Frameworks for Compound Active Documents (Draft)

Peter Wegner, Oct 20 1997
www.cs.brown.edu/people/pw

Abstract:

To provide a framework for component-based technology, we combine the bottom-up analysis of three specific systems with top-down conceptual models of interaction. CORBA/OpenDoc, COM/OLE/ActiveX, and Java/JavaBeans concretely illustrate emerging principles of component and document design, such as the events-properties-methods model. They support components with visual, interactive listening membranes that transform black-box computers into glass-box systems whose "picture windows" allow clients to both see and modify what is inside. Frameworks are viewed as extensible collaborating collections of components with goal-directed behavior. Collaboration is conceptually modeled as a constraint on behavior just as collaboration among partners in a marriage or workers on an assembly line constrains the behavior of people. Interactive structure among objects and components is expressed by patterns, which are notoriously difficult to formalize, in large measure because they specify interactive systems that cannot be formalized by first-order logic. A constraint-based specification paradigm for components is introduced to express the collaborative semantics of frameworks.

1. Documents, Components, and Frameworks

The concepts object, component, framework, document, and compound active document have context-dependent meanings but may, as a first approximation, be defined as follows:

object: container with identity and interface operations that share a persistent state
component: umbrella concept for variable granularity, reusable, possibly off-the-shelf entity
framework: has collaborating components with goal-directed extensible behavior through an API
document: component with visible interactive interfaces for browsing and authoring
compound active document: document with autonomous parts programmable through scripting

Components are an open-ended umbrella concept that include OpenDoc, COM, and JavaBeans components as well as off-the-shelf pluggable component-based software. Frameworks, the primary focus of this paper, are collections of collaborating components with domain-specific extensible client interfaces. Documents are specialized components whose visible interfaces support browsing and authoring, while compound active documents consist of parts with autonomous functionality supported by multiple threads and scripting.

OpenDoc extends the file/folder desktop paradigm of the 1980s to provide a powerful and simple compound document model on top of CORBA's interoperable distributed objects. ActiveX provides a document model for World Wide Web (WWW) documents on top of Microsoft's COM/OLE. JavaBeans provides a Java-based application environment for managing components and building platforms out of beans (classes that conform to certain interface conventions) supplied in Java archive repositories (JARs). Comparison of OpenDoc, ActiveX, and JavaBeans document models yields principles for the design and implementation of frameworks of collaborative components with visible, interactive interfaces.

OpenDoc, COM/OLE, and Java have very different component models. OpenDoc components have identity, a state, and visual interfaces, COM/OLE components consist of collections of interfaces that provide time-independent services and treat identity and state as properties of special state-sensitive interfaces for containers, monikers and data transfer, while Java components have a core interface and security model extended by modular class libraries and tools.

OpenDoc specifies a language-independent infrastructure and document model from the ground up through an industry-wide effort coordinated by OMG. Its conceptual and architectural elegance provide a baseline for understanding document architectures, though it has been cancelled as a product. ActiveX specializes Microsoft's component architecture COM/OLE to a Web-based document model. Java, with extensions specified in the class libraries java.lang, java.util and java.awt, provides not only a language but also an environment for component-based technology, while JavaBeans extends the Java environment to provide a toolkit for component and document composition.

CORBA has played a central role in formulating concepts and architectures of interoperability. COM's interoperability at the binary (machine-language) level and Java's interoperability within a single well-designed language are less ambitious than CORBA's multilanguage approach. Interoperability is much simpler in an integrated language, environment, and component model like Java, since accidental syntactic differences due to language are minimal and questions of semantic interoperability can be addressed directly. JavaBeans focuses on constructing composite beans from Java components, and leaves grafting of top-level application beans (applets) into Web or CORBA systems to higher-level software.

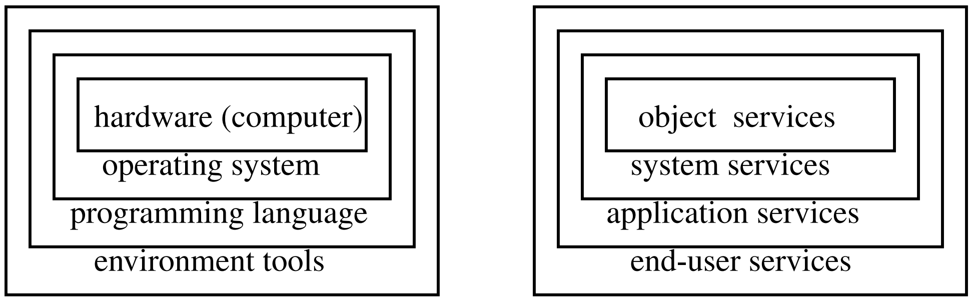
OpenDoc, ActiveX, and JavaBeans differ not only in their substance but also in their descriptive focus. This is reflected in our discussion, which focuses on structure and implementation for OpenDoc, controls for ActiveX, and interfaces and event models for JavaBeans.

