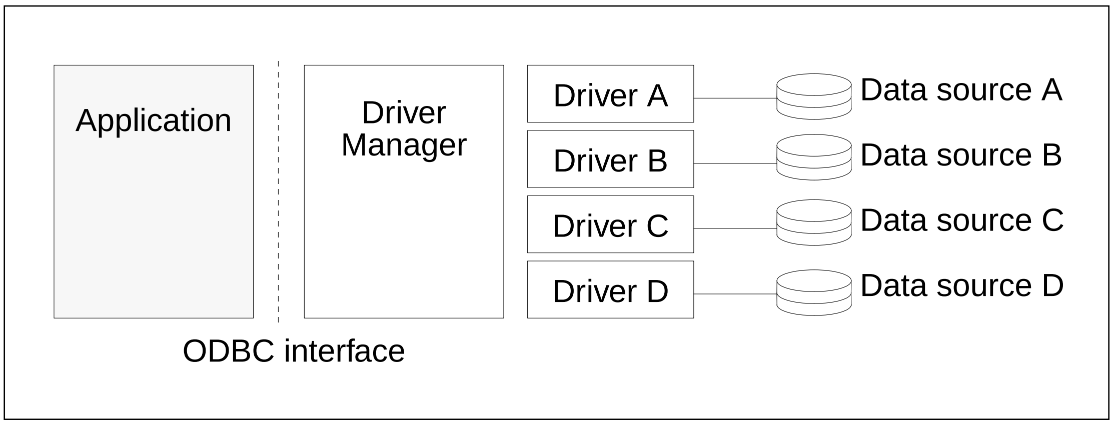


Figure 12.5 *ODBC 2.0 application architecture.*

12.2.3 ODBC 2.0

Although the Open Database Connectivity standard from Microsoft is more a standard for the interconnection of applications and databases, it is worth mentioning here for two reasons. First, it represents a much-needed standardization effort to isolate applications from the access to specific databases. Second, databases will be, at least in the near future, one of the most prominent reusable software components since they are responsible for object persistence.

The architecture of an ODBC 2.0 application is represented in figure 12.5. From the view point of the application, the access to the various data sources is transparent through the ODBC interface. The ODBC 2.0 standard interface provides the following:

- a standard way to connect to databases;
- a set of function calls that allows an application to connect to one or many databases, execute SQL statements, and retrieve the results;
- a standard representation for data types.

The Driver Manager loads drivers on behalf of the application, while the Drivers implement ODBC function calls and submit, when appropriate, requests to the associated data source. The Drivers are responsible for adapting to the specific syntax of the associated DBMS. ODBC 2.0 does not rely on object-oriented principles and is fairly low level in the sense that it provides a vendor-independent mechanism to execute SQL statements on host databases.

12.2.4 Apple's Interapplication Communication Architecture and OpenDoc

Like Microsoft, Apple devoted significant efforts to the definition and implementation of standard mechanism for the cooperation of applications. Also like Microsoft, Apple has a

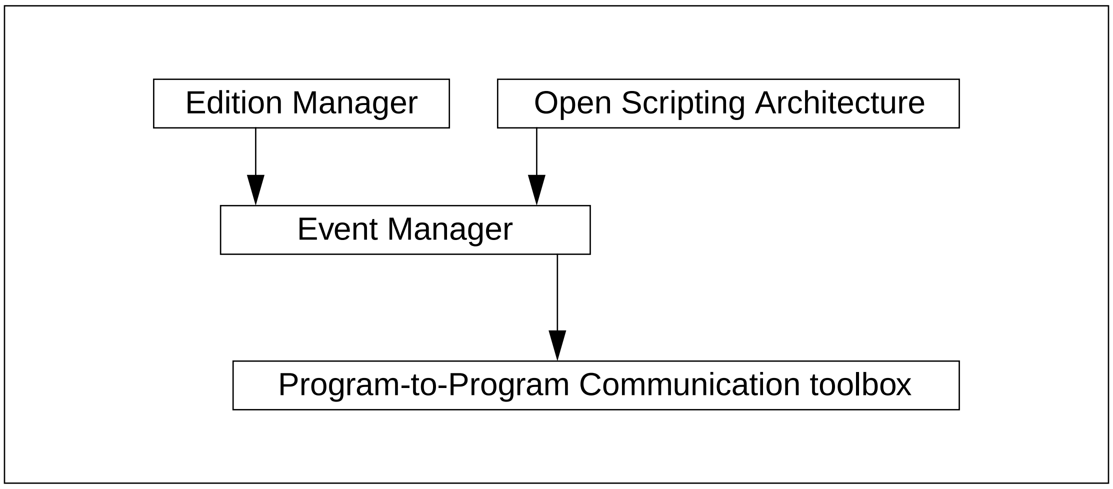

Edition Manager
Open Scripting Architecture
Event Manager
Program-to-Program Communication toolbox

Figure 12.6 *The layers of the Interapplication Communication Architecture.*

large developer base and a large software base that did not already fully adopt object-oriented tools. The consequence is that the migration towards an object-oriented platform started by the introduction of object-oriented concepts such as message passing into development environments that are not object-oriented.

12.2.4.1 Interapplication Communication Architecture

This migration was the driving force for the development of the Interapplication Communication architecture (ICA), which provides a standard mechanism for communication among Macintosh applications[1]. More specifically the goal is to allow applications to:

- exchange data through copy-and-paste operations;
- read and write data blocks from and to other applications;
- send and respond to Apple events;
- be controlled through scripts.

A significant effort has been devoted by Apple to define a common vocabulary of high-level messages, called Apple events, that are published in the Apple Event Registry: Standard Suites. To the best of our knowledge, this has been the only effort to date to standardize the messages that applications may respond to.

- Applications typically use Apple events to request services from other applications or to provide services in response to other applications requests. A *client application* is an application that sends an Apple event to request a service, while the application that provides the service is the *server application*. The client and server applications can reside on the same machine, or on different machines connected to the same network.

The ICA comprises the following:

- The *Edition Manager*, which provides support for copy-and-paste operations among applications and updating information automatically when data in the source document changes.
- The *Open Scripting Architecture*, which defines the standard mechanisms that allow for the external control of single or multiple applications. OSA is comparable, to some extent, to Automation in OLE 2.0. OSA is not tied to any specific scripting language. Each scripting language has a corresponding scripting component that translates the scripts into events.
- The *Event Manager*, which provides the support that allows applications to send and receive events. The Event Manager standard defines the architecture and the pieces of Apple messaging backplane.
- The *Program-to-Program Communication toolbox*, which provides low-level support that allows applications to exchange blocks of data in an efficient way. The Edition Manager and the Open Scripting Architecture provide the user level support. They both rely on the Event Manager to exchange data and messages across applications. The Event Manager, in turn, relies on the Program-to-Program Communication toolbox to transport data. Figure 12.6 illustrates how the different parts of the ICA are related.

