Chapter 10

Visual Composition of Software Applications

Vicki de Mey

Abstract Open applications can be viewed as compositions of reusable and configurable components. We introduce visual composition as a way of constructing applications from plug-compatible software components. After presenting related work, we describe an object-oriented framework for visual composition that supports open system development through the notion of domain-specific *composition models*. We illustrate the use of the framework through the application of a prototype implementation to a number of very different domains. In each case, a specialized visual composition tool was realized by developing a domain-specific composition model. We conclude with some remarks and observations concerning component engineering and application composition in a context where visual composition is an essential part of the development process.

10.1 Introduction

We define *visual composition* as the interactive construction of running applications by the direct manipulation and interconnection of visually presented software components. The connections between components are governed by a set of plug-compatibility rules specified within a *composition model*.

Visual composition is a response to the trends in software development towards more component-oriented lifecycles described in chapter 1. With a large number of components supplied by component engineers, application development becomes an activity of composing components into running applications. Visual composition can be used to communicate reusable assets from component engineers to application developers, reusable designs to application developers, and open applications to end-users. A visual composition framework enables the construction of environments and tools to facilitate component-oriented software development.

Vicki de Mey, “Visual Composition of Software Applications,” *Object-Oriented Software Composition*, O. Nierstrasz and D. Tsichritzis (Eds.), pp. 275-303, Prentice Hall, 1995.

In this chapter we present a framework for visual composition. The framework addresses four issues: (1) components, (2) composition models, (3) user interaction, and (4) component management. Components are made up of a *behaviour* and a *presentation.* The behaviour is responsible for the component's composition interface and the work the component was designed to do. The presentation is the visual display of the component. A component can have more than one presentation, and the presentations reflect the state of the component. A set of components can be grouped together to function as a single component through the *composite component* mechanism. Component composition is defined as communication between components through their composition interfaces. The framework defines the notions of *port* and *link* to handle the communication. A *composition model* is the set of rules for component composition in a particular application domain. Decoupling the rules for composition from components allows a variety of different software composition paradigms and increases the potential for reuse of a component.

Vista is a prototype implementation of the visual composition framework. A concrete implementation of a visual composition tool is obtained by a *component engineer* (chapter 1) by completing the framework with components, their presentations and the composition model governing their interconnection. Finally, the resulting tool can be used by an application developer to visually compose running applications, as shown in section 10.5.

In reference to software development environments, Ivar Jacobson made the following statement:

> In the long run, we shall see new development environments that place more emphasis on applications and less on technique. Developers will be application experts, not Unix or C++ experts. They will work with graphical objects presented in several dimensions, not simply text. The language of today may be handled as a machine language that is invisible to developers. [23].

These new development environments will have the potential to transform software development. End-users will play a larger role in putting applications together and new ways of creating applications will be necessary. Visual composition is one of these new ways.

10.2 Related Work

Visual composition is based on work done in many different fields from software lifecycles to graphical user interfaces and graphical object editors, visual programming, components and connectivity, and component integration. Since the latter two areas are the most relevant for this chapter, they will be discussed here.

Visual composition supports components and connecting components together to form running systems. Some exemplary systems based on these ideas are ConMan, Fabrik, Silicon Graphic's IRIS Explorer™, Apple's ATG Component Construction Kit and IBM's VisualAge™. ConMan [14] is a high-level visual language that allows users to build and modify graphics applications. To create an application the user interactively connects simple components using a directed dataflow metaphor. No concept of composite components exists. Fabrik [22] is a contemporary of ConMan. Fabrik is a visual programming environment that supplies a kit of computational and user interface components that can be wired together using a bidirectional dataflow metaphor. The environment can be used to build new components and applications. Composite components are supported through the gateway construct. IRIS Explorer is an application creation system and user environment that provides visualization and analysis functionality. It is based on a distributed, decentralized dataflow model. Its graphical user interface allows users to build custom applications by connecting modules together. Apple's ATG Component Construction Kit (CCK) [43] is a prototype component architecture and set of test components that allows end-users to plug components into a framework at run-time. The kit has four elements: (1) a component framework (the structure within which components are connected); (2) a component palette (source of components); (3) an inference engine for automatically connecting components; (4) a component inspector for display and modification of component information. Objects are the medium of communication between components in the CCK. VisualAge [20] is a product from IBM designed to build the client side of client–server applications, focusing on business applications and decision support systems. The tool is based on the "construction by parts" paradigm that is supported by a visual programming tool for creating applications non-procedurally. These systems are interesting but limited since some cater only to specific application domains or are based on one way of expressing the relationships between components.

A component can be seen as a separate tool, application or process. This brings up component integration issues that run very close to the issues of component interfaces and component interconnection. Visual composition needs component integration mechanisms to implement the connections between components. Integration issues can be viewed on two levels: coarse-grained and fine-grained. Coarse-grained integration concerns components that may be large objects that cooperate by exchanging messages, or tools that cooperate through shared files. Fine-grained integration concerns components that are smaller and usually need to communicate with each other more frequently. Harrison, Ossher and Kavianpour [17] have discussed this issue and proposed an approach called Object-Oriented Tool Integration Services (OOTIS). They believe that applications are moving more towards fine-grained integration, but that current systems, which are coarse-grained, must still be supported while this move takes place.

Many proposals have been made for specific solutions to coarse-grained integration. Some examples are: Unix facilities that provide a variety of different tools and tool integration mechanisms (character files, I/O redirection, pipes, shell programming); Hewlett Packard's Softbench environment [19]; and Sun's ToolTalk [26] for interapplication communication. Fine-grained integration solutions include efforts by the OMG (Object Request Broker), NeXT (Distributed Objects [36]), Microsoft (OLE [34]) and Apple (Apple events and the Apple event object model [1], and OpenDoc [2]). See also chapter 12 for a more thorough discussion of these commercial efforts, and chapter 3 for an example of an object-oriented framework to support interoperability.

10.3 A Framework for Visual Composition

The framework we present provides a simple and flexible core for visual composition. There are three pieces of information that are needed in order to use the framework: component behaviours, component presentations and rules for composition. This information is plugged into the framework to produce a visual composition tool for a specific purpose.

10.3.1 Component Definition

The framework defines a component as a *behaviour* together with one or more *presentations*. Such a “division of labour” has been seen in other frameworks including Smalltalk’s MVC framework [13], Unidraw [48] and Andrew [38]. Table 10.1 shows the corresponding terms in the different frameworks. This division promotes reuse, because different presentations can be reused with the same or different behaviours.

Behaviour

The behaviour is responsible for the following:

- Communication with the presentation(s).
- The component’s composition interface. The composition interface advertises the component’s services and requests for services. The composition interface allows the component to be reused in different contexts. A component’s context includes the components it is immediately connected to as well as the entire ensemble of components in which it finds itself embedded. The composition interface of a component consists of a set of *ports*, each of which has a name, a type and a polarity. Ports may be visually presented in a variety of ways, such as knobs, buttons, text fields, menus, etc., depending on the intended semantics.
- Executing whatever the component was designed to do. The behaviour reflects the inner part of the component. From the outside, two components can look like they have the same behaviour, but their internal implementations could be very different (e.g. implemented in different programming languages). A component can also behave differently depending on the other components it is connected to.

Visual composition	Behaviour	Presentation
MVC	Model	View, Controller
Unidraw	subject	view
Andrew	data object	view

Table 10.1 *Comparison of the behaviour/presentation division of labour.*

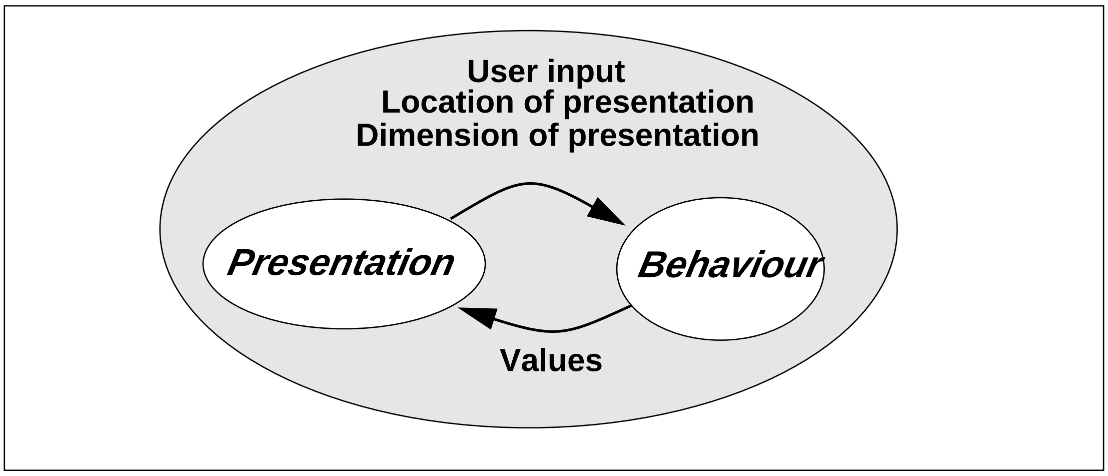


Figure 10.1 *Communication between the behaviour and presentation entities in the framework.*

Presentation

The presentation is responsible for the following:

- Communication with the behaviour.
- Visual display of the component. All components, whether inherently visual or not, have a visual presentation. A presentation can also process user input if it contains an interaction component such as a button or a text field.

Communication between the behaviour and presentation is pictured in figure 10.1. The presentation informs the behaviour of its location and dimensions so that this information can be passed on to other components that need it to display themselves. Also, input can be done through the presentation, and this information is communicated to the behaviour. The behaviour only informs the presentation of information that it might need to display on the screen.