2. CORBA Component-Based Software Architecture

CORBA's interoperability architecture aims to extend component reusability at the object level across multiple languages and distributed components. It offers four layers of progressively higher-level services:

CORBA's Layered Service Architecture:

object services: object-level interoperation among multiple languages, interfaces, and platforms
system services: system-level components that provide distributed object services
application services: services used to create applications organized as frameworks
end-user services: domain-specific services provided to end users



a) Layers of Transformation **b) Layers of Interaction**

Figure 1: Two Views of Application Architecture

The traditional "onion" of system layers (Figure 1a) includes an inner hardware layer, an operating-system layer, a programming-language layer, and an environment layer. The CORBA view focuses on interactive services at the object, system, application, and end-user levels. The difference in viewpoint between traditional layers of transformation and layers of interaction is significant from the viewpoint of interactive modeling, discussed in later sections. The four levels of interactive service in Figure 1b are associated with objects, components, frameworks, and user interfaces:

Primitive Architectural Elements:

objects: interoperable object specification in an interface definition language (IDL)
components (classes): system service requirements, CORBA common object services
frameworks: specified by API interfaces for developers, CORBA common facilities
user interfaces: end-user human-computer interface requirements

Since CORBA's object services and IDL have been widely described elsewhere [OHE, We4] and our primary concern is application services provided by frameworks, we briefly examine CORBA's object and system services as a prelude to a more detailed examination of application services.

3. CORBA Object Services

CORBA's object services are realized by an object request broker (ORB) that handles communication among application objects and provides system and library services. Standard interfaces are specified in an interface definition language (IDL) and stored in an interface repository. Interfaces of application objects are mapped to IDL by a language mapping that maps interfaces in specific object-oriented languages into language-independent IDL interfaces.

The ORB accepts requests from client objects through an IDL stub or a dynamic invocation for the services of server objects, transmits them for execution to the server, and returns the result to the client. It also accepts calls to system objects from both client and server objects and object-specific calls through an object adapter, as shown in Figure 2.

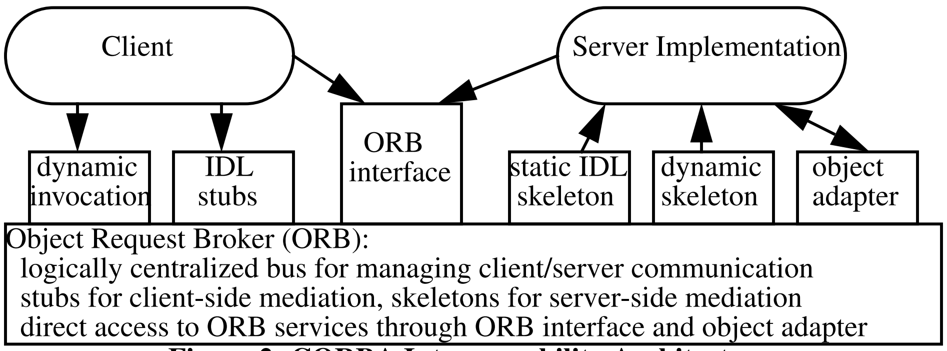

Client
Server Implementation
ORB interface
dynamic invocation
IDL stubs
static IDL skeleton
dynamic skeleton
object adapter
Object Request Broker (ORB):
logically centralized bus for managing client/server communication
stubs for client-side mediation, skeletons for server-side mediation
direct access to ORB services through ORB interface and object adapter

Figure 2: CORBA Interoperability Architecture

CORBA provides mediation services (half-bridges) from clients to the ORB and from the ORB to servers. Clients may invoke a service through a static stub or through a dynamic invocation created from the IDL specification at run time. The ORB validates client requests against the IDL interface and dispatches them to the server where arguments are unpacked (unmarshalled) methods are executed, and results are returned. Server-side software includes object adapters that bind object interfaces and manage object references, and a server skeleton that uses the output of object adapters to map operators to the methods that implement them.

4. CORBA System and Application Services

CORBA's system services (common object services) are described in [OHE], which lists 16 representative operating-system services implemented as classes of a class library:

CORBA System Services:

naming: for managing, querying, and navigating through object name spaces
events: for registering the interest of users in events, push and pull invocation
life cycle: for creating, copying, moving, and deleting compound objects
trader: repository of services available to users, analogous with yellow pages
transactions: for specifying and managing atomic transactions
concurrency control: acquire and release locks to control coordination of shared data
persistence: for managing components that persist beyond lifetime of creator in a database
query: for finding data that satisfies a query specification
collection: queries return collections of data that satisfy the query

relationships: for keeping track of dynamic relationships among components
externalization: for writing the content of components into a stream
properties: repository of properties dynamically associated with objects at run time
time: for synchronizing time in a distributed systems
security: for enforcing security and authentication of components
change management: version control services
licensing: for enforcing licensing restrictions

CORBA's application services (common facilities) are classified into vertical (domain-dependent) and horizontal (domain-independent) services implemented by frameworks. Vertical services support frameworks for applications like health and finance and are developed by service providers like hospitals and banks rather than by computer companies. Horizontal services, expressed as frameworks of collaborating components, include interface, data, task, and system-management services:

CORBA Domain-Independent (Horizontal) Application Services:
interface management: interfaces with a common look and feel, hooks for domain-specific services
information management: document storage and data interchange for compound documents
t**ask management:** managing workflows, long transactions, agents, scripting, rules, task automation
system management: instrumenting, configuring, installing, operating, and repairing components

User interface services are the visible part of an iceberg whose behavior is managed and implemented by hidden information, task, and system management services. Domain-independent services have hooks for domain-specific features that specialize services to a particular domain. OpenDoc specializes these services to documents: it has four subsystems that loosely correspond to the first three services above:

OpenDoc Specialization of application services:
layout: manages document presentation and layout within containers
storage: accommodates arbitrary nesting for documents of varying type and size
data transfer: cut and paste, drag and drop, integrity of linked documents
scripting: supports dynamic execution of programs within active documents

CORBA interface services correspond to OpenDoc's layout, information management to storage and data transfer facilities, and task management to scripting.