12.2.4.2 OpenDoc

As opposed to OLE, the ICA only deals with the problem of application interaction and does not define a standard for documents. Apple, together with other companies such as Novell and IBM, is proposing another standard, OpenDoc, that is quite similar in scope to OLE 2.0. It defines both standards for application interaction mechanisms and for structured documents. In reality, OpenDoc integrates three other standards: (1) System Object Model (SOM), which originated as a CORBA compliant IBM standard for interapplication message exchange; (2) BENTO, which standardizes the format of structured documents and (3) the Open Scripting Architecture that we already mentioned as part of Apple’s ICA.

BENTO defines the standard elements for structuring documents in OpenDoc. BENTO documents are stored in containers which are collections of objects. BENTO objects are organized as schematized in figure 12.7. An object has a persistent ID which is unique within its container. Objects contain a set of properties, which in turn contain values of some type. The values are where data is actually stored and their types describe the corresponding formats.

The ideas underlying OpenDoc are quite similar to those on which OLE 2.0 is based: composite documents may contain heterogeneous objects that are managed and manipulated using a variety of specialized software components. With OLE 2.0 the specialized components are heavyweight applications such as wordprocessors and spreadsheets. On the other hand, OpenDoc targets components that are more fine-grained. The goal is to make the concept of application vanish, giving place to a document-centered approach that promotes the document as the main user concept. Each part of a document retains an association with a specialized component that knows how to retrieve it. Naturally, the in-

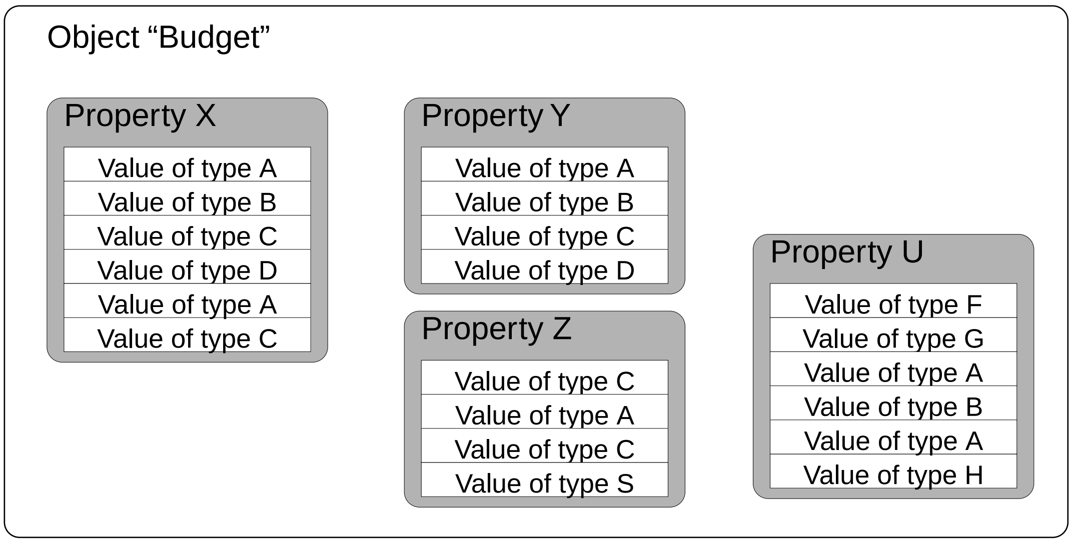


Figure 12.7 *A Bento object contains a collection of properties and properties contain values which are the placeholders where data is actually stored.*

vocation of the retrieving component is transparent to the user, who can easily increase the variety of the parts that can be incorporated into composite documents by purchasing new specialized software components.

12.2.5 Discussion

The considerable effort that has been devoted to designing, implementing and promoting the adoption of these cooperation standards suggests the critical role that such standards may play in future software technology. We may notice, however, that the various standards differ considerably in scope.

For example, OMG standards focus on interoperability among heterogeneous subsystems and they essentially provide mechanisms that allow software components to request services from other software components. Software components need to provide a standard layer that adapts them to the request broker in much the same way that ODBC 2.0 applications need drivers to adapt data sources to the ODBC 2.0 interface. Conversely, OLE 2.0 and OpenDoc each provide a complete integration platform-centered on a standard definition of composite document. The document-centered approach that underlies both standards seems appropriate for office information systems where the composite document seems to be indeed the fundamental user abstraction.

However, there exist many software application domains that do not revolve around the notion of document. For example, in real-time software and communications software, the notion of document does not play an important role. We may also notice that these standards do not promote interaction at the software component level, but rather at the application level, even though OpenDoc encourages document retrieval through a set of small and specialized retrieval units while OLE 2.0 promotes communication among full-fledged applications such as wordprocessors, spreadsheets, etc.

The ICA from Apple (in particular, the Apple events suite) takes a rather different approach, focusing on the standardization of operations. The goal is to promote a standard vocabulary for services so that applications that provide similar services (e.g. spreadsheets) can be replaced by one another.

Another observation is that any of the standards discussed requires mechanisms that are specific to object-oriented languages such as inheritance and encapsulation. In fact, they are being used as a vehicle for the migration towards object-oriented environments by introducing object-oriented concepts expressed in non-object-oriented languages. This is probably the reason why these standards focus mainly on interaction between applications; the same interaction rules do not usually apply to interaction of software components occupying the same address space.

12.3 Requirements for a Financial Framework

The application cooperation standards we have discussed address the needs of a generic software environment and reflect many other constraints not all related to software engineering, such as market constraints and applicability of standards to old development environments. Our financial framework targets applications that retrieve real-time and historical data from financial information sources. Typically, these applications display data such as the price of securities, interest rates and currency exchange rates, and allow users to explore real-time and financial historical information. These applications present to the professional user a window into financial activities which provides access to the distributed world-wide financial market.

Financial markets are characterized by rapidly evolving, complex relationships among the wide variety of financial instruments. Market relationships that hold among financial instruments are continuously evolving, and professional investors are constantly tracking that evolution in order to detect new investment opportunities. Decision support systems (DSSs) play an important role in supporting the user while finding such investment opportunities. The user needs to combine financial instruments, test the combination with various economic scenarios, look at the present cost of the combination, refine the choice of instruments, re-evaluate them, and eventually make an investment decision. In order to provide the appropriate support the DSS should allow the dynamic combination of financial instruments so that any instrument can be combined with any other instrument. This asks for a DSS architecture that facilitates the run-time interaction of software components. Furthermore, new financial instruments are frequently added so the DSS should be easily extendable with operational models for new instruments. To summarize, the architecture needs to provide capabilities for the dynamic connection of software components and facilitate the integration of new software components.