Composite Components

Components can be created by programming or by composition. When a component is created by programming, only the behaviour and presentation need to be specified (or re-used if an appropriate behaviour or presentation already exists) and hooked into the framework. A composite component is a set of components linked together that is considered useful as a component in its own right. To define a composite component, one must add to the set of components (1) a composition interface (by specifying which ports of the set of components are to become ports of the composite component), and (2) a visual presentation (which can be composed of existing presentations). The behaviour of a composite component is simply the behaviour of the components it encapsulates. Figure 10.2 illustrates the idea of composite component. The framework supports composite components by defining *external_port* and *external_view* entities. A set of external_port entities repre-

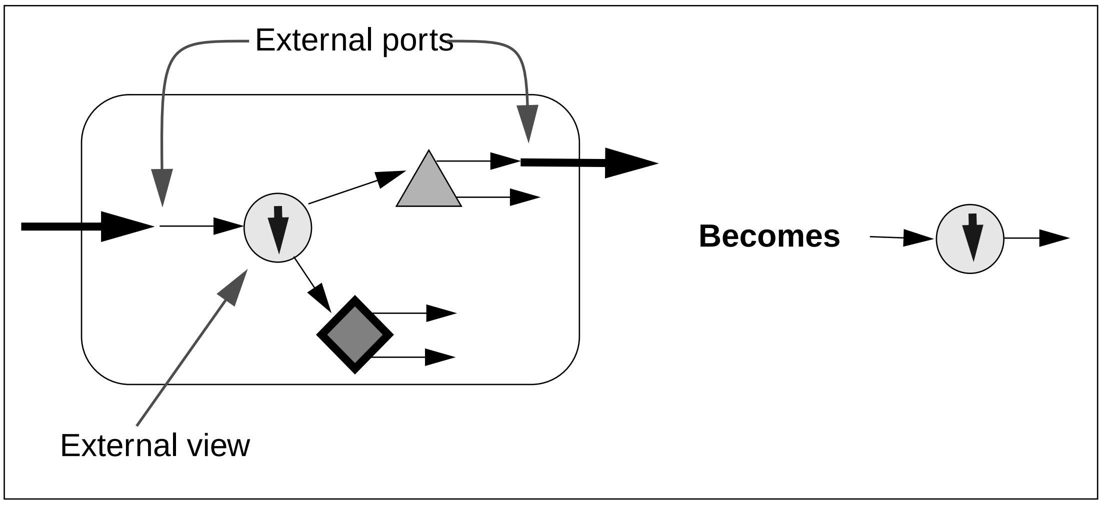


Figure 10.2 *Composite component.*

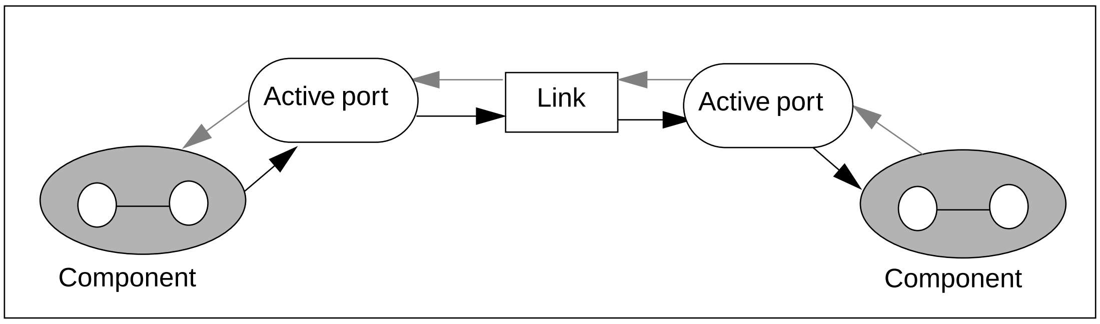


Figure 10.3 *Framework entities.*

sents the composite component's composition interface, and an external_view entity is the presentation used for the composite component. Both external_port and external_view entities are created interactively when a composite component is being defined.

10.3.2 Component Composition

The way component composition is supported in the framework is through the creation of networks. The framework uses components for the nodes in the network and defines *active_port* and *link* entities for edges in the network. Not all ports in the composition interface of a component need to be used. An active_port is created only when a port in a component's composition interface is connected to a port in another component's composition interface. A link represents the connection from one active_port to another. Figure 10.3 illustrates the relationship between the elements of the framework. Communication between components can be either one-way (the dark arrows in figure 10.3) or bidirection-

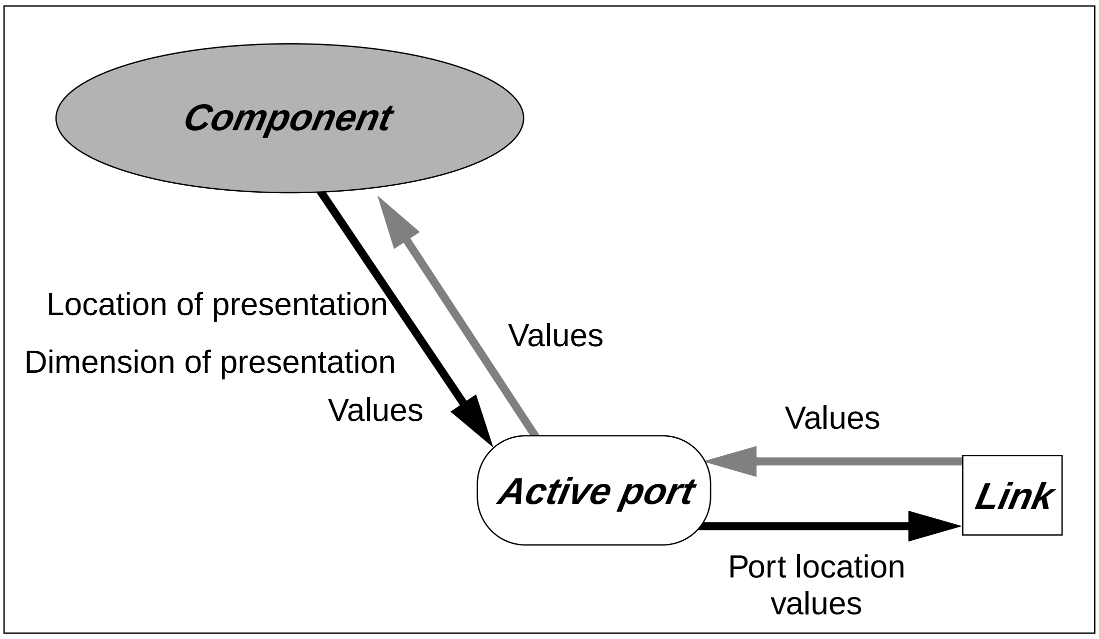


Figure 10.4 *Communication between a component, active_port and link.*

al (both the dark and the grey arrows in figure 10.3). The format of the information that passes from component to component is defined in the behaviour of the component. The format is not restricted by the framework. The display information is internal to the framework and should at least include the location and dimensions of the presentation so that the active_ports and links can be displayed correctly. Figure 10.4 summarizes the information that is communicated between the active_port and link entities of the framework.

As components are composed, networks, such as the one pictured in figure 10.5, are generated. The grey ovals in the figure are components, the black circles are ports, the clear circles are active_ports and the rectangles are links. These networks have certain characteristics:

- *Automatic network update*: Information must be automatically propagated through the network. Propagation occurs when a node indicates some change to the information on its outputs. This indicates the need for some type of constraint mechanism to specify the relations in the network that must always be satisfied. Information propagation could imply some change to the display that must be done automatically and immediately to support the requirement of direct manipulation. There must be the option for immediate propagation or batching for the display, since the update of a densely populated screen can be expensive and possibly postponed
- *Hierarchical decomposition*: The network must support nodes that are made up of other network structures.
- *Cyclic networks*: In visual composition, the relationships between components can create cycles in the network.

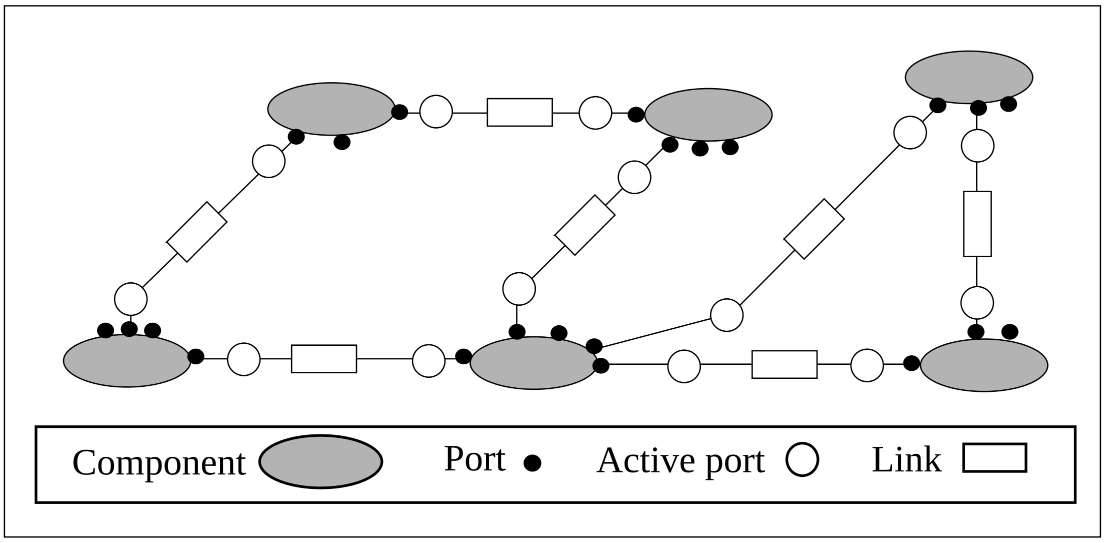


Figure 10.5 *Network of components.*

The division of labour among components, active_ports and links is flexible, but in general, a link is used to transport information between two active_ports, an input active_port controls whether information should be passed into a component, and an output active_port packages up information for leaving a component. Both active_ports and links can have an associated visual display.