5. OpenDoc: A CORBA Framework for Compound Active Documents

OpenDoc is a CORBA-based document framework that evolved from HyperCard's browsing, authoring, and scripting facilities. Document technology was pioneered by Apple Computer in the early 1980s, was present in a mature form in Hypercard in the late 1980s, and has further evolved in the second and third generation systems described here. Our goal is to describe component-based document architectures of the mid-1990s, and to identify underlying architectural models.

Document Structure:

Documents have active parts whose interface behavior is specified by parts editors. They are dynamically created as sessions that persist and evolve over their life cycle.

OpenDoc documents are built up from active parts that know how to draw and print themselves and otherwise control their own destiny [OPG]. Interface behavior is controlled by part editors, implemented by subclasses, that specialize general-purpose editing facilities to the particular part. Part editors define how the part is initialized, drawn, and moved (externalized) and specify events in which the part is interested. Operations for part handling common to many parts are realized by superclasses of a class hierarchy whose part-specific operations are specified in subclasses.

Each part autonomously controls the layout of its subparts, its interaction with other subparts, its animation and mobility, and its lifetime. Parts editors may be dynamically modified or entirely replaced, so that the binding of part editors to parts is looser than that of operations to objects. Parts effectively have multiple editing interfaces that may evolve or be replaced over the lifetime of the part. Loose binding between an object and its interfaces scales up better to large persistent components than tight binding of operations in traditional objects.

Documents are dynamically created as sessions within a document shell environment: sessions of a shell are analogous to instances of classes, but scaling up instance creation to large components requires new technology to handle dependence on support tools. Each session supports a name service for binding and accessing names, a storage service for managing persistent documents, an event service for registering and handling interesting events, a drawing service, and a user interface service that handles document presentation. Documents are heavy components that preserve object-based notions of interfaces and instances but have greater interface-accessing and instance-creation overhead than traditional objects:

Layout

OpenDoc's layout determines the visual rules of engagement that realize the user interface model, storage facilities implement static document structure, data transfer facilities express component mobility, while scripting supports user-specified functionality.

OpenDoc's *layout facilities* specify visual appearance through the use of frames to represent the visible space (real estate) occupied by a part, facets to control the visible behavior of parts, and canvases to describe the drawing environment in which drawing and rendering tools actually render the document part. Frames may have any shape (though they are usually rectangular) and may contain nested frames with autonomous behavior. A given part may appear in many frames with different representations and parts editors. Document parts can be directly selected independent of their level of nesting, and multiple parts of the document can be active at the same time provided there are no shared resource conflicts. Views in all frames are automatically updated when a part's state is changed.

Facets control the geometric relation of a frame to its containing document. Static canvases are used to render parts that do not change, while dynamic canvases are needed to represent video displays or animated objects. Documents can be reconstructed by reading in the frames using facets to determine containment relations and asking facets to draw themselves on canvases.

Storage

OpenDoc's layout functionality is supported by a flexible *storage system* called Bento (after the Japanese compartmentalized food boxes). Bento manages storage for documents by a four-part hierarchy:

storage system: containers -> documents -> parts -> drafts -> storage units
a **storage system** has multiple flexible **containers**
each containing a compound active **document**
consisting of multiple **parts** each having one or more **drafts**
each drafts resides in a flexible **storage unit**

Storage units, the primitive low-level unit for storing document parts, have a complex structure to model the complexity and versatility of document parts, with about fifty methods for managing the properties and contents of documents. Storage units have properties with values that can be arbitrarily large and include references to other storage units. Property-value combinations, called focuses, provide a context for method invocation by pattern matching and permit access to values through a stream interface.

Drafts capture development history and facilitate version control. Documents consist of heterogeneous collections of versions of document parts that are stored in a container. The four-level hierarchy of containers, documents, drafts, and storage units supports very flexible parts at the lowest level, history and version control at the next level, composite documents with varied components at the third level, and a general notion of file-structured containers at the top level.

Data Transfer

The *data transfer facilities* of OpenDoc include drag-and-drop and cut-and-paste, and support automatic updating of links to transferred data. The receiver can decide whether to embed the data as a separate part or to incorporate it within a destination part (this corresponds to the difference between Cons and Append in LISP). If transfers involve a large amount of data, the source can issue a promise that the destination can call in when it is ready to receive the data. Data transfer, which entails the creation of a temporary storage unit, involves a complex sequence of actions.

Though assignment between fixed-size registers of computer memory is simple, scaled-up assignment with variable-size data containing links and other contextual information is complex and shows that scaling up requires qualitative changes in design. memory structure likewise scale up from fixed-size memory registers to flexible multi-level containers. Part structure scales up from objects whose operations are tightly bound to a state to parts loosely associated with multiple parts editors, while the class-instance relation scales up so instances that must fend for themselves after they are created are replaced by sessions whose evolution over time is managed by system services:

Scaling up from languages to components:

assignment -> data transfer of components (parts) with links, editors, and visible properties
memory register -> container with flexible parts, multiple drafts, and visual properties
tightly bound object operations -> loosely bound multiple interfaces that evolve over time
class instances -> document sessions with name, storage, and event services

Scripting

User-defined actions on document parts are specified by *script facilities* that trigger "events" handled by the event model of Open Scripting Architecture (OSA). Scripts that execute on occurrence of an event are registered as events in an event registry.

Scripts provide a mechanism for adding functionality to a document after it has been created. Dialog boxes when documents are opened can enforce security and perform such other tasks as automatically adapting user interfaces to the skill level, language, and goals of the user. Roaming agents that gather information and represent the interests of nonlocal components can be created as active scriptable documents. Scripts can record and subsequently duplicate sequences of goal-directed user actions to learn patterns of frequent higher-level user behavior.