12.3.1 Towards a Protocol-Centered Framework

As we already mentioned, the goal of a framework is to provide a set of classes that are designed to work together. This operational compatibility can be achieved in many ways. The Object Management Group focuses on compatibility mediated by an object request broker. They impose no restriction on the software components themselves. Their main concern is to provide interoperability in heterogeneous environments. OLE 2.0 and OpenDoc emphasize the compound document as the main shared entity. Their main concern is to provide the most flexible environment for document retrieval. Apple's ICA approach, on the other hand, attempts to standardize common operations by defining a standard operations vocabulary and its associated semantics. ICA pursues two main goals. The first goal is to make the access to core standards functionality, such as common spreadsheet operations, database access and wordprocessor tasks, application independent. The second goal is to offer powerful scripting capabilities to automate tasks and to compose applications together.

The goal of the financial framework is to provide support for dynamic control of the interaction between software components. To achieve such a goal we need to provide a mechanism that allows for dynamic interconnection of software components.

12.3.2 Standardizing a Service's Vocabulary

During the early stages of the framework's design we considered a number of alternative intercomponent interaction principles. The goal was to find a mechanism that could provide the highest degree of dynamic interconnection for the kind of applications we are targeting with the financial framework. We tried, for instance, to standardize a set of core services so that each service is associated to a unique name called a verb, much in the same way as the Apple events suite standardizes the operations vocabulary of common services provided by wordprocessors, spreadsheets, databases, etc.

12.3.2.1 The Advantages

The intuition behind this approach is that we can identify among the services provided by the various software components of a framework many services that, although not identical, have comparable semantics. For example, most components in our environment provide services such as *evaluate*, *print*, and *notify*. We attempted to identify within the scope of the financial framework the principal groups of services and we ended up with the list shown in table 12.1. Software components may provide other services as well. These services belong either to more specialized groups, such as a group that is related to real-time services, or they do not belong to any group since they are too specific to a particular class of components.

A major advantage of this approach is simplicity. A service request can be performed by sending a message, a mechanism that every object-oriented environment offers. The advantage of standardizing a vocabulary for services is perhaps more compelling for many

Service group name	Description
Common services	Services that are usually provided by most components such as: *print*, *show-services*, *identify-error,* and *store*
Messaging and notification	Services related to messaging and event notification such as: *call-back*, *notify*, *add-to-broadcast-list, message, forward-message*
Computational	Services related to computational servers: *evaluate*, *iterate*, *perform-aggregation*, *set-value*, *get-value*
Display	Services related to visual operations such as: *display*, *undisplay*, *front*, *drag-and-drop*
Object management	Services related to software component management such as: *create, replicate, destroy, add-object, instance-of-class, component-id*

Table 12.1

reasons. First, reusing components is made easier since services with similar or close semantics bear the same name on all the software components, thus simplifying the name space. Second, dynamic interconnection of software components is improved because if a component provides a service conforming to a standard protocol, such as a print service, then that service can be invoked by any client understanding the same protocol. Third, interchangeability of software components is increased since two software components that provide similar functionality will most probably show a fair degree of commonality in their interfaces.

12.3.2.2 The Shortcomings

We noticed, however, that this approach is not the best in terms of dynamic interconnection. The main reason is that, in general, the interaction between two or more components involves more operations than simply sending a message. Although we can compose software components by specifying the appropriate sequences of messages to be exchanged between the components, a collection of sequences of messages is not the appropriate way to specify components' interactions. All but the simplest interactions involve a state, and the set of permissible messages that can be exchanged between interacting components, at a given point in time, usually depends on the present state of the interaction. Real-time financial environments provide many illustrations. Consider, for example, a software component, called the server, that offers real-time data updating services to other components. A component may register to be notified for data updates. Registering starts an interaction that ends, hopefully, when the client component requests the server to stop notification. Such interaction may comprise many data updates, error messages, notification of temporary interruption of real-time services, with subsequent service resumption, etc. Another example is a database transaction. A server may execute a database transaction on a client component's behalf. The transaction may involve many different opera-

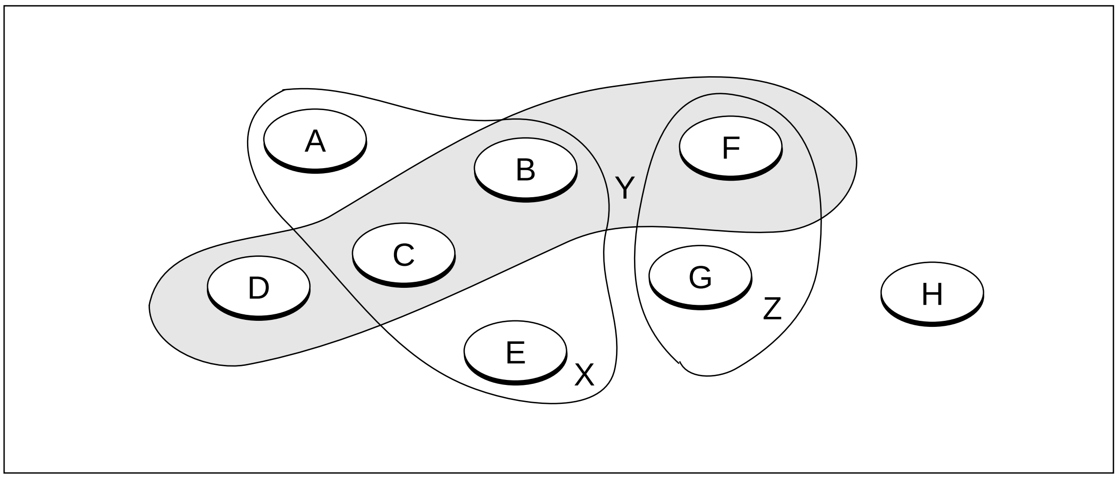


Figure 12.8 *Protocols define compatibility classes inside which members are able to interact. Protocol X allows interaction between objects A, B, C and D, while protocol Y defines an interaction pattern between D, C, B and F. Object H cannot interact with other objects since it does not adhere to any specified protocol.*

tions that individually succeed or fail. The transaction succeeds if all its operations succeed, otherwise the transaction fails. The interaction between the client and the server depends on the state of the transaction, which can be defined as the logical "and" of the individual operation results. Whenever, the state condition switches to fail, the already executed operations need to be unrolled before terminating the transaction.

The two examples illustrate the need for a higher-level mechanism to specify components' interactions that allows for interaction states and state-dependent actions. We call such a mechanism a component's interaction protocol. These observations lay the foundations that lead us from message-based frameworks to protocol-centered frameworks which focus on protocols as the main components interaction mechanisms.