Composition Model

The active_port and link framework entities do not impose rules on connections; they just enable connections. Whether or not a connection is valid is determined by a *composition model*. The composition model is the set of rules for component composition in a particular application domain. The rules determine compatibility between components, i.e. which component can be linked to which other component. The type of rules is open-ended and based on the component, the component's composition interface, and other application domain-specific information. Different from many other frameworks, the compatibility between components is not determined by the components. Decoupling composition models from components supports a variety of different software composition paradigms and increases the potential for reuse of a component, because a component can be reused without modification in different application domains by associating it with different composition models. A composition model is *active* when it is dynamically applied to a set of components to get them to cooperate in a specific application. The composition model can have some knowledge of what components it can be used with, but usually components do not have to be designed with particular composition models in mind.

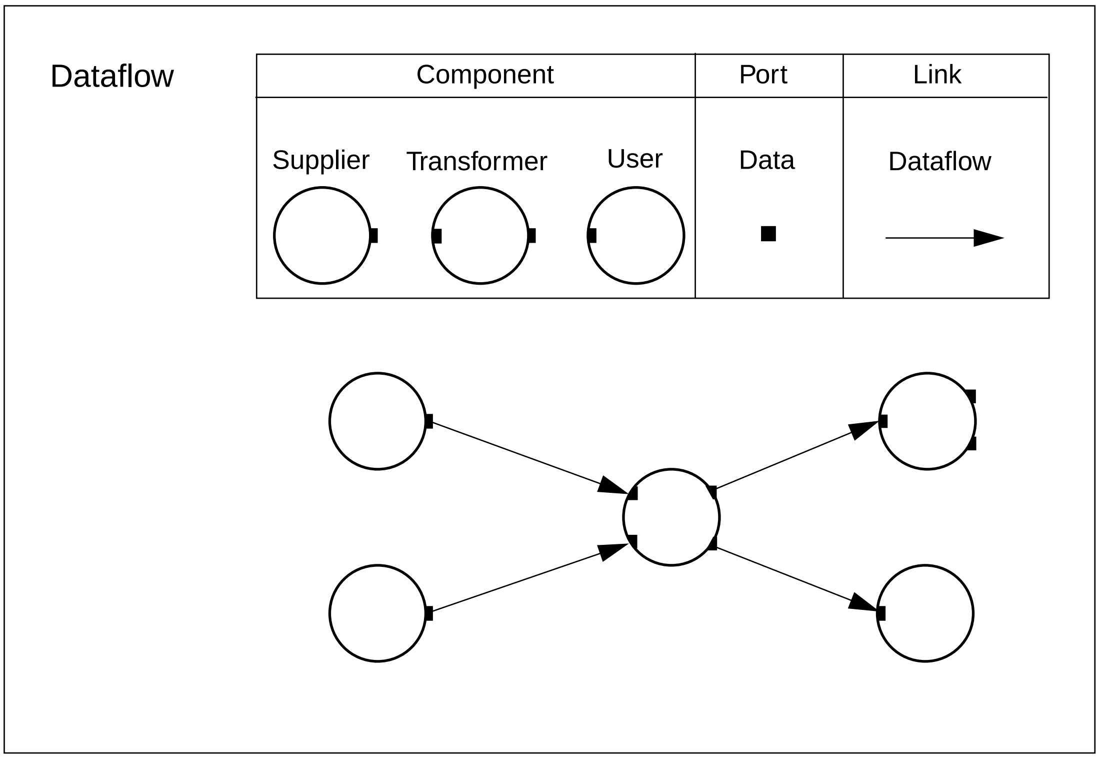


Figure 10.6 *Dataflow composition model and some connected components.*

Composition Model Examples

Three examples of composition models will be given: dataflow, two-dimensional graphic objects and class hierarchy diagrams. Dataflow composition is used for specifying flows of data in a network of components. Components have input and output ports through which dataflow. Each data value is associated with some component responsible for computing the value as a function of its inputs. The component makes the data value available at one or more of its output ports. Input and output ports can be joined by links. All ports have an associated type reflecting the type of the data that passes through the port. If a component has only output ports, then it is a supplier of data; if it has only input ports, then it is a user of data; and if it has both input and output ports, then it is a transformer of data. Links represent data flowing between components. Links are primarily responsible for enforcing valid dataflow networks. They allow ports to be connected only if they have compatible types and compatible directions (input to output and output to input). The dataflow components and composition model are pictured in figure 10.6

Another example is a composition model for two-dimensional graphic objects. This model is used to attach and keep two-dimensional graphics objects connected. A component (a graphic object) has ports that represent points on the object, i.e. ports are of type point. These ports are either input or output depending on the operation being carried out on the graphic object to which they belong. The points are the location of the object in two-

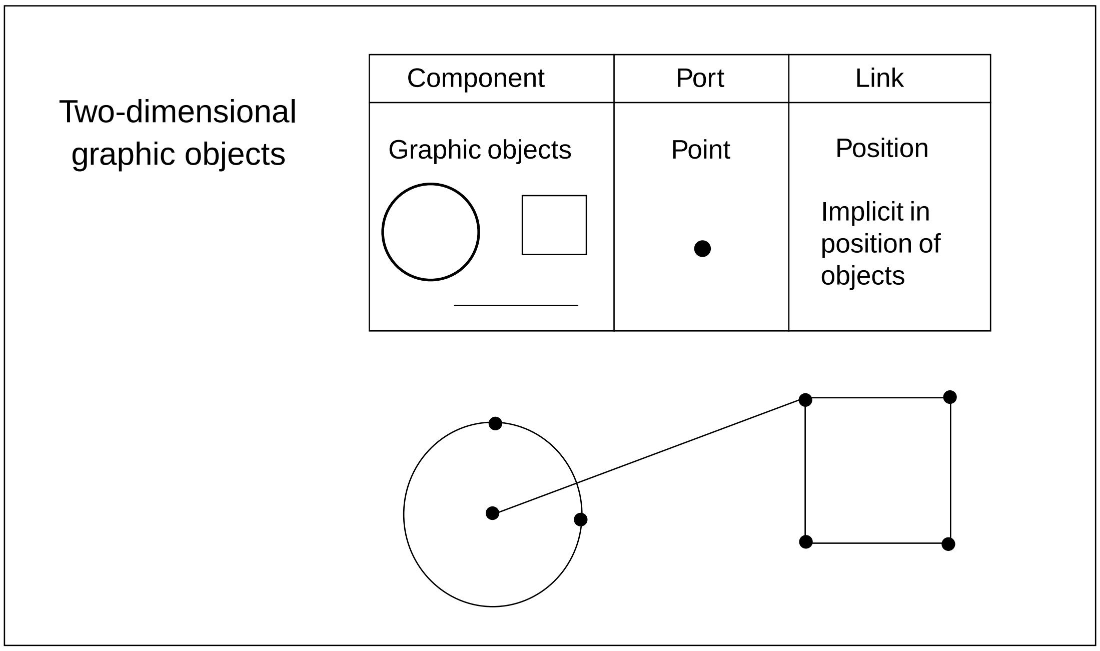

Two-dimensional graphic objects
Component
Port
Link
Graphic objects
Point
Position
Implicit in position of objects

Figure 10.7 *Two-dimensional graphic composition model and some connected components.*

dimensional space. Links are created between components by placing one point on top of the other. Any point can be linked to any other point. When some point changes, as when a graphic object is moved, all other points linked to this point will be updated. This update guarantees the connectivity between graphic objects. The composition model for two-dimensional graphic objects is pictured in figure 10.7.

A third example is that of a class hierarchy diagram. Class hierarchy diagrams are used to show the class structure of an object-oriented application. There is one kind of component for this type of diagram, namely the class component. The class component can have ports of type subclass and superclass. Some class components will not have ports of type superclass; these are leaf classes in the class hierarchy. Some class components will not have subclass ports; these components are top-level classes in the class hierarchy. Links, which represent class relationships, are unidirectional and connect ports of type subclass to ports of type superclass. Links do not pass data from one class component to another; their role is to make relationships between classes so that when a class is asked about its superclasses and subclasses, it will be able to respond. The class hierarchy components and composition model are pictured in figure 10.8.

10.3.3 Interactive Environment

The interactive environment is responsible for ensuring that the framework entities are used correctly. The interactive environment supplies ways for the user to manipulate

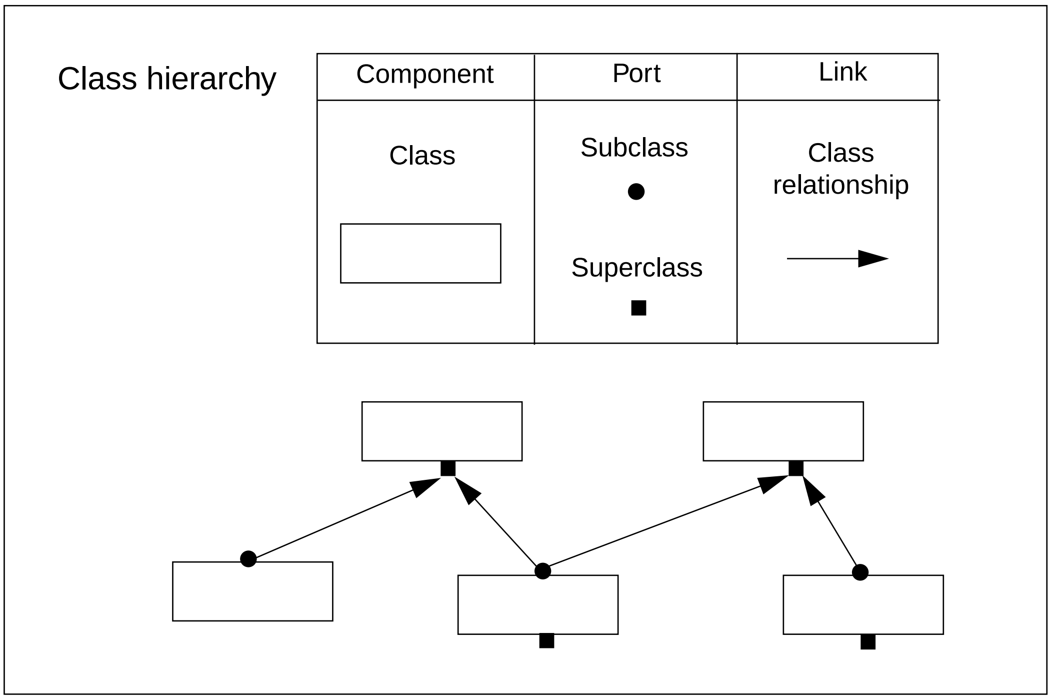


Figure 10.8 *Class hierarchy composition model and some connected components.*

components, ports and links. This is accomplished using various user interface metaphors depending on the implementation. The operations listed in table 10.2 make up a minimal set. The interactive environment is customizable so that operations can be added and/or removed.