OpenDoc's attempt to extend Hypercard's neat interface and scripting functionality to distributed interoperable components taught us much about document design, though this project has been terminated. Though the above description is terse, it allows comparison of OpenDoc with COM/OLE/ActiveX and Java/JavaBeans.

6. Microsoft's Compound Document Architecture: COM/OLE/ActiveX

The component architecture of COM/OLE focuses on interoperability among nondistributed components conforming to interface and interaction constraints imposed by Microsoft's component object model. ActiveX specializes OLE to HTML-based WWW document technology.

COM supports interoperability of interfaces through function tables at the binary (machine-language) level. It does not directly support higher-level interface definition languages (IDLs) for interoperation between C++, Smalltalk, and Java. Binary interoperability is universal, just as Turing machines are a universal computation mechanism, but requires extra work to support language-sensitive interfaces.

COM's object model differs from that of traditional objects. It defines an object as a collection of interfaces rather than as a collection of operations sharing a common state. All objects have an interface called "Iunknown" that supports multiple interfaces through a query interface function:

QueryInterface: query the set of interfaces to find a specific interface

COM interfaces specify a component's plugs and sockets but do not directly support identity, so that clients are not guaranteed access to the same object on successive occasions of access [OHE]. This model expresses state-independent services but not objects like bank accounts whose services depend on their state. COM expresses objects by special-purpose interfaces like the "dataobject" interface for data transfer and by monikers that provide a powerful naming facility for associating names with storage structures.

Though CORBA and COM differ in their component models, the functionality of each is supported by the other. Tight binding between object identifiers with a state, supported directly in CORBA, is realized in COM by monikers, while loose binding to an interface, the normal accessing mode in COM, is a special case of state-dependent access, and is realized in CORBA/OpenDoc by part editors and other state-independent object services. However, CORBA provides interoperability among distributed objects, while COM/OLE in its 1997 incarnation does not support distributed objects.

OLE stands for Object Linking and Embedding: the distinction between data transfer by copying and linking to a shared copy was central to its design. OLE supports containers that contain compound components with visual interfaces that can be animated by scripts. It captures OpenDoc's document abstraction by a component abstraction that expresses interface behavior and resides in a container that supplies a persistent identity. Both containers and their contents are specified as collections of interfaces that may be accessed through multiple views, corresponding to OpenDoc's accessing of a document part through different part editors. Data transfer through cut-and-paste and drag-and-drop with updating of links is supported. Transferred data can be referenced by a link or can be directly embedded.

COM/OLE specifies primarily logical rather than visual properties of components and relies on languages like Visual Basic or Visual C++ to provide visual functionality. ActiveX extends the functionality of languages like Visual Basic so that the user can create and customize visual interfaces for controlling interaction. ActiveX is a cross-platform, language-independent technology that facilitates the creation, customization, and management of visual "ActiveX controls", either by adding them to a visual programming language or by enhancing Web browsers like Netscape Navigator or Internet Explorer with a collection of customizable primitive controls for common tasks. ActiveX supports creation of controls by customization of generic standard controls of by specification of new controls from scratch.

Controls (called widgets in some systems) are visible interface elements like buttons and dialog boxes that control the behavior of components. They are a visual analog of language control structures that control interactive behavior by user actions rather than controlling algorithmic instruction sequencing. Controls may be activated by user events like depressing a mouse or by system events like storing a value of a property. Microsoft's primary goal for ActiveX is to enhance Internet Explorer, which aims to supplant Netscape Navigator as the browser and document manager of choice for Web documents.

ActiveX controls control objects on Web pages and support interaction among components by listening for the occurrence of events to realize collaborative behavior. They provide document control-functionality comparable to that of OpenDoc parts editors and Java abstract windowing toolkit (AWT), but are an add-on not as well integrated into the underlying document model as comparable features of OpenDoc and Java. However, factoring out interface control as an independent part of document models allows powerful interface control technology to be developed. ActiveX provides a growing library of components for layout, animation, virtual reality, and other forms of enrichment of HTML documents.

ActiveX controls can be implemented in a variety of scripting languages including VBScript for Visual Basic and JScript, which is an open implementation of Javascript. Scripts are directly embedded in HTML code and compiled when the document is read by Internet Explorer. The idea of visual controls whose functionality can be enhanced by scripting, which was already highly developed in the 1980s in Hypercard's buttons, fields, and dialog boxes, is central to compound active document technology.

ActiveX integrates stand-alone applications like Excel into Web documents by enhancing the functionality of Internet Explorer to handle directly the document format of specific applications. Spreadsheets and charts created in Excel can be directly opened through Internet Explorer or any other suitably enhanced browser. Instead of converting documents to HTML and viewing them with an HTML browser, the document browser is modified to accept documents in a variety of native formats.

The COM/OLE/ActiveX model is open in the sense that it can be extended to new document styles and scripting languages, but is proprietary in that it is designed to work with Microsoft-supported software like Internet Explorer. Microsoft's strategy appears designed to make Internet Explorer the browser of choice for Web documents by supporting major competing systems within the COM/OLE component paradigm, while making it hard to run Microsoft systems on CORBA or Java platforms.

7. Java Interfaces, Applets, and Beans

Java has been specifically designed to correct deficiencies of earlier object-oriented languages and support secure, modular component-based technology. It has a clean component model that is extended to visual document interfaces through the class library java.awt and integrated with a well-designed component management system JavaBeans to provide a comprehensive compound active document technology.