12.3.3 Component Interaction Protocols

Software component protocols share many similarities with computer communication protocols. Both specify object interaction patterns. As such they fulfil two important functions. First, they provide a mechanism or a formalism to specify the rules of interaction between objects. Second, protocols define compatibility classes in the sense that entities that obey the same protocol display an interaction compatibility as illustrated in figure 12.8

12.3.3.1 Requirements for Interaction Protocols

Software component interaction protocols should support a number of important features. First, they should be appropriate to specify various aspects of component interactions such as synchronization, negotiation and data exchange. Second, they should play the role of "contracts" or "interaction agreements" that represent the necessary and sufficient conditions for a software component to interact with other software components that comply with the same "agreement". Helm *et al.* [12] focus on this important aspect of interactions. Third, the interaction specifications should be multilateral agreements rather than bilateral interaction agreements between two software components.

Another desirable property of component interaction protocols is that their implementations reside as much as possible outside the components since as an agreement, a protocol does not belong to any component. We may observe, looking at programs written in an object-oriented language, that a significant fraction of a component's code is devoted to the communication of the component with other software components. Most of the communication functionality is inside the component. This has two main objectionable consequences. First, components tend to become "hard-wired" to their environments, which has the undesirable side effect of reducing their reuse potential within other environments. Second, the intermix of code responsible for interaction with the code that is proper to the component reduces readability and maintainability. Naturally, it might be impossible and perhaps undesirable to strip all the interaction code out of a component. The goal is to leave inside the component only the sufficient interaction functionality that can be used by many different protocols. For example, we will keep inside the component methods to export values, methods to notify events, and methods to send generic messages since they do not implement any interaction among specific components and represent the hooks necessary to build protocols.

12.3.3.2 Roles and Interplay Relations

We will be more precise now about what we mean by a protocol. A protocol specifies the interaction between software components. A *protocol P = (R, I, F)* consists of a set of *roles, R,* an *interplay relation, I,* and a *finite state automaton, F.*

P defines a set of roles:

$$R = \{R_1, R_2, \ldots, R_r\}$$

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 client's role, the server's role, or both depending on the specific responsibility assigned to the components. 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 interplay relation is defined by a set:

$$I = \{I_1, I_2, \ldots, I_i\}, \text{ where } I_k \subseteq R, I_k \neq \emptyset, 1 \leq k \leq i$$

Moreover, if $R = \{r\}$, then $I = \{I_1\} = \{\{r, r\}\}$. 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, $I = \{\{\text{server,client}\}\}$ 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:

$$I = \{\{server, client\}, \{server, server'\}\}$$

Each object of the environment O_i eventually conforms to roles of one or many protocols. Let $roles(O_i)$ 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) $I_k = \{x_s, \ldots, x_t\}$, $x \in R$ of its interplay relation defines a domain of interaction compatibility $D = (P, I_k)$. Domains of interaction compatibility play an important role in our framework since they define which are the components that can potentially interact. 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 interactions between participants in the protocol. (See chapter 4 for a formal treatment of two-party protocols based on finite state processes.) In the following section we will see examples of how the state of a protocol can be specified, and how it controls the interactions between components.

12.4 Gluons

Gluons encapsulate and implement interaction protocols by instantiating an interplay relation for a given protocol. The principle idea underlying gluons is to standardize and encapsulate protocols, rather than just standardizing service names, since interaction protocols should represent one of the primary resources to be reused. Gluons support a protocol-centered reuse strategy. By embedding interaction protocols inside gluons we can use them as agents to implement many different interaction strategies.

Applications that we developed with the financial framework show that with this approach we can achieve the following:

- *A high degree of dynamic interconnection* — The reuse of interaction protocols provides significantly more flexibility to express interaction patterns than the reuse of a naming convention. In particular, we typically need a small set of interaction protocols to express interactions that would require a large quantity of standard service names to achieve the same result. For example, all interactions between two software components that involve a service request followed by an agreement on the data types to be exchanged, and ending with a notification of both components about the result of the operation, can be expressed with just one protocol. Service name standardization would require standard names for each possible service request, and would probably ask for additional code to build the sequence of messages needed to perform the interaction. This point will be better illustrated later with examples of gluons from the financial framework.

- *Easy integration of new software components into an environment* — This stems from the fact that the unique interoperability constraint is that the new component reuses existing interaction protocols that can be instantiated through gluons.

12.4.1 Gluons and Software Design

We already mentioned that in a protocol-centered framework the primary reuse resource is the protocol. The adoption of a protocol-centered approach has a significant impact in software design. While methods such as CRC [5] promote an iterative design procedure that emphasizes identification of the responsibilities and collaboration for each component, in a protocol-centered framework the design team attempts to identify the typical interaction protocols for the specific environment prior to any other design decisions. Once the choice of the basic interaction protocols has been made, we then proceed with the identification of the components' responsibilities and the collaborations needed to fulfil such responsibilities.

At first, we seem to be adding just another layer (i.e. the definition of the reusable interaction protocols) to the design process. However, experience shows that, at least in the case of the financial framework, the addition of such a layer simplifies significantly the whole design process provided the reusable protocols are properly defined. Our first design defined only eight protocols that allowed us to express most of the interactions in a simple system. The reusable interaction protocols represent the "glue" that allow for the connection of software components.

12.4.2 Anatomy of a Gluon

In terms of its internal structure, a gluon is a software component that handles a finite state automaton with output to control the execution of a protocol's interplay relation. It contains a start state and any number of intermediate states. A gluon can provide many end states (i.e. accepting states in finite automation parlance) but for simplicity it is better to have a unique end state. Figure 12.9 shows the symbols that can appear in a gluon's finite state automaton. States and state transitions are the common constituents that can be found in any finite state automaton [8]. A participant's role stores a reference to a software component that is compatible with the role defined by the interplay relation, while a message selector container stores an arbitrary message selector.

A state transition triggers the execution of an action which is composed of operations. A state transition is fired whenever the gluon receives a message.