The interactive environment supports direct manipulation [42]. Direct manipulation makes objects more concrete and thus helps people to grasp better the ideas and concepts being described. Users get immediate feedback to their actions and are informed when any of their actions cause a change in the system. Ideally, users always know what to expect from the system. Development becomes more of an exploratory activity where a “try it and see what happens” attitude is encouraged. Being able to see immediately what is going on in an application is important in the early stages of application development. At the same time, the interactive environment is as transparent as possible so that users do not have to do things that make no sense in their specific application domain.

The environment is visual — a mixture of text, graphics (two-dimensional and/or three-dimensional), and other media like video, sound and images — and therefore visibility control is very important. Visibility control is used to modify the presentations of components, ports and links. There are operations (see table 10.2) that allow a user to make elements invisible according to different criteria like, for instance, what group they are in, what component they are attached to, etc. An application developer using visual composition usually needs more objects visible than an end-user. The developer is working in more general terms, while the end-user is working in a specific domain. To accommodate this situation, visibility control can hide information that is not appropriate. The composite

Components	Ports	Links
instantiation make a functional copy of the chosen component	**identification** name, type and polarity of port	**create** Manual creation relies on the user manually selecting the start port and target port for the link Automatic connection would attempt to automatically connect compatible ports when a component is placed in a composition composition model verifies link
determine location in space fixed into a particular location in space, fixed in relation to other components' locations, or left free to be moved		
copy state is also copied		
delete links referencing the deleted component are also deleted		**delete**
replace The links that referenced the original component are reconnected to the new component		
display different presentations		

Table 10.2 *Manipulations on components, ports and links.*

component capability can also be used to shield an end-user from unwanted or unnecessary detail.

Some of the more common characteristics of interactive environment must also be considered. It is necessary that some type of grouping mechanism be available, such as windows in the desktop metaphor. The environment supports the undo/redo functionality, because all users make mistakes. Skill levels, such as novice or expert, are ways of helping users learn to use a system. These levels usually assume that a novice is not familiar with the system and therefore needs a bit of “hand-holding.” Expert users, on the other hand, could consider such hand-holding distracting. Skill levels can be implemented by allowing or disallowing certain actions on the objects being manipulated. Different presentations of a component can be used to reflect different skill levels. To further facilitate the usage of the environment, mode switches, such as those between building and running an application, are minimized. The literature [39] suggests that avoiding such mode switches is important for new users because it gives them the flexibility simultaneously to use and modify an application, and that typically there is almost no confusion about when input is directed to the tool and when it is directed to the application.

10.3.4 Component Management

The activities of storing, organizing and retrieving components are external to the framework, but the framework supplies a simple mechanism that records enough information about the network to a text file so that when the file is read, the network can be re-created. Composition models also need to be stored, organized and retrieved. The information stored in the composition model must be activated when a network is used and constantly accessible since it is consulted whenever a link is created and possibly whenever information flows through a link. Some notation, possibly textual, for the composition model is necessary. The framework supports “hooks” for more sophisticated tools, such as the software information base described in chapter 7, for these activities.

10.4 Vista — A Prototype Visual Composition Tool

Vista is a prototype visual composition tool based on the framework described above. The prototype is meant as a test-bed for the visual composition framework. Vista [29][37] was developed as part of ITHACA’s application development environment. (See the preface and chapter 7 for more information about the ITHACA project.) Vista is a second generation prototype; some of the ideas of visual composition were demonstrated in VST, an earlier prototype based on a Unix composition model [45]. Vista is meant to be a simple and evolutionary prototype.

The layers of software supporting Vista are pictured in figure 10.10. The major parts of Vista are in the shaded region of the figure. The implementation of Vista conforms to the ITHACA software platform which includes C++, X Windows and OSF/Motif. Graph management for Vista is supplied by a set of class definitions and functions extracted from the Labyrinth System [27], a generic framework for developing graphical applications. In Vista, components and links are displayed on the screen using Motif widgets and the display capabilities of Labyrinth.

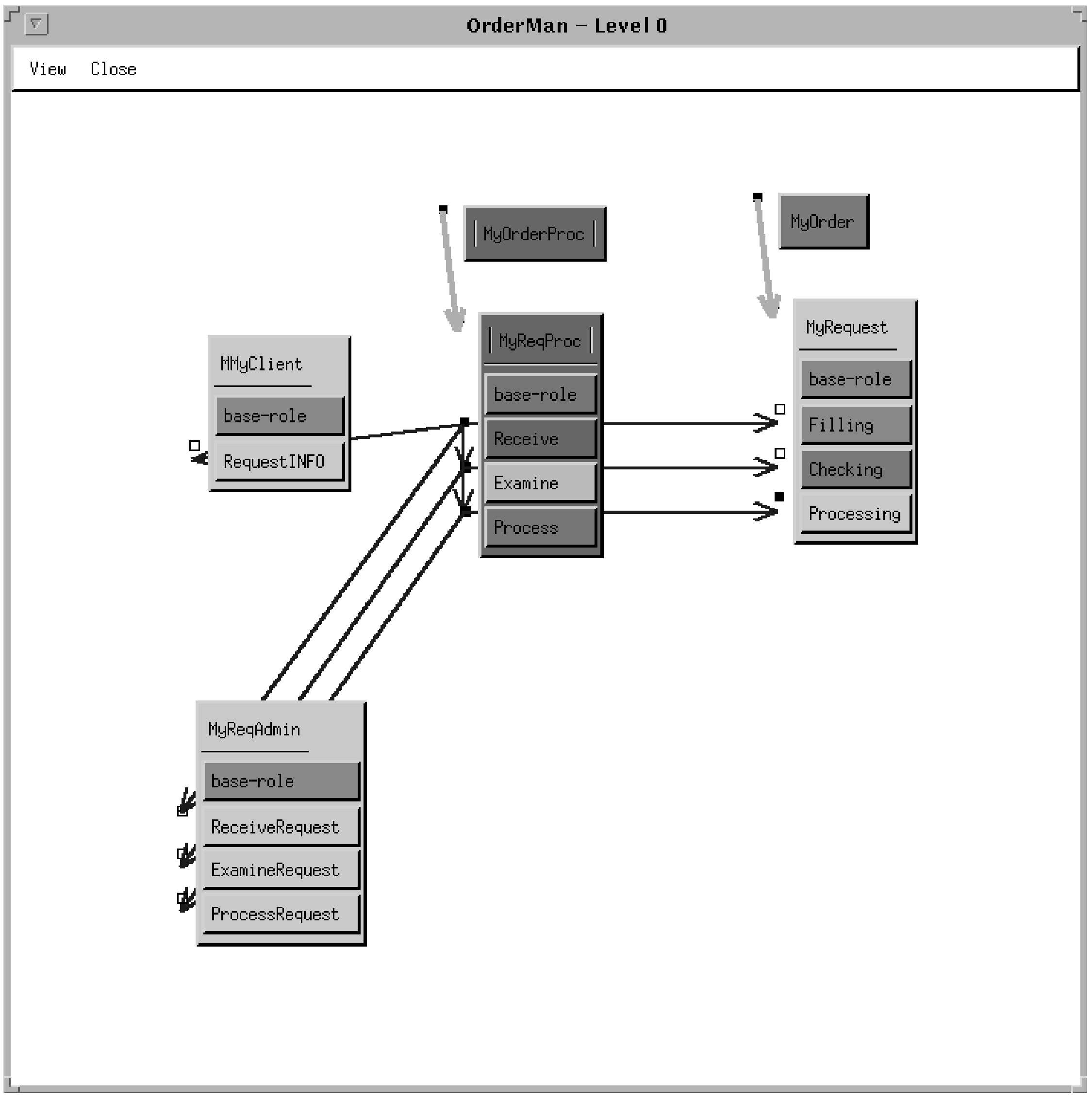


Figure 10.9 *RECAST example.*

Composition Model Manager

There are at least two ways to go about implementing the functionality of the composition model. One possibility is to implement a composition model manager as an oracle that oversees the correct functioning of the framework based on the active composition model.

The other possibility is to delegate responsibility for checking and maintaining compatibility rules to framework entities. If composition models are used to define global compatibility between components or global rules encompassing large groups of components, an oracle would probably be the best choice. The oracle can have an overview of sets of

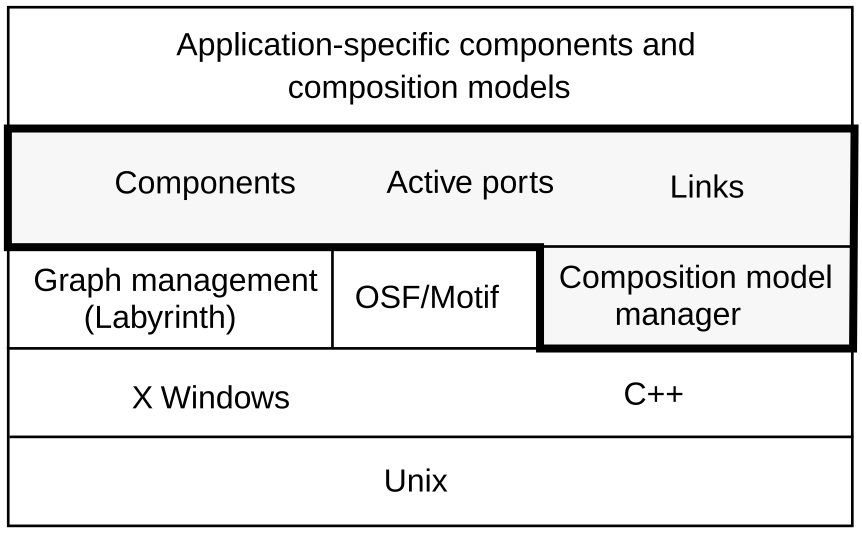

Application-specific components and composition models
Components
Active ports
Links
Graph management (Labyrinth)
OSF/Motif
Composition model manager
X Windows
C++
Unix

Figure 10.10 *Layers of software supporting Vista.*

networks on which to base global decisions. Some powerful graph management systems can also work on a global level in the network; thus the decision for a certain type of graph manager could influence the implementation of the composition model manager. In Vista, the composition model manager is an oracle implemented as a C++ class. A composition model is expressed in a textual notation that expresses port compatibility and is parsed by the composition model manager.