Extending Interfaces and Implementing Classes

Java's interface model provides a foundation for its component technology by explicitly distinguishing between state-independent interfaces and state-dependent objects and classes [AG]. Java interfaces may be viewed as an extreme form of abstract classes for which all operations have a deferred implementation:

classes specify both object interface behavior and its implementation
  class inheritance ambiguously mixes behavior and implementation inheritance
abstract classes specify object interface behavior, may implement some methods
  partial implementation that must be completed before instances can be created
Java interfaces support pure behavior specification without any implementation
  interactive composition of behavior is better modeled by interfaces than by classes

The clean distinction between interfaces that specify services and implementations that track evolution over time allows concerns of these two aspects of modeling to be separated. Inheritance is better addressed at the interface level independently of issues of implementation. In particular, multiple inheritance of interfaces avoids problems that arise in multiple inheritance of classes. Java separates behavior extension by inheritance and behavior implementation by classes, as shown in Figure 2. Extended interfaces can be further extended, while extension of classes may create problems because of the conflicting open/closed role of classes as hooks for extension and templates for implementation.

Java permits class extension for compatibility with other object-oriented languages, but forbids it for multiple inheritance. Though Java permits extension of classes, it should be avoided in clean programs so that extension (inheritance) is restricted to interfaces and conceptually independent of class implementation. This accords with the European view that the semantics of inheritance should be defined by specification as opposed to the US (Smalltalk) view that inheritance should be defined by implementation

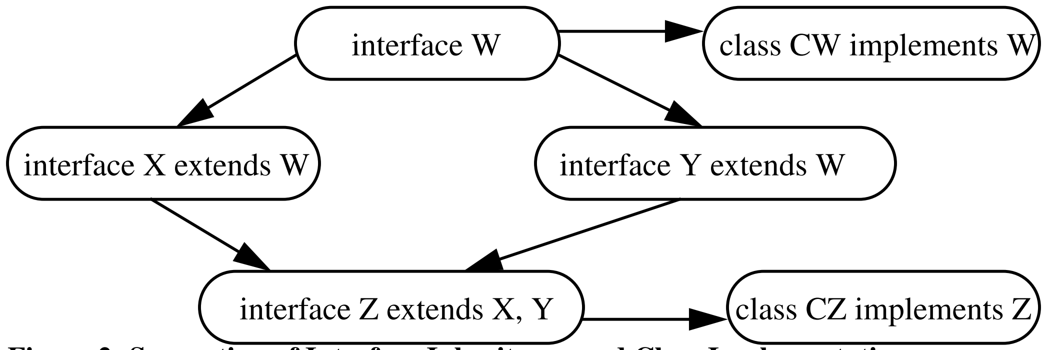


Figure 2: Separation of Interface Inheritance and Class Implementation

Interfaces are transducers (filters) whose composition realizes composite transducers. Interface composition is an object-based analog of instruction composition and the role of interfaces as a primitive for interaction is an analog of instructions as a primitive for algorithms. When interfaces are attached to components with state they lose their purity and interaction among operations of an interface and among multiple interfaces makes their behavior difficult to specify. Extending pure interfaces is weaker in the behavior it can describe than extending classes because it cannot model sharing of state at the subclass level: such behavior can be realized because Java allows extending classes as well as extending interfaces, but should be used only in cases where sharing at the subclass level is a necessary part of the behavior. Interfaces attached to components with state provide persistent services over time whose behavior is described by interaction histories rather than by transformations [We2]. Methods of an interface are listening mechanisms triggered by events. Event models that provide a foundation for the semantics of interfaces are discussed in the next section in exploring the component model of Java Beans.

Applets and JavaBeans

Java supports a flexible client-server model with lightweight servers that dynamically request functionality for specific tasks. Lightweight server technology is exemplified by Hot Java, whose applets can be moved from the server to the client to enhance client functionality dynamically by delegating and distributing server tasks to clients (see Figure 3). The greater flexibility of Hot Java that traditional browsers like Netscape Navigator is due to late (lazy) binding of protocols and modes of viewing on a call-by-need basis. Interactive client/server computing encourages *demand-driven* computing by clients that use mobile services on a call-by-need basis, while traditional client/server computing supports *supply-driven* computing by monolithic server resources.

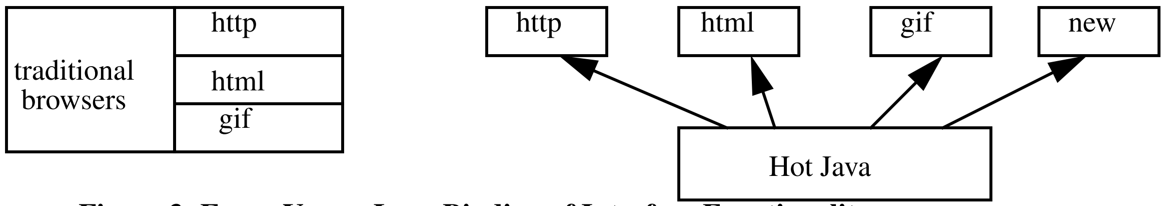


Figure 3: Eager Versus Lazy Binding of Interface Functionality

Applets have environment-independent functionality that supports late (lazy) interactive binding of clients. They are mobile processes whose functionality can be harnessed by browsers independently of the environment in which they execute. Java gains great flexibility by its separation of interfaces and implementation, which yields not only cleaner interface inheritance but also process mobility through environment-independent interfaces and interactive binding of interface functionality.

Applets provide stand-alone functionality for documents (Web pages), but can neither be used by builder tools in constructing composite components nor interact with their containing document. Java Beans facilitates the construction of composite beans from collections of component beans, providing a systematic Java-based realization of the compound active document paradigm.