There are three types of operations that compose an action: *messages sends, object assignments* and *message selector assignments*. A message send is what its name implies: the gluon sends a message to a software component requesting a service. Object assignments allows a gluon to keep a reference to software components. Message selector assignments are similar to object assignment operations, the difference lies in the fact that

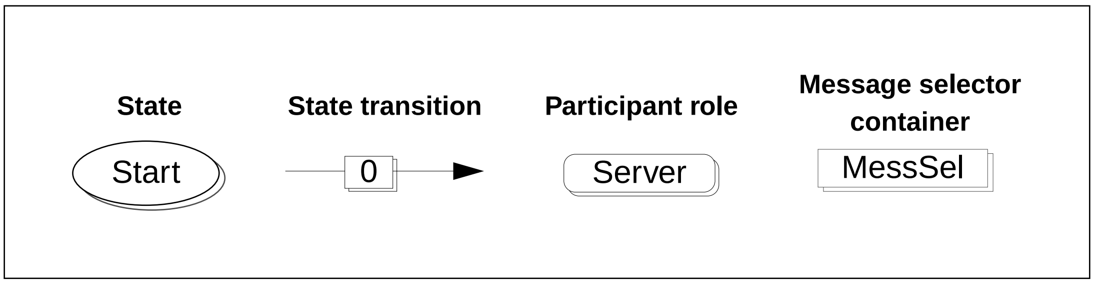


Figure 12.9 *Symbols for the gluon's finite state automaton.*

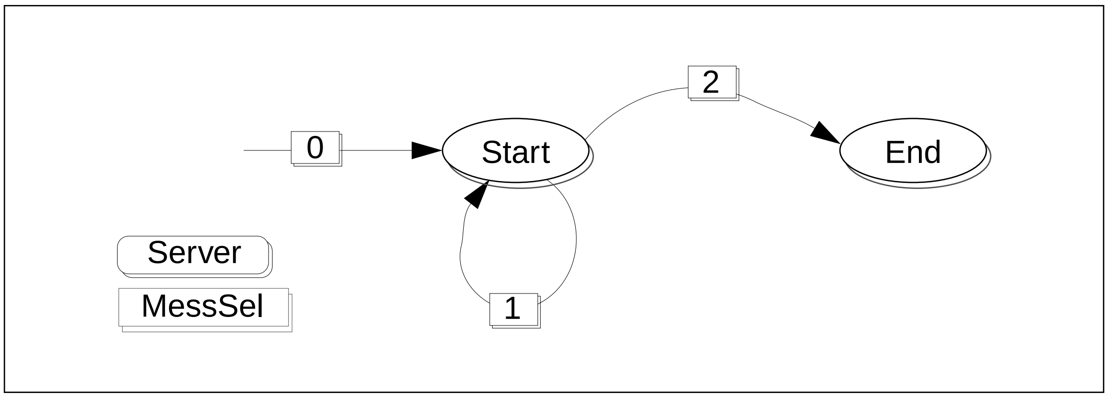


Figure 12.10 *The SimpleGluon finite state automaton. SimpleGluons forward messages to an attached software component called the Server.*

the gluon keeps a reference to a message selector instead of a reference to a software component. These are the only allowable operations in a gluon's action. Furthermore, the only assignments allowed are those that involve either a participant's role or a message selector container in the left side of the assignment.

Figure 12.10 shows the finite automaton embedded inside the simplest gluon provided by the financial framework. The SimpleGluon contains two states, Start and End, and three transitions. The diagram also shows a participant, the Server and a message selector MessSel that can store an arbitrary message selector. State transition triggers and the actions associated with state transitions are shown in table 12.2.

The SimpleGluon handles an asymmetric interaction protocol between a server and a client. The protocol handles message forwarding. The asymmetry stems from the fact that a gluon is associated with a unique server component while the client can be any component that can send a message to the gluon. The association between the server and the gluon is requested by the server component by sending message registerServer to the gluon (refer to table 12.2). This message triggers state transition 0 which initiates the gluon's protocol. Any component can now send messages to the gluon and these messages are forwarded to the server with transition 1. Finally, the gluon can be disconnected from the server by sending it the message exit. SimpleGluons are used in the financial framework

Protocol transitions			Event / action
State	Transition	State	
	0	**Start**	Source: registerServer{server}
			Server := server
Start	**1**	**Start**	<any_obj>: <message>
			MessSel := <message> <message> $\longrightarrow$Server
Start	**2**	**End**	<any_obj>: exit
			gluonDisconnecting{self} $\longrightarrow$ Server *Server :=* none

Table 12.2 *Protocol transition table for the SimpleGluon.*

for two main purposes. The first purpose is to isolate services from service providers. By assigning different components to the server's role, the clients can be granted services from different components. The SimpleGluon plays here the role of a proxy. The second typical usage of SimpleGluons requires a slightly modified gluon with multicasting capabilities. The modified version accepts the registration of multiple servers so that the messages sent by the clients are forwarded to all the servers.

12.5 Gluons and the Financial Framework

Gluons are the architectural elements of the financial framework that are responsible for the way in which other components are composed. The financial framework offers other components as well. One such component, the RealTimeRecord acts as a container for real-time information. This component plays a central role in the distribution of real-time information. The RealTimeRecord plays usually the role of a server to clients request update notifications. The structure of RealTimeRecords is illustrated in figure 12.11. Each entry of the record is a pair *(key, obj_ref)*, where the key allows for the lookup of an object by name.

Most of the components in an application act as data sinks, data sources or both. They are connected through notifications chains so that updates are readily broadcast down the chain. Pure data sources are those components that are either connected to external data source such as those provided by Reuters, or are associated to files providing streams of data. Components that act both as data sinks and data sources are data transformers. They usually get information from data sources, transform it and redistribute it to client components. Pure sink components usually correspond either to display components or to com-

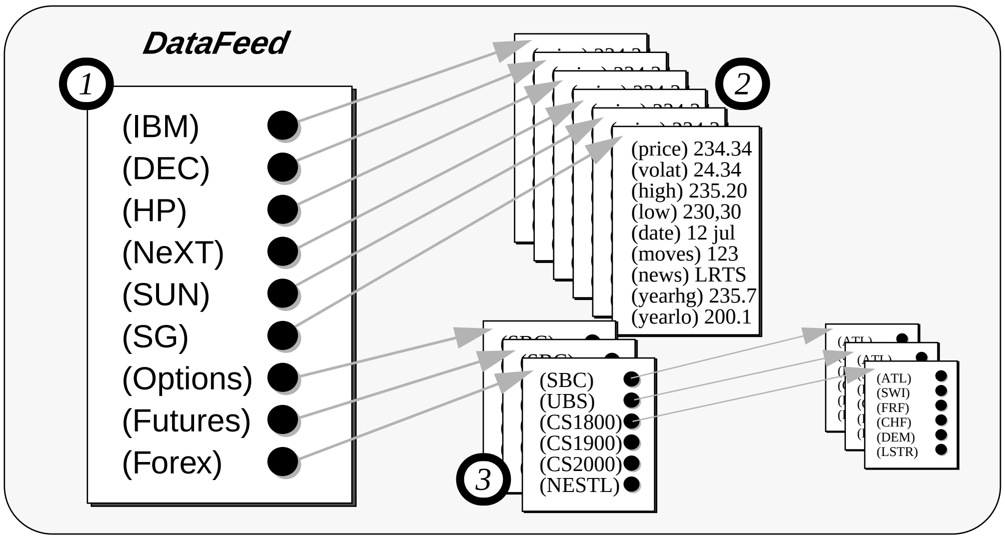


Figure 12.11 *Structure of the RealTimeRecord component. The data is contained in dictionaries. Dictionary 1, for instance, contains references to all the information updated in real time by a data source. The other dictionaries contain either values (2) or references to other objects (3).*

ponents that write to files. So an application can be seen as a set of components connected by a notification web.