Components, Active_ports and Links

In Vista, components (behaviours and presentations), active_ports and links are implemented as C++ classes. Vista supplies default active_port and link classes. To implement the component behaviour and presentation, four classes are defined. The behaviour of a component is implemented in the *cmp_Node* class and subclasses of the abstract class *V_Base*. The presentation of a component is implemented in the *view_Node* class and subclasses of the abstract class *Framer*. The cmp_Node and view_Node classes are internal to Vista and maintain network connectivity. The V_Base and Framer classes are subclassed by the user of Vista. This division is advantageous because Vista is implemented in C++ and modifications to superclasses necessitate recompilation of subclasses. The division avoids much of this recompilation since the information added from outside the system does not directly impact the internal framework classes and vice versa.

Dividing up the responsibilities of components, active_ports and links is not always straightforward. For example, if two ports of different types are defined to be compatible by a particular composition model, and they are linked, where should the coercion between port types take place? The input active_port participating in the connection has a type and can coerce things that it receives into that type. The link participating in the connection knows it is connecting ports of compatible types and can coerce the type of the information from the output active_port to the type that the input active_port expects.

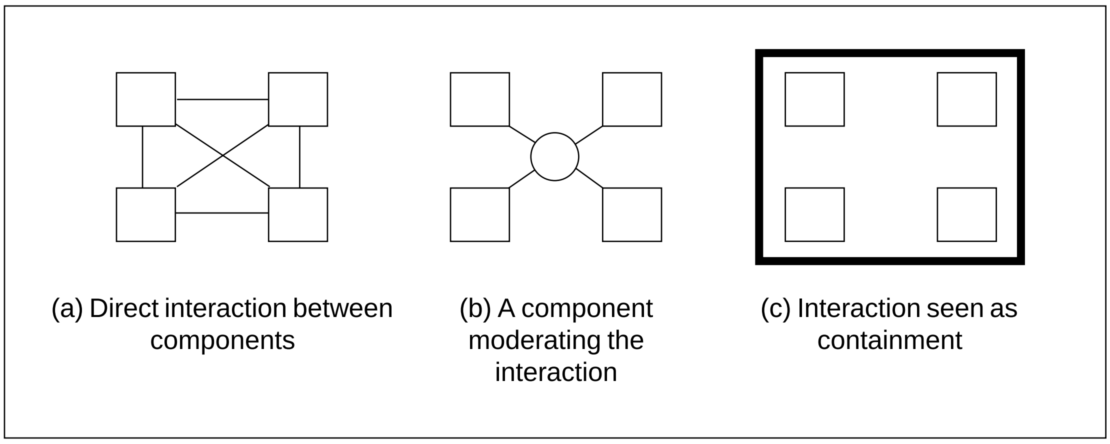

(a) Direct interaction between components
(b) A component moderating the interaction
(c) Interaction seen as containment

Figure 10.11 *Four-way component interaction.*

Responsibilities can be divided up in certain ways to ameliorate the user's experience. For example, a four-way interaction where all four components interact. The interaction can be a new component with the original four components linked to it by "interaction" links or all of the components can be linked together. These two options are pictured in figure 10.11. Figure 10.11(a) might not look hard to understand as it is pictured here, but the picture would become indecipherable if twenty components were interacting. Ports could accept more than one connection, but this does not reduce the number of links. Adding components to a system does not necessarily imply that the system becomes more complex. The number of components is not that important; it is how the components are presented that is important, as seen in figure 10.11(b) and 10.11(c). In figure 10.11(c) links are represented by the location of components (e.g. all components contained in another component interact) and not by lines connecting the components.

In Vista, active_ports and links can be used to implement any of the scenarios described above. But as classes are developed, the goal should be to keep active_ports and links simple, since too much hidden behaviour will make the system unpredictable and harder to understand.

Application-Specific Components and Composition Models

Components and composition models for specific domains are defined using the classes and functionality of the lower layers of software. For components, only the behaviour and presentation need to be defined as subclasses of abstract classes supplied by Vista. Composition models are expressed in the textual notation supported by Vista.

10.5 Sample Applications

Johnson and Russo [25] have stated that:

a use of a framework validates it when the use does not require any changes to the framework, and helps improve the framework when it points out weaknesses in it.

Iteration is a major part of the validation cycle of a framework. The framework should get better (more reusable) as results are gathered from its usage. To this end, the visual composition framework was used in three sample applications. The sample applications came from actual ongoing projects and were not artificially devised as test cases for visual composition.

The first application was part of a project that addresses the creation of a framework and rapid prototyping environment for distributed multimedia applications [32] (see chapter 11). The visual composition framework was then used to implement a visual composition tool for multimedia applications.

The second application was part of a project that addresses the requirements collection and specification phase of software development. The project defines a methodology that functions as a formal basis for requirements specification and support tools. The visual composition framework was used to implement one of the support tools called RECAST.

The third application was part of a project that addresses workflow applications. The project defines a complete environment for designing and running coordination procedures. The need for some type of visual representation of coordination procedures was recognized by the participants in the project. The visual composition framework satisfied this need and was used to draw pictures of coordination procedures and generate the code that the procedures represented.

The scope of these sample applications varies considerably. The first deals with running applications and components represent actual executing modules.The other two are related to software specification, where components represent elements of the software specification model. We will describe the first and second sample applications here. More information about all three applications can be found in the author's thesis [30].

Sample Application 1: Multimedia Component Kit

Chapter 11 introduces the basic concepts of a multimedia framework and multimedia components. Based on this work, multimedia components and composition models were created for visual composition. The visual composition tool for multimedia applications is a rapid prototyping tool for experimenting with different combinations of multimedia components.

A multimedia application is implemented by large number of interconnected hardware and software components. Visual composition can be used to interactively plug components together — rather than permanently "hard-wiring" them — thus making applications more flexible and reconfigurable.

Various composition paradigms are appropriate to multimedia applications. Three example composition models follow:

1. *Dataflow composition* describes an application as a configuration of media components and the data (media streams) that flow between them.

2. *Activity composition* describes the behaviour of an application with respect to activities and events.

3. *Temporal composition* describes relationships between temporal sequences and is a special case of activity composition [31].

These three ways of viewing multimedia applications can be reflected in composition models that determine the types of components useful in the applications and how the components interact. The dataflow composition model has been implemented and will be discussed in detail here.

Viewed from a dataflow perspective, a typical multimedia application accepts user input and displays multimedia information related to the input. This type of application can be built from components that represent input devices and multimedia renderers. Some example multimedia components for dataflow composition are listed in table 10.3. The GeoBall and Navigator components are responsible for getting user input into the application. The GeoBall component represents an actual hardware device, pictured in the presentation of the component, that generates 4×4 geometric transformation matrices. This component can be considered a producer of information of type GeoTransSeq (sequences of 4×4 geometric transformation matrices). The Navigator component, a transformer component, is responsible for taking the information produced by the input device and transforming it into a type understandable by other components in the application. Here, the Navigator component produces information of type MoveSeq (sequences of Render requests dealing with movement of objects in the three-dimensional world).

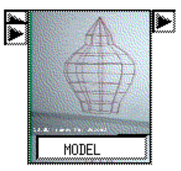
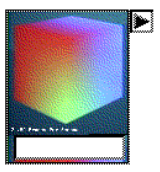
GeoBall	Renderer	Modeler	Navigator	ActiveCube
				
out: GeoTransSeq	in: RenderSeq	in: MoveSeq in: RenderSeq out: RenderSeq	in: GeoTransSeq out: MoveSeq	out: RenderSeq

Table 10.3 *Components for dataflow composition.*

The Modeler component represents the content that will be displayed by the application. The Modeler gathers together all the information in the application that will contribute to the content and prepares the information for display. The Modeler can accept information from the Navigator component to incorporate user input into the display, as well as information from other components that generate content such as the ActiveCube component. The ActiveCube component represents a graphical cube object that can be displayed. The Modeler produces information of type RenderSeq (sequences of Render