Java Beans are reusable software components that provide both run-time functionality and design hooks for creating composite structures that can be manipulated in a builder tool [JB]. JavaBeans is a construction environment that requires a builder and some beans (usually supplied in a Java archive resource (JAR) file) used as primitive components in constructing composite beans. Builders are tailored to alternative visual target environments and may provide general Web support like ActiveX, domain-specific layout or server applications, or document editing. The builder uses application beans from the JAR, system beans such as buttons, and introspection and customization provided by the JavaBeans system to build applications. Completed applications, for example applets, may be tested in system-supplied testbeds.

JavaBeans + java.awt + builder + JAR -> new beans (applets) constructed by the JavaBeans construction kit using Java's abstract window toolkit, domain-specific beans in JAR and a builder that understands the target domain

JavaBeans realizes interoperability among components by adaptors interposed between client and server beans. JavaBeans adaptors mediate only semantic needs and can therefore be much simpler than CORBA adaptors whose main task is to mediate between different languages and interface presentation styles. Some forms of adaptation, such as that between buttons and their clients, can be realized by automatically generated adaptors.

8. The Event Model of Component Interaction

Event models allow actions of *event listeners* to be triggered by the occurrence of events at an *event source*. They provide a new control structure for component communication that extends the flexibility of procedure call and message passing. Procedure calls can be viewed as specialized events whose only function is to call a server to request a service and supply data parameters so the service can be provided.

Events cause actions as side-effects, may be oblivious to the actions that they cause, and may cause multiple events in multiple components. Events decouple control from statement execution to a greater extent than procedure calling. Exceptions in traditional programming languages are a restricted form of events that cause the normal flow of control to be modified when exceptional actions are required. They are implemented by exception handlers that handle (catch) exceptions when they are raised (thrown). Event models elevate the exception mechanism to be the primary control structure and generalize it so that occurrence of an event can cause multiple components to be notified of its occurrence.

Event models are especially useful in modeling external input, since the effect of mouse clicks or other user-initiated inputs can be modeled by events occurring at the user interface that cause the system to perform desired actions. Thus "onmousedown" was the most common event of the Hypercard event model, which also allowed events like "onopencard" caused by inner system actions. Since event models provide a uniform control structure to handle both external and internal events, all component and document models offer an event model for both user-interface control and control of certain inner operations.

Since JavaBeans has a nicely described event model we use the JavaBeans model as the basis for discussion. It requires event listeners to register themselves in a registry for the event and causes event sources to notify all registered listeners whenever the event occurs. The data structure transmitted to listeners when they are notified of an event is an instance an event state class. Event handling interfaces are defined in event listener interfaces that are specified by subclasses of an eventlistenrs class.

Visual interfaces catch user events and cause actions by components listening for those events. They handle visual interaction through "listening membranes" activated by mouse clicks that transform black-box computers to glass-box containers with windows that provide access to selected inner structure in browsing, authoring, and scripting modes. Input through a given interface is sequential, though input to a system with multiple input ports involves distributed multiple input streams that cannot in general be expressed as a sequential (serializable) input stream [We1].

Components with visual interfaces, such as beans, are abstractly modeled by events, properties, and methods, which extend notions of control, variable, and operation to express the semantics of visual interaction. Components associated with user interfaces have interactively generated sequential streams of input events controlled by multiple internal threads that generally execute one at a time as in monitors. Events trigger listening actions, properties have visual and/or internal attributes, and methods are callable both interactively and by system calls. Beans may have a visible GUI representation or be invisible and known just by their ability to call methods, fire events, and manage a persistent state.

Extension from algorithms to visual, interactive components:
control: execute unique next instruction -> notify listeners of nonunique next event
variable: hidden internal representation -> visible attribute with interactive interface
operation: procedure call -> message event caused by interaction or inner system condition

Properties (named attributes) extend variables so their values can uniformly represent both visual and inner attributes. Property attributes can be visual (such as being red or rectangular) and occupy real estate on the screen. Properties can be set either interactively by the user or algorithmically by the program. Setting the value of a property can be used to control actions through the event mechanism, so that interactive setting of visual attributes can trigger interactive events. Visual input-sensitive interfaces are porous two-way listening membranes that expose what is inside and can be programmed to receive interactive mouse clicks on any part of their visual real estate at any time.

Methods can be invoked either by traditional system calls or through user- or system-generated events. Invocation of methods by events changes the execution paradigm, allowing nonsequential nondeterministic execution controlled by user interaction as well as program properties.

Traditional programs have restricted input events handled by read statements, do not allow properties (variables) to be set directly by the user, and do not allow methods to be interactively invoked by the user. Visual interfaces require notions of control, variables, and operations to be extended. Visibility and interaction play distinct roles in the new paradigm: visibility by itself facilitates conceptual understanding through browsing, while visual interaction extends computational problem-solving power [We1]:

Visual (browsing) interfaces: Visual interfaces that support browsing enrich understanding and provide system programmers with alternative ways of presenting data. Netscape Navigator and Internet Explorer illustrate the power of noninteractive visual browsing interfaces.

Interactive (authoring, scripting) interfaces: Interaction at the level of authoring includes the use of existing operations for modifying interface objects, while scripting involves the creation of new properties and operators. Interactive interfaces have multiple listening points (one for each mouse position) that allow the user to select among alternative choices and thereby to quickly find objects in spaces with an exponential number of possibilities. The role of interaction in document systems is very different from that of visualization: interaction is not only more convenient but actually more expressive than algorithms as a problem-solving method.

The AWT/JavaBeans event model records the occurrence of events in event objects that are responsible for notifying registered listeners that the event has occurred. Event types are specified by subclasses of the event object class. Condition events notify the associated event object, which in turn notifies listeners. Mouse clicks and other input events are transformed through special hardware and software into messages to associated event objects. Visual interfaces like buttons have implicit events activated by user-initiated mouseclicks on the visible real estate.