The rest of this section illustrates the financial framework by providing two examples of gluons that play an essential role in the framework: the dragging gluon and the real-time data notification gluon.

12.5.1 The Dragging Gluon

The dragging gluon implements the common dragging mechanism we are acquainted with from most windowing systems (see figure 12.12). 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 12.13 illustrates the finite state automaton associated with the dragging gluon, while table 12.3 shows the events that fire each state transition and the associated actions.

To simplify the understanding of how the dragging gluon works it is useful to consult simultaneously figure 12.13, which shows the state transitions, and table 12.3, 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 12.13 represent the roles of the components that participate in the dragging process.

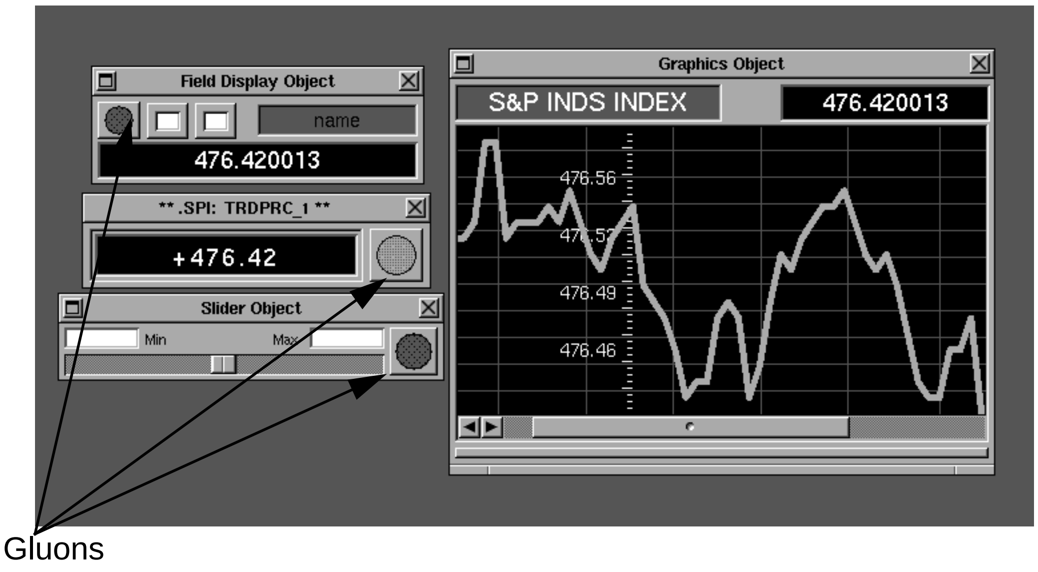


Figure 12.12 *User interfaces of some software components available. The gluons that allow for the connection of the components are indicated by arrows. To connect the components the user drags the circle from one gluon to another.*

The server is the component that initiates the interaction by sending the message startDragging to the gluon with its object identifier as parameter (see table 12.3, transition 0). Upon receipt of this message the gluon enters state Start followed by the execution of an action that makes the gluon 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 destination role is not yet determined. The WindowManager responds to the first the message by sending back to the gluon the dragCandidateEntered message. The reception of this message triggers state transition 1 on the gluon. 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 gluon 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 gluon is

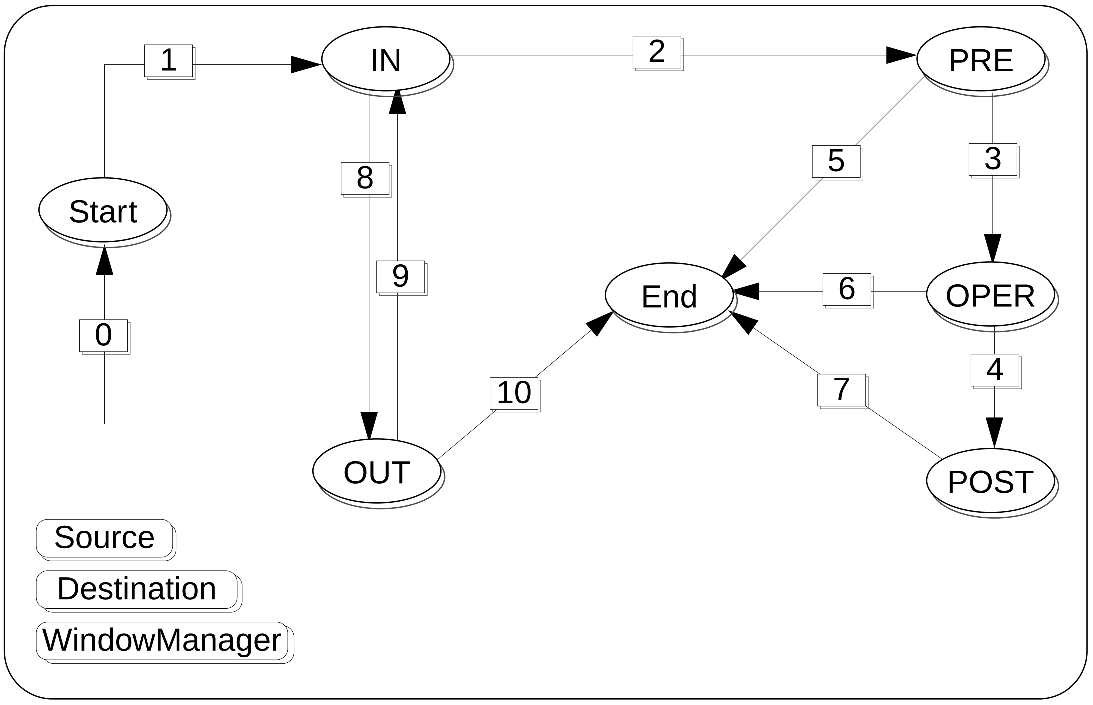


Figure 12.13 *Finite automata for the dragging protocol. The ellipses represent the states while the the arrows represent state transitions. The three boxes at the lower left corner represent the roles of the components that participate in the interaction.*

in the IN state, the gluon undergoes state transition 2 which puts the gluon 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 gluon 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 gluon illustrates the generality and usefulness of an interaction protocol specified as a finite state automaton. Such generic protocols are intended to be refined. Typically, when the source component negotiates an operation with the destination component, they agree on another gluon to which both are compatible. This gluon manages the execution of an operation, or in other terms it mediates the delivery of a service. In the implementation of a visual workbench for the retrieval of real-time financial information, called ReutersLab [25], which has been implemented with the financial framework we extensively use the dragging protocol together with another protocol that negotiates the type