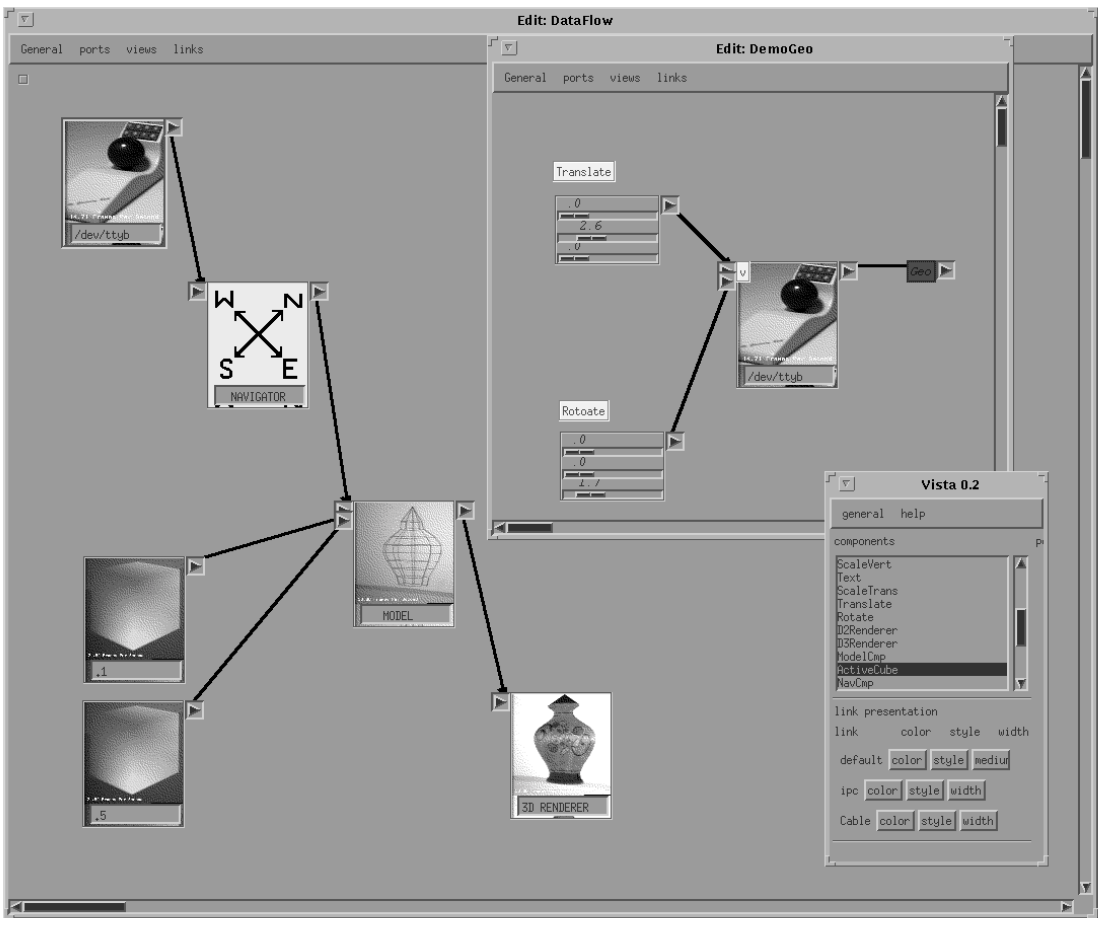


Figure 10.12 *Multimedia dataflow composition.*

requests) that represent the content in a format suitable for rendering. The Renderer component accepts information of type RenderSeq and is responsible for its display.

A simple composition model for dataflow can check to make sure that only ports of the same type are connected as a user interactively creates an application. This model ensures that the components making up the application are correctly connected but cannot ensure that the application is producing the desired result. More semantic information can be put into the composition model to produce a particular desired result. For example, a rule like “if a GeoBall component is in the application, then it must be connected to a Navigator component” would eliminate the need to explicitly connect them, and would guarantee the correct use of these two components.

Figure 10.12 shows a screen image of the tool displaying a simple dataflow of a multimedia application using the components described in the preceding paragraphs. The GeoBall component is implemented as a composite component. The internal view of the composite component is pictured in the upper right side of the figure. The composite component contains a geometry ball component and two sets of horizontal sliders. The sliders adjust the parameters of the device (*x*, *y*, *z* translation and *x*, *y*, *z* rotation sensitivity). These parameters can be changed interactively and thereby modify the behaviour of the input device as a user navigates through the museum. The input device is connected to a Navigator component, which connects to a Modeler component, which, in turn, is connected to a Renderer component. ActiveCube components, which move cube-shaped graphic objects given a velocity, are also connected into the Modeler component.

Sample Application 2: RECAST

The RECAST tool [3][4] in the ITHACA software development environment uses a composition-based approach to requirements specification and provides assisted inspection of available components by accessing the software information base (SIB). RECAST assumes that requirements are specified according to an object-oriented specification model, called the Functionality in the Objects with Roles Model (F-ORM), which is used for requirements representation. The model is based on the object-oriented paradigm extended with the concept of roles [8] to represent the different behaviours that an object can have during its lifetime. F-ORM is a textual definition language and RECAST is a tool that graphically manipulates F-ORM requirements. RECAST was built using the visual composition framework. F-ORM classes and class elements are represented by components and class relationships are represented by links. The class relationships are recorded in a composition model so that a user of RECAST is assured of using F-ORM correctly.

The composition concept of RECAST is reflected in a set of diagrams defined by the ITHACA object-oriented methodology [9]. The methodology defines five types of diagrams at the application design level: class diagram, cluster tree diagram, cluster cooperation diagram, state/transition diagram and IsA/PartOf diagrams. These five diagrams are used by the application designer when specifying requirements. The class diagram implemented in RECAST is described here.

The class diagram represents F-ORM classes and roles along with the dependencies between classes and roles. Classes have corresponding Class components in the framework. The presentation of the Class component contains at least the name of the class. Roles have corresponding Role components and are graphically displayed embedded in the Class component to which they belong. Class components can be in their *base* representation, where all the roles are visible, or in a *compact* representation, where only the class name is visible. Role components are connected by message links that represent either unidirectional or bidirectional message flows.

Colours are used extensively to distinguish different types of components and different types of links. Shading is used to signal if certain classes or roles are selected. For example, if a class or role is not selected, it is not shaded. If a class or role is darkly shaded, then it was selected by the user. If a class or role is lightly shaded, then it was selected by a possible design action. If a class or role is moderately shaded, then it was selected as a consequence of a design action.

The components for class diagrams are summarized in table 10.4. The behaviour of the Class component is responsible for changes in the F-ORM requirements specification. These changes are made by direct manipulation of the graphical representation of the class. Such changes include the addition of classes to the diagram, adding and deleting roles, and the transformation of a class into a set of classes. The behaviour is also responsible for the display of the class (base or compact) and the display of information about the class. The behaviour of the Role component is responsible for managing design actions of the role, the presentation of the role, access to the properties of the role, and access to the state/transition diagram for the role. The behaviour of the Cluster-reference component is responsible for mediating the connection between the active class diagram and the cluster the reference is representing.

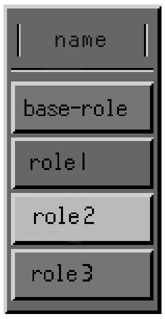
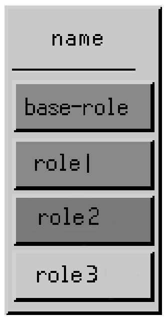
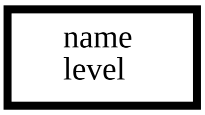
Process class	Resource class	Role	Cluster-reference
			
in: message in: role out: message out: class in/out: class in/out: class in/out: message	all from process class plus: in/out: resclass in/out: resclass	in: message in: class in: stdiagram out: message out: role in/out: message	in: message out: message in/out: message in/out: class in/out: class in/out: resclass in/out: resclass

Table 10.4 *Class diagram components.*

Figure 10.13 shows the class diagram for the OrderManagementSystem cluster. This diagram is generated by using RECAST to transform a request-processing application into an order-processing application. Interacting with RECAST in the following way produces the OrderManagementSystem cluster:

1. The user starts a new application called OrderManagementSystem.
2. The user consults the SIB for the application domain and a generic application frame. In this case the application domain is sales and the generic application frame is request processing. The RequestProcessing class is chosen.
3. The RequestProcessing class is copied into the diagram generating a Class component called MyReqProc. The MyReqProc component has a set of associated roles, represented as Role components in the diagram, which are embedded inside the presentation of MyReqProc.