Introspection is the process of automatically figuring out the events, properties, and methods of beans. An introspection class provides a uniform method of introspecting by the use of syntactic patterns to automatically discover the events, properties, and methods of classes.

JavaBeans builds on the Java language, class libraries that specify the event and introspection models, and special-purpose interface hardware and software to provide a kit for constructing composite beans and applets from components. We would like to specify the abstract structure of beans by extending specification techniques of traditional programming languages. The event-property-method model of components is a first step in systematically specifying JavaBeans and more generally in describing design structure for compound active documents.

Beans may, as a first approximation, be viewed as collections of collaborating bean components with a beanlike interface. Thus beans are frameworks for compound active documents, document interfaces are GUIs of frameworks, and bean and framework specification are related. Specification techniques for beans and frameworks are further examined below in the context of structured object-oriented programming.

JavaBeans has a language-dependent document model built on a sound language that supports pure interfaces and a lightweight client-server discipline exemplified by Hot Java. Its underlying model provides a simple foundation for dynamic document architectures. To achieve compatibility with CORBA components JavaBeans supports IDL wrappers for Java components that provide bridges from the JavaBeans API to other component model architectures, allowing JavaBeans to operate in ActiveX, OpenDoc, or yet to be designed document environments.

Java and JavaBeans provide a language, system, and application framework that conforms to the layered CORBA services of Figure 1b but is better designed and better integrated than earlier systems. It may well become a standard that supersedes earlier designs and makes language interoperability with earlier languages unnecessary except for legacy code. The cancelling of OpenDoc by Apple may well be a prelude to the wholesale cancelling of CORBA by OMG in favor of a Java/JavaBeans component model.

9. Modes of Interaction

The semantics of software components is specified by their mode of interaction, while that of algorithms is specified by their mode of execution. Modes of interaction provide an outside-in view of components that complements the inside out view of algorithms. The analysis of systems by their mode of interaction is an interactive analog of the analysis of algorithms by their mode of execution. The expressiveness of components is measured by their ability to perceive and interact with the external world, while that of algorithms is measured by their ability to transform inputs noninteractively into outputs.

The irreducibility of components to algorithms can be proved very simply by showing that interactive systems cannot be modeled by Turing machines with finite initial input tapes [We1]. Greater expressiveness follows from the fact that Turing machines with infinite tapes are known to be more expressive than regular Turing machines. The formal proof is corroborated by informal evidence such as:

Fred Brooks' assertion that there is no silver bullet for simply specifying systems
the assertion that everyone is talking about object-oriented programming but no one knows what it is
the failure to realize the goals of computing by logic of the Japanese 5th-generation project
the inability of pattern theory to develop formal pattern specifications

The design space of interactive systems, which includes software engineering (SE), artificial intelligence (AI), and virtual reality (VR) systems is much richer in its variety of behaviors than the design space of algorithms. In the programming-language world, types and classes specify values by their modes of interaction, while in the software-engineering world interfaces specify systems by their modes of interaction. Modes of interaction provide a qualitative framework for analysis, since quantitative complexity analysis is not applicable to systems.

To illustrate the power of qualitative analysis of interactive behavior, we briefly review modes of behavior for SE, AI, and VR. SE focuses on reactive systems that provide services by reacting to the requests of clients. AI focuses on proactive agents that act on their environment to realize external goals. VR focuses on real-time, multimodal, cognitively realistic interaction.

Reactive systems that passively supply services have simpler environment models than proactive agents that try to understand and change the world. If interactive expressiveness is defined by external modeling power, then AI and VR systems are more expressive than reactive SE systems. The reactive services provided by components to clients can be very complex, but clients are modeled as components that make syntactically simple requests specified by operations with parameters. Component interfaces provide both an abstraction of the external world to proactive agents looking outward and an abstraction of the inner worlds of reactive components to clients looking inward.

Modes of interaction provide a unifying descriptive framework for SE. Software architecture deals with alternative modes of interaction like pipes, client-server, and blackboard models. Interoperability examines interaction among heterogeneous components that differ in platform and interface definition. Object-oriented design models specify interaction among objects differently from computation within objects. Design patterns and frameworks determine modes of interaction that can be classified and reused but cannot be proved correct, formally specified, or formally composed.

Modes of interaction also provide a unifying descriptive framework for AI. Learning, planning, and acting have characteristic modes of interaction. Proactive agents actively learn about the external world, build complex internal models of the world, and perform actions to change it. Agents use incremental data to update their model of the world as a basis for action. Planning systems combine updating their world model with the execution of policies that maximize their expected reward over finite or infinite time horizons, while learning systems explore the world and build models for later action.

VR aims to create machines that can simulate the real world in interacting realistically with humans. VR achieves its realism by integration of spatial, stereo, and temporal perspectives: footprints in space are combined with footprints in time to create cognitively realistic virtual worlds. Though VR simulates environments rather than human behavior it requires a detailed cognitive model to provide cognitively acceptable inputs and respond to human actions. Simulating each of the five senses involves entirely different modes of communication and perception that integrate mathematical techniques of pattern description with cognitive properties of perception. The cognitive reconstruction of three-dimensional images from stereo and temporal projections requires interaction patterns to match cognitive expectations if motion sickness is to be avoided. Making spatio-temporal inputs consistent with human cognitive limitations provides an extra dimension of difficulty over and above that of creating adequate interaction patterns.