Protocol transitions			Event / action
State	Transition	State	
	0	**Start**	Source: startDragging{Source}
			startDragging{source} ⟶ WindowManager *Source := source* *Destination :=* none
Start	**1**	**In**	WindowManager: dragCandidateEntered{candidate}
			Destination := candidate dragEnter{Source} ⟶ Destination
In	**2**	**Pre**	WindowManager: endDragging
			preOperation{Source} ⟶ Destination
Pre	**3**	**Oper**	Destination: ACK{destination} | Source: ACK{source}
			operation{source} ⟶ Destination
Oper	**4**	**Post**	Destination: ACK{destination} | Source: ACK{source}
			postOperation{Source} ⟶ Destination
Pre	**5**	**End**	Destination: NACK{destination}
			slideDragViewBack ⟶ WindowManager
Oper	**6**	**End**	Destination: NACK{destination}
			slideDragViewBack ⟶ WindowManager
Post	**7**	**End**	Destination: ACK{destination}
			operationComplete{Destination} ⟶ Source
In	**8**	**Out**	WindowManager:dragCandidateExit{candidate}
			dragExited ⟶ Destination *Destination :=* none
Out	**9**	**In**	WindowManager: dragCandidateEntered{candidate}
			Destination := candidate dragEnter{Source} ⟶ Destination
Out	**10**	**End**	WindowManager: endDragging
			dragAborted ⟶ Source slideDragViewBack ⟶ WindowManager *Source :=* none

Table 12.3 *Dragging gluon protocol transition table.*

of data to be exchanged between the source and destination components. Once the components agree on a data type, they interact under the control of another type of gluon that establishes a real-time update notification between the components. The real-time notification gluon is discussed next.

12.5.2 Real-time Data Notification Gluon

Since the financial framework is intended to support the access to information sources that are updated in real time, the framework provides a gluon that supports notification between data sources and client components so that after data updates on the source side the client can be updated to reflect the information change. In a typical situation the client component registers with the source to request update notification. The request creates a link between the source and the client.

In order to provide for flexible notification, the framework allows for three types of notification links — cold, warm and hot — which correspond to the three type of links provided by Microsoft DDE depicted in figure 12.3. The reason for providing three types of notification links stems from the fact that different components have different data update requirements. For example, a client software component that handles a visual display of real-time data usually needs to be updated as soon as the information changes on the source side since the user is expecting the fastest update possible. These requirements correspond to a hot link between the client and the source. Other components expect change notifications but they only need actually to update the values in a few cases. These correspond to the typical requirements for a warm link where the source is in charge of notifying the client while the client is responsible for eventually issuing an update request to the source. The least demanding kind of link is the cold link in which the client is responsible for requesting updates to the source at its own pace with no notification from the source. A typical usage of cold links is portfolio evaluations that require access to market data only when the portfolio is evaluated with no need for further updates.

Figure 12.14 represents the finite automaton embedded in a real-time data notification gluon. The protocol defines three roles: the source, the client and the data. The role of the source and client components has been discussed above, while the component that assumes the data role acts as an information container that is exchanged between the source and the client. The states COLD, HOT and WARM, correspond to three types of links available. When the link is established between the source and the client, the gluon enters the COLD state and waits for a message from the client requesting an update. Upon reception of the client's request the gluon enters state CUP in which it waits until an update message issued by the source puts the gluon back in state COLD through transition 3. A gluon can be requested to switch from one type of link to another provided it is in any one of the three states, COLD, HOT or WARM, so that the update mechanism can be changed at any point in time to adapt to evolving requirements on the client's side. We may notice that state WARM has a self-looping state transition (i.e. number 11), which is fired when the source notifies the client for an update, and two transitions (i.e. transitions 9 and 10) with an intermediary

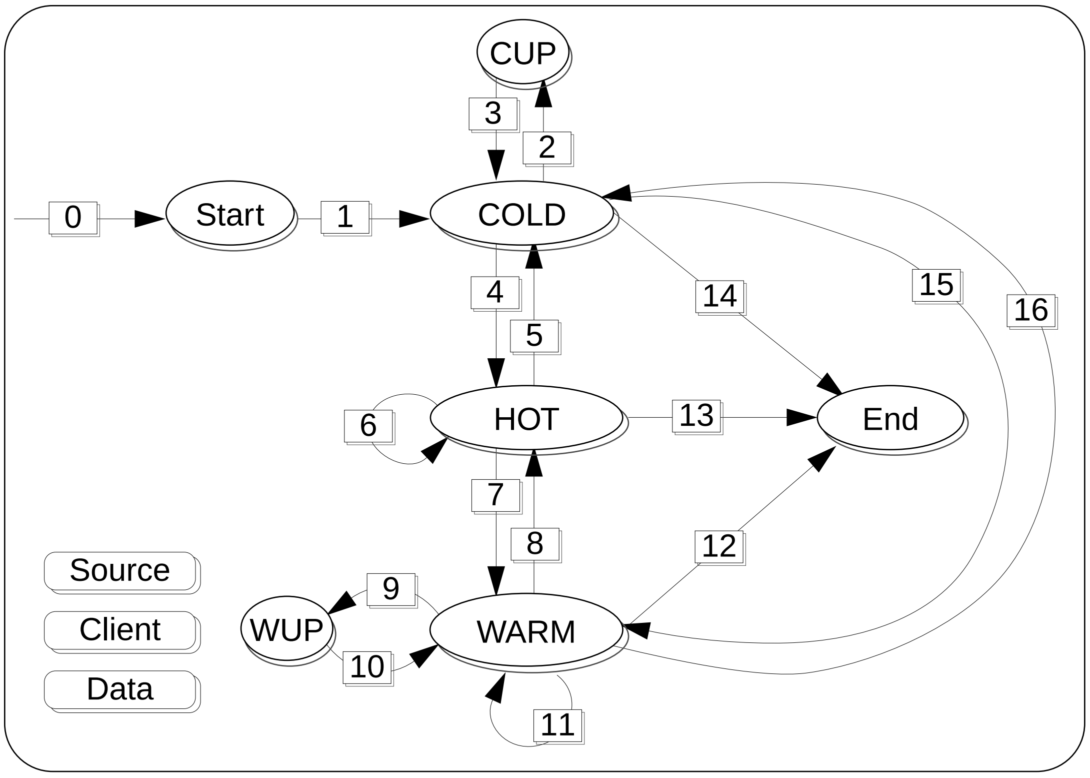


Figure 12.14 *Finite automata for a real-time data notification protocol.*

state WUP which handles the update request from the client component. As expected, the actions associated with transitions 9 and 10 are similar to actions associated with transitions 2 and 3 since they perform the same task.