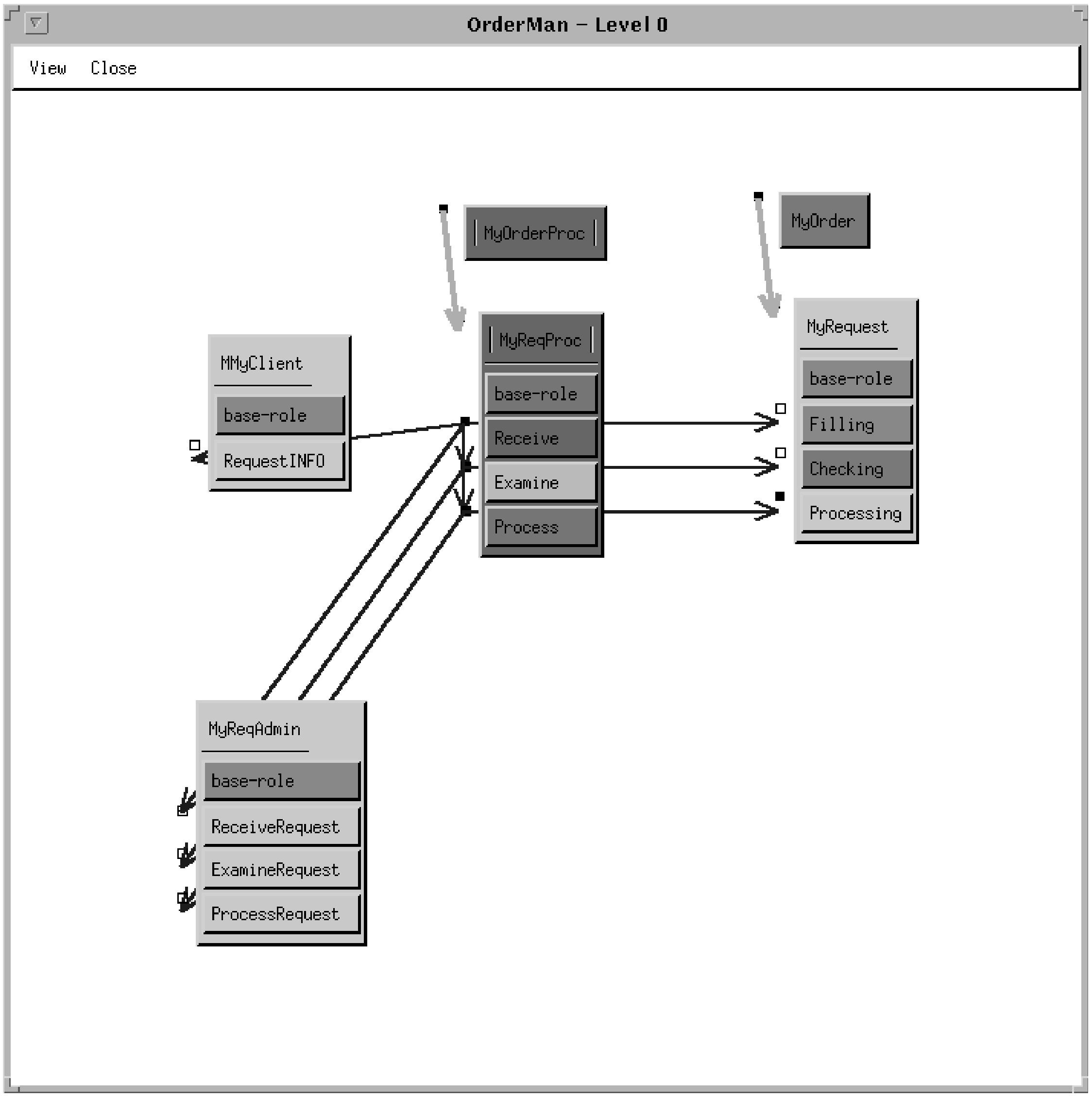


Figure 10.13 *RECAST example.*

4. The Receive Role component is selected. Because of the selection, RECAST consults its design suggestions and suggests adding a request manager agent and a client agent to the diagram. The client agent represents the source of the information for MyReqProc. The request manager agent represents the requested information of MyReqProc.

5. The suggestions are executed and the corresponding Class components are retrieved. For the client agent, MyClient class is retrieved. For the request manager agent, MyRequest and MyReqAdmin are retrieved. The message links, represented by thin black arrows, between all the Class components are displayed.

6. MyReqProc is specialized to MyOrderProc and MyRequest is specialized to MyOrder. Specialization is represented by thick grey arrows.

The sample applications demonstrate that the visual composition framework makes the process of building graphical, component-based applications easier by supplying much of the infrastructure such applications require.

10.6 Discussion

As a result of the work done on the visual composition framework, the prototype implementation of a visual composition tool and the sample applications, certain suggestions concerning component definition, composition and visualization can be made.

10.6.1 Component Definition

The choice of components for a particular application is still somewhat *ad hoc*, and results from other fields, such as the reverse engineering of applications, could supply useful information for component design. Work by, among others, Johnson [24] [11] on framework design has highlighted some key strategies such as finding common abstractions, decomposing problems into standard components, parameterizing and finding common patterns of interaction. With this work in mind and the experience from the sample applications described in the previous section, we present the following list of guidelines for designing reusable components:

- If a concept is used, or looks like it could be used, in a number of different places or application domains, this concept should probably be a component.
  If the same concept were included explicitly in every component, there would be the same functionality spread throughout the component set, proliferating redundant information and negating efforts of encapsulation and reuse.
  Parameterization can be used to generalize concepts that might look different at first glance but, with further investigation, they could really be the same concept with different values for a few different parameters.
  User interface components are a good example since they represent concepts that are reused in many applications. The Modeler component in the multimedia component kit is also a good example. If the Modeler component had not been separate from the 3DRenderer component, the same model information would not be able to drive two different renderers, say a three-dimensional and a two-dimensional renderer, at the same time.
- If a component requires an undetermined or variable number of resources, these resources should probably be components.
  Since this type of information can be quite variable, it seems natural to define a set of lower-level primitive components and compose them into different configurations.

An example of this is found in RECAST for the implementation of the class and role components. Since a class can have a variable number of roles, and roles can be dynamically added or removed, it is much easier to make classes and roles different components that share the relationships roleOf and class than to include the role as part of the class.

- Composite components should contain a small number of components and take advantage of hierarchical decomposition.
  This allows concepts to be organized more effectively and reduces the amount of screen clutter.
- The number of components should be small but extensible.
  A small set of components helps people remember what components they have to work with, but the set must be extensible so that when valuable new primitives are discovered they do not have to be simulated with the existing components.
- Components should strike a balance between concreteness and abstraction.
  From their experience with the world around them, people are more used to thinking concretely rather than abstractly. Visual composition uses a set of components, which are abstractions, for application construction. A balance between the two must be made. Components cannot require the user to fill in every detail — if they had to do that, then visual composition would be worthless. But components cannot conceal all the details since the user would never figure out what to do with them.
- Big components can be reusable.
  It is claimed that the bigger a component gets, the harder it is to reuse [5]. Here, "bigger" means more complex *and* more specific. A component being more specific does make it harder to reuse, but being complex does not have to cause problems. As long as the composition interface of a component correctly reflects its behaviour, more complex components can be reused just as well as less complex components.

10.6.2 Composition

Choosing an effective set of rules for composition is not easy. Visual composition delegates this decision to the users, on the basis that they know best how components should communicate in their particular application. More effort is needed to determine common sets of rules that are often observed in applications as well as domain-specific rules. Efforts such as the Law of Demeter [28], contracts [18], law-governed systems [35] and design by contract [33] all contribute to this area.

Alternative ways for expressing composition models are needed. The composition model was implemented as a small textual language (essentially just listing the port types and their compatibilities) in Vista. It could be useful to use visual composition to describe composition models. Certain rules, like no cycles allowed between components, could be illustrated by diagrams like the ones pictured in figure 10.14. If a component set contains a huge number of components, and a composition model has a huge number of rules, both

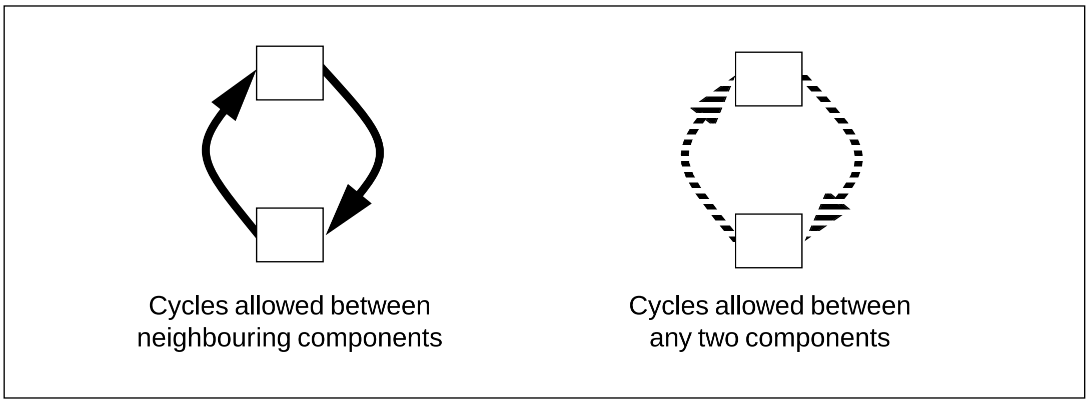

Cycles allowed between neighbouring components
Cycles allowed between any two components

Figure 10.14 *Example visualizations of some composition model rules.*

the textual and graphical representation of the model could get awkward, and another strategy might become necessary.

10.6.3 Visualization

The visual aspect of visual composition is one of its most important features. Being able to see the pieces that make up applications and manipulate these pieces directly contributes greatly to the understanding of an application. Seeing the impact of certain design choices is very advantageous. But, as with all visual communication, a suitable presentation must be chosen otherwise the effectiveness and quality of the visual expression will be brought into question. What is suitable can vary from person to person, so flexibility is important, but people still have to understand each other. Any enhancements to the visual presentation must give a user a more comfortable, informative and familiar environment in which to work.

Visualization is the visual representation of information using, for example, two-dimensional/three-dimensional computer graphics. Solutions and problems can often be easier and faster to recognize visually than having to sort through program text. Brooks does not favour visualizing software, saying: "In spite of progress in restricting and simplifying the structures of software, they remain inherently unvisualizable, and thus *do not permit the mind to use some of its most powerful conceptual tools*" [6] (emphasis added). Despite Brooks's pessimistic view, a person's visual capacity is such a powerful conceptual tool that it must be explored as a possible aid in dealing with complex systems. It is possible that not every level of software is visualizable, but this should not limit attempts to try to profit from visualization where appropriate. Scientific visualization has "demonstrated previously unknown flaws in numerical simulations and made possible new knowledge and insight" [41]. Harel [15] takes a more optimistic view about the possibilities of visualization. Like Brooks, he agrees that the "traditional" diagrams, such as flowcharts, are not what is needed for modeling systems. But a lot of the conceptual constructs underlying software systems can be captured naturally by notions from set-theory and topology that have natural spatial/graphical representations. Harel calls these representations *visual formalisms* [16] and they all have rigorous mathematical semantics. Concepts such as containment, connectedness, overlap and adjacency as well as shape, size and colour are used to depict a system. Combining these techniques can trigger many useful mental images of what is going on in a system.

In Vista, only two-dimensional presentations have been used, and it would be interesting to see if a third dimension could enhance the effectiveness of the tool. Situations that currently use up too much screen space could be more effectively treated in three dimensions. Examples of this type of work exist [7][40][46]. Different conclusions can be drawn if the application domain is inherently non-graphic or inherently graphic. Animation has also been used to help understand systems [10][12][44] and would enhance visual composition. For example, a "data map" that shows where data is, how it is used and how it moves around the application could give a global picture of data usage. Data could be tagged and followed through the executing application.

10.7 Conclusion

The landscape of software is changing from monolithic closed applications to open applications composed of reusable components. As the landscape changes, end-users become application developers and application developers become component engineers. To support this new landscape, the software industry needs to promote the idea of investment in components. Among other things, this means developing repositories of components and tools for developing applications from components. Visual composition can lead to new tools and environments which would contribute to the fulfilment of our duty, as Harel puts it, "to forge ahead to turn system modeling into a predominately visual and graphical process."