Modes of interaction can be modeled mathematically as projections of the world on the input sensors of an agent. Stimuli S from an external world W are projections S = P(W) onto input sensors such that the inverse cannot be completely known (W cannot be reconstructed from S). Projection mappings P provide a mathematical tool for analyzing modes of interaction. Incompleteness of S in specifying the world W is related to Godel incompleteness.

An agent's knowledge of the world is expressed by Plato's cave metaphor, which compares humans to cave dwellers who can observe only shadows on the walls of their cave (retina) but not the external world. S is the shadow cast by W on the walls of the agent's cave. However, S can include stereo inputs from multiple sensors (eyes and/or ears) and temporal inputs at successive points of time, and indeed can be a stereo-spatio-temporal interaction pattern more complex than a two-dimensional image on the walls of a cave.

10. Specifying Frameworks by Constraints on Component Behavior

Whereas algorithms are specified by the composition of primitive functionality to define composite functionality, interaction is more naturally specified by constraints on all possible behavior than by how it is built up from primitives. Composition from primitives is necessary to create components and systems, but once components have been created their interactive effect is better described by types and interfaces that constrain interaction than by how it has been constructed for two distinct reasons:

abstraction:. interactive behavior directly describes the effect on the user
noncompositionality: composite behavior cannot be specified by the composition of primitive behavior

Driving is initially learned by understanding controls like the accelerator and the steering wheel, but rules of the road are better specified by constraints on keeping to the road and not hitting obstructions. Behavior for frameworks is more naturally specified by constraints on a space of all possible interactions than by the composition of primitive behaviors.

Constraint-based specification is a “sculpture paradigm” that removes unwanted behavior until only the desired behavior is left, just as the sculptor chips away unwanted material until the desired form emerges.. Constraint specifications are realized by progressively constraining the superset of all possible behavior to a desired form, just as a sculpture is realized by progressively removing material from a block of marble. Behavior specification by constraints that eliminate possible behaviors rather than by composition of behavior is a new specification paradigm that uses the counterintuitive principle “less is more” for system specification.

Frameworks realize collaborative richness by sacrificing the freedom of components that collaborate, just as marriage partners give up some freedom to realize collaborative richness. Constraints are a more powerful behavior-specification technique than composition because they make no assumption about the behavior being constrained, allowing the behavior of nonalgorithmic noncompositional collaborative components to be described.

The abstract modeling of composition by behavior constraints, developed in [We1], is illustrated in Figure 4, where the component C1 gives up behavioral freedom when constrained to collaborate exclusively with C2. Components that in isolation are interactive open systems, free to interact with any client, become noninteractive closed systems when constrained to interact exclusively with each other. Composition may cause open interactive systems to become closed and noninteractive: an open server that provides services to any client becomes closed if its services are entirely preempted by (dedicated to) a particular client. However, *dedicated (closed) composition* that entirely preempts the services of a component is a special case of *open composition* that yields a new open system that can be constrained by further composition.

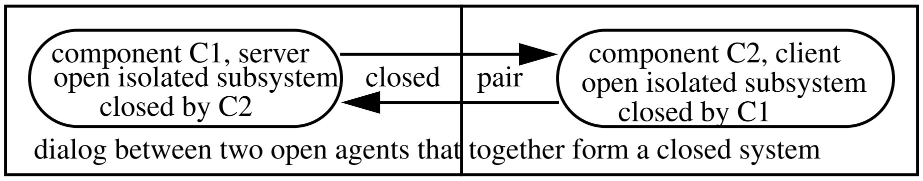


Figure 4: Component Composition as a Constraint on Behavior

The sacrifice of freedom for discipline to realize collaborative behavior is a feature of both software components and people who lose their freedom when they become cogs (algorithms) in corporations. The advantages of application frameworks over class libraries are realized by sacrificing behavioral freedom of isolated objects and classes for collaborative discipline to serve users. Frameworks support the inversion of control from client-side calls to server-side callbacks through strong inner control constraints on components to realize useful interaction with clients.

Viewing composition as a constraint on interactive behavior provides a systematic basis for modeling mathematically dual to algorithm composition. Each interactive composition step constrains behavior to a subset of its free behavior. Constraints are more widely applicable than composition because they can be applied to constrain noncompositional as well as compositional behaviors.

Goal-directed behavior may be created by bottom-up composition of smaller units or by top-down constraints on already existing components. The creation of frameworks from collections of objects, classes, and components clearly falls into the second category. Michelangelo could not have created his David statue by gluing together small bits of marble.

11. References

[AG] Ken Arnold and James Gosling, *The Java Programming Language*, Addison Wesley 1996

[CS] James Coplien and Douglas Schmidt, *Pattern Languages of Program Design*, Addison Wesley 1995.

[Di] Edsger Dijkstra, Goto Considered Harmful, *CACM* 1968.

[JB] Java Beans 1.0 Specification, Javasoft, December 1996

[Kn] Donald Knuth, Structured Programming with Goto Statements, *Computing Surveys*, December 1974.

[OHE] Robert Orfali, Dan Harkey, and Jeri Edwards, *The Essential Distributed Objects Survival Guide*,

Wiley 1996.
[OPG] *OpenDoc Programmers Guide*, Addison Wesley, 1996.
[Ru] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, 1990.
[We1] Peter Wegner, Interactive Foundations of Computing, To appear in *Theoretical Computer Science*, available from www.cs.brown.edu/people/pw
[We2] Peter Wegner, Why Interaction Is More Powerful than Algorithms, *CACM*, May 1997.
[We3] Peter Wegner, Interactive Foundations of Object-Based Programming, *IEEE Computer*, Oct. 1995.
[We4] Peter Wegner, Interactive Software Technology, *Handbook of Computer Science and Engineering*, CRC Press, December 1996.