12.6 Conclusion

We have addressed in this chapter the problem of defining patterns of interaction among software components. We adopt the point of view of component-oriented software design and development which promotes an approach to software construction based on the connection of software components.

We provide a survey of previous efforts that address similar problems. The focus is on work from large software houses since they represent significant efforts to standardize and promote approaches that may have a considerable impact, in the near future, on software design and development. The survey suggest that the sizeable differences that can be observed between such approaches reflect differences in design goals and differences in the requirements of the target environments.

Our development framework targets financial applications that retrieve real-time data and require support that allows for fast reconfiguration of the patterns of interaction among the software components as well as mechanisms that facilitate the introduction of new software components. These requirements can be equated to support for dynamic interconnection of software components. Unfortunately none of the approaches surveyed achieves the desired level of dynamic interconnection capabilities.

We propose a new approach which focuses on the reuse of component interaction protocols. We call a framework based on such principle a protocol-centered framework. Our experience with a financial framework shows that we can achieve a fairly high degree of dynamic interconnection with a small number of reusable protocols (typically less than twenty). However, the applications that we developed have a scope that is too narrow to allow us to infer that the approach is of wide applicability.

References

[1] Apple Computer Inc., *Inside Macintosh: Interapplication Communication*, 1993.

[2] Constantin Arapis, "Specifying Object Interactions," in *Object Composition*, ed. D. Tsichritzis, Centre Universitaire d'Informatique, June 1991.

[3] Constantin Arapis, "Dynamic Evolution of Object Behavior and Object Cooperation," Ph.D. thesis no. 2529, Centre Universitaire d'Informatique, University of Geneva, Switzerland,1992.

[4] Nabajyoti Barkakati, Peter D. Hipson, *Visual C++ Developer's Guide*, Sams, Carmel, 1993.

[5] Kent Beck and Ward Cunningham, "A Laboratory for Teaching Object-Oriented Thinking," *Proceedings of OOPSLA '89, ACM SIGPLAN Notices*, vol. 24, no. 10, Oct. 1989, pp. 1–6.

[6] Ted J. Biggerstaff and Alan J. Perlis, *Software Reusability, Volume I, Concepts and Models*, Frontier Series, ACM Press, 1989.

[7] Kraig Brockschmidt, *Inside OLE 2 : The Fast Track to Buiding Powerful Object-Oriented Applications*, Microsoft Press, Redmond, Wash., 1993.

[8] Daniel I. A. Cohen, *Introduction to Computer Theory*, John Wiley, 1986.

[9] L. Peter Deutsch, "Design Reuse and Frameworks in the Smalltalk-80 System," in *Software Reusability*, ed. T.J. Biggerstaff and A.J. Perlis, ACM Press, 1989, pp. 57–71.

[10] Erich Gamma, Andre Weinand and Rudolf Marty, "Integration of a Programming Environment into ET++," *Proceedings of ECOOP '89*, British Computer Society Workshop Series, Cambridge University Press, Cambridge, 1989.

[11] Simon Gibbs, Dennis Tsichritzis, Eduardo Casais, Oscar Nierstrasz and Xavier Pintado, "Class Management for Software Communities," *Communications of the ACM*, vol. 33, no. 9, Sept. 1990, pp. 90–103.

[12] Richard Helm, Ian Holland and Dipayan Gangopadhyay, "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems," *ACM SIGPLAN Notices*, vol. 25, no. 10, Oct. 1990, pp.169–180.

[13] Dan Ingalls, "Fabrik: A Visual Programming Environment," *Proceedings of OOPSLA '88, ACM SIGPLAN Notices*, vol. 23, no. 11, Nov. 1988, pp. 176–190.

[14] Ralph E. Johnson and Brian Foote, "Designing Reusable Classes," *Journal of Object-Oriented Programming*, vol. 1, no. 2, 1988, pp. 22–35.

[15] Chris Laffra, "Procol, a Concurrent Object Language with Protocols, Delegation, Persistence, and Constraints," Ph.D. Thesis, Amsterdam, 1992.

[16] Michael Mahoney, "Interface Builder and Object-Oriented Design in the NeXTstep Environment," Tutorial Notes of CHI '91, available through anonymous ftp at nova.cc.purdue.edu.

[17] Microsoft Press, *OLE 2 Programmer’s Reference: Working with Windows Objects*, Vol. 1, Redmond, Wash., 1994.

[18] Microsoft Press, *OLE 2 Programmer’s Reference: Creating Programmable Applications with Ole Automation*, Vol. 2, Redmond, Wash., 1994.

[19] NeXT Computer Inc., *NextStep Concepts Manual*, 1990.

[20] Oscar Nierstrasz, Dennis Tsichritzis, Vicki de Mey and Marc Stadelmann, “Objects + Scripts = Applications,” in *Object Composition*, ed. D. Tsichritzis, Centre Universitaire d'Informatique, June 1991, pp. 11–30.

[21] Object Management Group, *Common Object Request Broker: Architecture and Specification*, 1991.

[22] Object Management Group, *Object Management Architecture Guide*, 1992.

[23] Object Management Group (OMG), *The Common Object Request Broker: Architecture and Specification, Object Management Group and X Open*, OMG document 91.12.1, revision 1.1, 1992.

[24] Xavier Pintado, Dennis Tsichritzis, “Gluons: Connecting Software Components,” in *Object Composition*, ed. D. Tsichritzis, Centre Universitaire d'Informatique, 1991, pp. 73–84.

[25] Xavier Pintado, Betty Junod, “Gluons: A Support for Software Component Cooperation,” in *Object Frameworks*, ed. D. Tsichritzis, Centre Universitaire d'Informatique, 1992, pp. 311–330.

[26] Xavier Pintado, “Gluons: a Support for Software Component Cooperation,” in *Proceedings of ISOTAS ’93, International Symposium on Object Technologies for Advanced Software*, ed. S. Nishio and A. Yonezawa, Kanazawa, Japan, November 1993, Springer-Verlag, pp. 43–54.

[27] Xavier Pintado, “Fuzzy Relationships and Affinity Links,” in *Object Composition*, ed. D. Tsichritzis, Centre Universitaire d'Informatique, 1991.

[28] Rajendra Raj, Henry Levy, “A Compositional Model for Software Reuse,” *Proceedings of ECOOP ’89*, British Computer Society Workshop Series, Cambridge University Press, Cambridge, 1989, pp. 3–24.

[29] Jeffrey S. Rosenschein and Gilad Zlotkin, *Rules of encounter : Designing Conventions for Automated Negotiation Among Computers*, MIT Press, Cambridge, Mass., 1994.

[30] Al Williams, *OLE 2.0 and DDE Distilled : A Programmer’s Crash Course*, Addison-Wesley, Reading, Mass., 1994.