Acknowledgements

The work described in this chapter has benefited from the efforts of many past and present members of the Object Systems Group. The prototype visual composition tool described here includes efforts by Betty Junod, Oscar Nierstrasz, Serge Renfer, Marc Stadelmann and Ino Simitsek. The precursor to Vista — VST — was implemented by Jan Vitek and Marc Stadelmann, with contributions from Gerti Kappel. The work on the multimedia framework includes efforts by Christian Breiteneder, Laurent Dami, Simon Gibbs, Michael Papathomas and Dennis Tsichritzis.

Much of this work was done in the context of the ITHACA project. Roberto Bellinzona of Politecnico di Milano worked on the RECAST sample application and Hayat Issad of IFATEC worked on the component set of the workflow sample application.

References

[1] Apple Computer, Inc., *Inside Macintosh: Interapplication Communication*, Addison-Wesley, Reading, Mass.

[2] Jeff Alger, “OpenDoc vs. OLE,” *MacTech Magazine*, vol. 10, no. 8, Aug. 1994, pp. 58–70.

[3] Roberto Bellinzona and Mariagrazia Fugini, “RECAST Prototype Description,” ITHACA.POLIMI.91.E.2.8.#1, Politecnico di Milano, Nov. 28, 1991.

[4] Roberto Bellinzona, Mariagrazia Fugini and Giampo Bracchi, “Scripting Reusable Requirements Through RECAST,” ITHACA.POLIMI.92.E.2.9.#1, Politecnico di Milano, July, 1992.

[5] Ted J. Biggerstaff and C. Richter, “Reusability Framework, Assessment and Directions,” *IEEE Software*, March 1987, pp. 41–49.

[6] Fred P. Brooks, “No Silver Bullet,” *IEEE Computer*, April 1987, pp. 10–19.

[7] Stuart Card, G. Robertson and J. Mackinlay, “The Information Visualizer: An Information Workspace,” *CHI ’91 Conference Proceedings*, New Orleans.

[8] Valeria De Antonellis, Barbara Pernici and P. Samarati, “F-ORM METHOD: A F-ORM Methodology for Reusing Specifications,” *IFIP WG 8.4 Working Conference on Object-Oriented Aspects in Information Systems*, Quebec, Oct. 1991.

[9] Valeria De Antonellis and Barbara Pernici, “ITHACA Object-Oriented Methodology Manual — Introduction and Application Developer Manual (IOOM/AD),” ITHACA.POLIMI.UDUNIV.91.E.8.1, Oct., 1991.

[10] Wim De Pauw, Richard Helm, Doug Kimelman and John Vlissides, “Visualizing the Behavior of Object-Oriented Systems,” in *Proceedings OOPSLA ’93, ACM SIGPLAN Notices*, vol. 28, no. 10, Oct. 1993, pp. 326–337.

[11] Erich Gamma, Richard Helm, JohnVlissides and Ralph E. Johnson, “Design Patterns: Abstraction and Reuse of Object-Oriented Design,” in *Proceedings ECOOP ’93*, ed. O. Nierstrasz, *Lecture Notes in Computer Science*, vol. 707, Springer-Verlag, Kaiserslautern, Germany, July 1993, pp. 406–431.

[12] Steven C. Glassman, “A Turbo Environment for Producing Algorithm Animations,” in *Proceedings IEEE Symposium on Visual Languages*, Aug. 1993, pp. 32–36.

[13] Adele Goldberg, “Information Models, Views and Controllers,” *Dr. Dobb’s Journal*, July, 1990.

[14] Paul E. Haeberli, ”ConMan: A Visual Programming Language for Interactive Graphics,” *ACM Computer Graphics*, vol. 22, no. 4, Aug. 1988, pp. 103–111.

[15] David Harel, “Biting the Silver Bullet,” *IEEE Computer*, vol. 25 no. 1, Jan., 1992, pp.8–20.

[16] David Harel, “On Visual Formalisms,” *Communications of the ACM*, vol. 31, no. 5, May 1988, pp. 514–530.

[17] William Harrison, Harold Ossher and Mansour Kavianpour, “Integrating Coarse-Grained and Fine-Grained Tool Integration,” *Proceedings CASE ’92*, July 1992.

[18] Richard Helm, Ian Holland and Dipayan Gangopadhyay, “Contracts: Specifying Behavioural Compositions in Object-Oriented Systems,” *Proceedings OOPSLA/ECOOP ’90, ACM SIGPLAN Notices*, vol. 25, no. 10, Oct. 1990, pp. 169–180.

[19] *HP Journal*, vol. 41, no. 3, June 1990 (HP SoftBench).

[20] IBM, VisualAge documentation and demo diskette.

[21] ITHACA Tecnical Annex, Sept. 1988.

[22] Dan Ingalls, “Fabrik: A Visual Programming Environment,” *ACM SIGPLAN Notices*, vol. 23, no. 11, Nov. 1988, pp. 176–190.

[23] Ivar Jacobson, “Is Object Technology Software’s Industrial Platform?” *IEEE Software*, vol. 10, no. 1, Jan. 1993, pp. 24–30.

[24] Ralph E. Johnson, “How to Design Frameworks,” OOPSLA ’93 tutorial notes.

[25] Ralph E. Johnson and Vincent F. Russo, “Reusing Object-Oriented Designs,” University of Illinois, TR UIUCDCS 91-1696.

[26] A. Julienne and L. Russell, “Why You Need ToolTalk,” *SunExpert Magazine*, vol. 4, no. 3, March 1993, pp. 51–58.

[27] Manolis Katevenis, T. Sorilos and P. Kalogerakis, “Laby Programmer’s Manual (version 3.0),” ITHACA report FORTH.92.E3.3.#1, Foundation of Research and Technology — Hellas, Iraklion, Crete, Jan. 1992.

[28] Karl Lieberherr and Ian Holland, “Assuring Good Style for Object-Oriented Programs,” *IEEE Software*, Sept. 89, pp. 38–48.

[29] Vicki de Mey, Betty Junod, Serge Renfer, Marc Stadelmann and Ino Simitsek, “The Implementation of Vista — A Visual Scripting Tool,” in *Object Composition*, ed. D. Tsichritzis, Centre Universitaire d’Informatique, University of Geneva, June 1991, pp. 31–56.

[30] Vicki de Mey, “Visual Composition of Software Applications,” Ph.D. thesis no. 2660, University of Geneva, 1994.

[31] Vicki de Mey, Christian Breiteneder, Laurent Dami, Simon Gibbs and Dennis Tsichritzis, “Visual Composition and Multimedia,” *Proceedings Eurographics ’92*.

[32] Vicki de Mey and Simon Gibbs, “A Multimedia Component Kit,” *Proceedings ACM Multimedia ’93*.

[33] Bertrand Meyer, “Applying ‘Design by Contract’,” *IEEE Computer*, Oct. 1992, pp. 40–51.

[34] Microsoft, Object Linking and Embedding Programmer’s Reference (pre-release), version 2, 1992.

[35] Naftaly H. Minsky and David Rozenshtein, “ A Law-Based Approach to Object-Oriented Programming,” *Proceedings OOPSLA ’87* , Oct. 1987, pp. 482–493.

[36] NeXT, Distributed Objects, release 3.1, 1993.

[37] Oscar Nierstrasz, Dennis Tsichritzis, Vicki de Mey and Marc Stadelmann, “Objects + Scripts = Applications,” *Proceedings, Esprit 1991 Conference*, Kluwer, Dordrecht, 1991, pp. 534–552.

[38] Andrew J. Palay, “Towards an ‘Operating System’ for User Interface Components,” in *Multimedia Interface Design*, ed. M. M. Blattner and R. B. Dannenberg, Frontier Series, ACM Press, 1992, pp. 339–355.

[39] Randy Pausch, Nathaniel R. Young and Robert DeLine, “SUIT: The Pascal of User Interface Toolkits,” *Proceedings of the Fourth Annual Symposium on User Interface Software and Technology*, Nov. 1991, pp. 117–125.

[40] Steven P. Reiss, “A Framework for Abstract 3D Visualization,” in *Proceedings IEEE Symposium on Visual Languages*, Aug. 1993, pp. 108–115.

[41] Lawrence J. Rosenblum and Gregory M. Nielson, “Guest Editors’ Introduction: Visualization Comes of Age,” *IEEE Computer Graphics and Applications*, vol. 11, no. 3, May 1991, pp. 15–17.

[42] Ben Shneiderman, “Direct Manipulation: A Step Beyond Programming Languages,” *IEEE Computer*, vol. 16, no. 8, Aug. 1983, pp. 57–69.

[43] David C. Smith and Joshua Susser, “A Component Architecture for Personal Computer Software,” in *Languages for Developing User Interfaces*, ed. B. Myers, Jones & Bartlett, 1992, pp. 31–56.

[44] Randall B. Smith, “Experiences with the Alternate Reality Kit: An Example of the Tension Between Literalism and Magic,” *IEEE Computer Graphics and Applications*, Sept. 1987, pp. 42–50.

[45] Marc Stadelmann, Gerti Kappel and Jan Vitek, “VST: A Scripting Tool Based on the UNIX Shell,” in *Object Management*, ed. D. Tsichritzis, Centre Universitaire d’Informatique, University of Geneva, July 1990, pp. 333–344.

[46] John T. Stasko and Joseph F. Wehrli, “Three-Dimensional Computation Visualization,” in *Proceedings IEEE Symposium on Visual Languages*, Aug. 1993, pp. 100–107.

[47] Dennis Tsichritzis and Simon Gibbs, “Virtual Museums and Virtual Realities” *Proceedings International Conference on Hypermedia & Interactivity in Museums, Archives and Museum Informatics,* Technical Report no. 14, Pittsburgh, Oct. 14–16, 1991, pp. 17–25.

[48] John Vlissides, “Generalized Graphical Object Editing,” Technical Report CSL-TR-90-427, Stanford University June 1990.